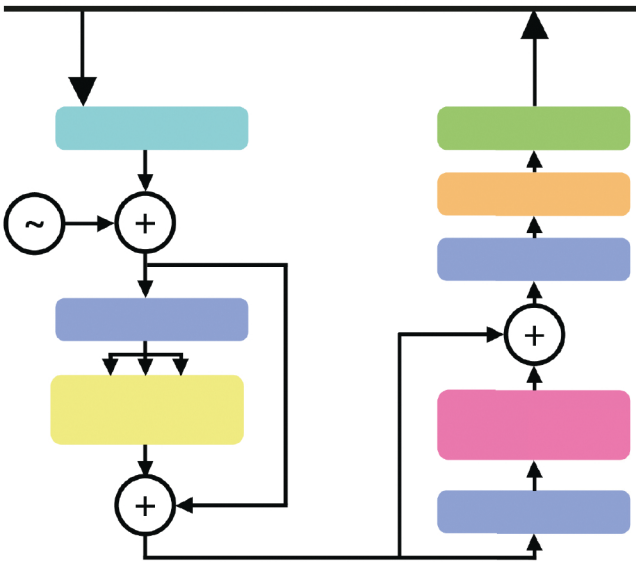
EXPERT INSIGHT

4178 596 44 29483

LLM на практике

49924



Большие языковые модели
от идеи до внедрения

Предисловие

Жюльен Шомон
соучредитель и технический директор Hugging Face

Хамза Тахир
соучредитель и технический директор ZenML

Антонио Гулли
старший директор Google



Пол Юстин
Максим Лабонн

SPRINT BOOK
<packt>

LLM Engineer's Handbook

Master the art of engineering Large Language Models from concept to production

Paul Iusztin
Maxime Labonne

packt

LLM на практике

Большие языковые модели
от идеи до внедрения

Пол Юстин
Максим Лабонн

SPRINT book 2026

Пол Юстин, Максим Лабонн

LLM на практике. Большие языковые модели от идеи до внедрения

Перевела с английского В. Корягина
Научный редактор А. Лашнёв

ББК 32.813+81.11
УДК 004.8+004.912

Юстин Пол, Лабонн Максим

Ю90 LLM на практике. Большие языковые модели от идеи до внедрения. — Астана: «Спринт Бук», 2026. — 480 с.: ил.

ISBN 978-601-14-1155-4

Искусственный интеллект стремительно развивается, и именно большие языковые модели (LLM) задают направление всей индустрии. Погрузитесь в процесс проектирования, обучения и развертывания LLM в реальных бизнес-задачах, опираясь на лучшие практики MLOps. Авторы шаг за шагом показывают, как создать экономичную, масштабируемую и модульную систему на основе LLM. Вместо простых экспериментов в изолированных блокнотах Jupyter научитесь строить комплексные системы, готовые к полноценной эксплуатации.

Вы узнаете об особенностях инженерии данных, тонкой настройки моделей и их развертывания, попробуете создать LLM-двойника и внедрить компоненты MLOps в свои проекты. Кроме того, вы познакомитесь с передовыми методами оптимизации инференса, выравниванием по предпочтениям и обработкой данных в реальном времени.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

Права на издание получены по соглашению с Packt Publishing. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-601-14-1155-4

© Packt Publishing 2024.
First published in the English language under the title ‘LLM Engineer’s Handbook – (9781836200079)’
© Перевод на русский язык ТОО «Спринт Бук», 2026
© Издание на русском языке, оформление ТОО «Спринт Бук», 2026

Изготовитель: ТОО «Спринт Бук». Место нахождения и фактический адрес:
010000, Казахстан, город Астана, район Алматы, проспект Ракымжан Кошкарбаев, д. 10/1, н. п. 18.

Дата изготовления: 07.2026. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 25.06.26. Формат 70×100/16. Бумага офсетная. Усл. п. л. 38,700. Тираж 700. Заказ 0000.

Краткое содержание

Предисловие 13
Об авторах 16
О научном редакторе 17
Введение 18
От издательства 24
О научном редакторе русскоязычного издания 25

Глава 1. Концепция и архитектура LLM-двойника 26
Глава 2. Инструменты и установка 53
Глава 3. Инженерия данных 82
Глава 4. RAG-пайплайн признаков 119
Глава 5. Тонкая настройка с учителем 191
Глава 6. Тонкая настройка с учетом предпочтений 244
Глава 7. Оценка LLM 273
Глава 8. Оптимизация инференса 300
Глава 9. RAG-пайплайн инференса 328
Глава 10. Развертывание пайплайна инференса 361
Глава 11. MLOps и LLMOps 405

Приложение. Принципы MLOps 463

Оглавление

Предисловие 13
Об авторах 16
О научном редакторе 17
Введение 18
Кому следует прочитать эту книгу 19
Структура издания 19
Как получить максимальную пользу от книги 21
Загрузка файлов с кодом из примеров 22
Загрузка цветных изображений 22
Условные обозначения 22
От издательства 24
О научном редакторе русскоязычного издания 25

Глава 1. Концепция и архитектура LLM-двойника 26
Понимание концепции LLM-двойника 27
Что такое LLM-двойник 27
Причины для создания LLM-двойника 28
Почему не стоит использовать ChatGPT (и подобные чат-боты) 30
Планирование MVP для LLM-двойника 31
Что такое MVP 31
Определение MVP для LLM-двойника 32
Построение ML-систем с использованием пайплайнов признаков, обучения и инференса 33
Проблема построения ML-систем 34
Проблема традиционных решений 35
Решение: ML-пайплайны для построения ML-систем 38
Преимущества FTI-архитектуры 42

Проектирование архитектуры системы LLM-двойника 43
Перечень технических требований к архитектуре LLM-двойника 43
Проектирование архитектуры LLM-двойника с помощью FTI-пайплайнов 45
Некоторые соображения по поводу архитектуры LLM-двойника и паттерна FTI 50
Резюме 51
Список используемых источников 52

Глава 2. Инструменты и установка 53
Экосистема Python и установка проекта 54
Poetry: управление зависимостями и виртуальным окружением 55
Poe the Poet: инструмент для выполнения команд 57
Инструменты MLOps и LLMOps 58
Hugging Face: реестр моделей 59
ZenML: оркестратор, артефакты и метаданные 60
Comet ML: система отслеживания экспериментов 72
Opik: мониторинг промптов 74
Базы данных для хранения неструктурированных и векторных данных 74
MongoDB: NoSQL-база данных 75
Qdrant: векторная база данных 75
Подготовка к работе с AWS 76
Настройка аккаунта AWS, ключ доступа и интерфейс командной строки 76
SageMaker: вычислительная платформа для обучения и инференса 78
Резюме 80
Список используемых источников 80

Глава 3. Инженерия данных 82
Проектирование пайплайна сбора данных для LLM-двойника 83
Реализация пайплайна сбора данных для LLM-двойника 87
Пайплайн ZenML и его шаги 88
Диспетчер: инициализация нужного краулера 91
Краулеры 94
Документы в хранилище данных NoSQL 104
Сбор исходных данных в хранилище 112
Устранение неполадок 116
Резюме 117
Список используемых источников 118

Глава 4. RAG-пайплайн признаков ... 119
Принципы работы механизма RAG ... 119
Причины для использования RAG ... 120
Базовая архитектура RAG ... 121
Что такое эмбеддинги ... 126
Векторные базы данных ... 134
Обзор продвинутой RAG-архитектуры ... 137
Этап подготовки запроса к поиску ... 138
Этап поиска ... 142
Этап работы с найденными документами перед генерацией ... 143
Архитектура RAG-пайплайна признаков для LLM-двойника ... 146
Решаемая задача ... 146
Хранилище признаков ... 147
Откуда поступают исходные данные ... 148
Проектирование архитектуры RAG-пайплайна признаков ... 148
Реализация RAG-пайплайна признаков для LLM-двойника ... 159
Класс Settings ... 159
Пайплайн и шаги ZenML ... 160
Доменные сущности Pydantic ... 169
Слой диспетчеризации ... 177
Обработчики ... 180
Резюме ... 188
Список используемых источников ... 189

Глава 5. Тонкая настройка с учителем ... 191
Создание набора данных с инструкциями ... 191
Процесс создания наборов данных с инструкциями ... 192
Подготовка данных ... 196
Фильтрация на основе правил ... 197
Удаление дубликатов ... 198
Очистка данных от пересечений ... 200
Оценка качества данных ... 200
Исследование данных ... 204
Генерация данных ... 206
Аугментация данных ... 208
Создание собственного набора данных с инструкциями ... 210

Тонкая настройка с учителем и ее методы ... 219
Когда стоит проводить тонкую настройку ... 219
Форматы наборов данных с инструкциями ... 220
Шаблоны ведения диалога ... 221
Эффективные методы тонкой настройки параметров ... 223
Гиперпараметры обучения ... 230
Тонкая настройка на практике ... 235
Резюме ... 242
Список используемых источников ... 242

Глава 6. Тонкая настройка с учетом предпочтений ... 244
Особенности наборов данных о предпочтениях ... 245
Данные о предпочтениях ... 245
Генерация и оценка данных ... 248
Создание собственного набора данных о предпочтениях ... 253
Согласование предпочтений ... 260
Обучение с подкреплением на основе обратной связи от человека ... 260
Прямая оптимизация предпочтений ... 262
Реализация метода DPO ... 265
Резюме ... 271
Список используемых источников ... 272

Глава 7. Оценка LLM ... 273
Оценка моделей ... 273
Оценка LLM и ML-моделей ... 274
Оценка LLM общего назначения ... 275
Оценка предметно-ориентированных LLM ... 277
Оценка узкоспециализированных LLM ... 280
Оценка RAG-систем ... 283
Инструментарий Ragas ... 284
Фреймворк ARES ... 286
Оценка модели TwinLlama-3.1-8B ... 288
Генерация ответов ... 288
Оценка ответов ... 290
Анализ результатов ... 294
Резюме ... 298
Список используемых источников ... 299

Глава 8. Оптимизация инференса ... 300
Стратегии оптимизации модели ... 301
Кэш ключей и значений ... 302
Непрерывная пакетная обработка ... 305
Спекулятивное декодирование ... 305
Оптимизированные механизмы внимания ... 308
Модельный параллелизм ... 309
Параллелизм данных ... 310
Конвейерный параллелизм ... 311
Тензорный параллелизм ... 312
Комбинирование подходов ... 314
Квантование модели ... 315
Принцип квантования весов ... 315
Квантование с использованием формата GGUF и библиотеки llama.cpp ... 320
Квантование с использованием форматов GPTQ и EXL2 ... 322
Прочие методы квантования ... 324
Резюме ... 325
Список используемых источников ... 326

Глава 9. RAG-пайплайн инференса ... 328
Устройство RAG-пайплайна инференса LLM-двойника ... 329
Продвинутые техники RAG, используемые в системе LLM-двойника ... 331
Оптимизация этапа подготовки к поиску: расширение запроса и самозапрос ... 334
Оптимизация этапа поиска: векторный поиск с фильтрацией ... 341
Оптимизация этапа работы с найденными документами: повторное ранжирование ... 343
Реализация RAG-пайплайна инференса для LLM-двойника ... 347
Реализация поискового модуля ... 347
Реализация RAG-пайплайна инференса ... 354
Резюме ... 359
Список используемых источников ... 359

Глава 10. Развертывание пайплайна инференса ... 361
Критерии выбора стратегии развертывания пайплайна инференса ... 362
Пропускная способность и задержка ... 362
Данные ... 363
Инфраструктура ... 363

Стратегии развертывания пайплайна инференса ... 365
Онлайн-инференс в режиме реального времени ... 366
Асинхронный инференс ... 367
Пакетная офлайн-обработка ... 368
Монолитная и микросервисная архитектура в контексте обслуживания модели ... 369
Монолитная архитектура ... 370
Микросервисная архитектура ... 371
Выбор между монолитной и микросервисной архитектурой ... 373
Стратегия развертывания пайплайна инференса для LLM-двойника ... 374
Развертывание сервиса LLM-двойника ... 378
Реализация LLM-микросервиса с использованием AWS SageMaker ... 379
Реализация бизнес-микросервиса на базе FastAPI ... 395
Автомасштабирование для обработки всплесков нагрузки ... 398
Регистрация масштабируемого сервиса ... 400
Создание политики масштабирования ... 400
Границы масштабирования ... 401
Резюме ... 403
Список используемых источников ... 403

Глава 11. MLOps и LLMOps ... 405
Истоки LLMOps в DevOps и MLOps ... 406
DevOps ... 406
MLOps ... 410
LLMOps ... 414
Развертывание пайплайнов LLM-двойника в облаке ... 420
Обзор инфраструктуры ... 420
Настройка MongoDB ... 423
Настройка Qdrant ... 424
Настройка ZenML Cloud ... 425
Добавление LLMOps-компонентов в систему LLM-двойника ... 437
CI/CD-пайплайн LLM-двойника ... 438
Краткий обзор платформы GitHub Actions ... 441
CI-пайплайн ... 442
CD-пайплайн ... 446
Тестирование CI/CD-пайплайнов ... 448
CT-пайплайн ... 450

Мониторинг промптов ... 455
Система оповещений ... 460
Резюме ... 460
Список используемых источников ... 461

Приложение. Принципы MLOps ... 463
1. Автоматизация, или эксплуатация ... 463
2. Версионирование ... 465
3. Отслеживание экспериментов ... 466
4. Тестирование ... 466
Типы тестов ... 466
Предмет тестирования ... 467
Примеры тестов ... 468
5. Мониторинг ... 470
Логи ... 470
Метрики ... 471
6. Воспроизводимость ... 476

Предисловие

Как часто говорит мой коллега и соучредитель компании Hugging Face Клеман Деланг (и я с ним полностью согласен), использование искусственного интеллекта (ИИ) становится новой нормой в сфере разработки.

За последние три года большие языковые модели (large language model, LLM) оказали глубокое влияние на развитие технологий, и в ближайшие пять лет оно только усилится. Они будут интегрироваться в еще большее число продуктов и, как мне кажется, займут центральное место во всех областях человеческой деятельности.

Так, разработчики уже используют LLM и перестраивают свой рабочий процесс, сосредотачиваясь на более сложных задачах, а рутинные действия делегируя машинам. Музыканты применяют ИИ-инструменты, чтобы быстрее исследовать пространство музыкального творчества. Юристы повышают результативность своей работы благодаря методам генерации, дополненной поиском, и обширным базам судебных решений.

В Hugging Face мы всегда выступали за то, чтобы модели ИИ не принадлежали одной компании или узкому кругу специалистов, а были доступны людям из самых разных слоев общества, независимо от их опыта.

Максим Лабонн и Пол Юстин сыграли ключевую роль в этом движении за демократизацию LLM. Написав эту книгу, они сделали все, чтобы как можно больше людей могли не просто применять эти модели, но и адаптировать их, настраивать, обучать и делать достаточно эффективными для практического использования.

Их работа имеет фундаментальное значение, и я рад, что они поделились своим опытом с сообществом, так как это действительно способствует расширению границ человеческого знания.

Жюльен Шомон,
соучредитель и главный технический
директор Hugging Face

Как человек, глубоко погруженный в сферу MLOps (Machine Learning Operations — непрерывная разработка систем машинного обучения), я искренне рекомендую к прочтению книгу «LLM на практике». Момент для выхода этой замечательной книги подобран идеально: спрос на специалистов по большим языковым моделям стремительно растет в самых разных отраслях.

Одним из достоинств книги является прикладной подход. Проводя читателей по всем стадиям создания LLM-двойника (цифрового двойника на основе ИИ), авторы помогают преодолеть разрыв между теорией и практикой. В издании рассматриваются все ключевые этапы работы с LLM — от инженерии данных и тонкой настройки модели до таких продвинутых тем, как пайплайны RAG и оптимизация инференса (того, как быстро и эффективно модель отвечает пользователю).

Мне особенно понравилось, что в книге много внимания уделяется принципам MLOps и LLMOps. По мере того как все больше организаций внедряет LLM, крайне важно понимать, как строить масштабируемые, воспроизводимые и надежные системы. Авторы не ограничились теорией — описание стратегий оркестрации и облачной интеграции подчеркивает их стремление дать читателям навыки, которые действительно нужны для работы в продакшене.

Вне зависимости от того, являетесь ли вы опытным специалистом по машинному обучению, желающим углубиться в работу с LLM, или программистом, стремящимся войти в эту быстро развивающуюся область, прочитав эту книгу, вы получите фундаментальные знания и познакомитесь с передовыми практиками. Четкие объяснения, практические примеры и акцент на проверенных подходах делают ее ценным ресурсом для всех, кто всерьез намерен овладеть техниками работы с LLM.

В эпоху, когда ИИ радикально преобразует многие отрасли, «LLM на практике» служит надежным проводником в мир больших языковых моделей. Это не просто книга — это карта, помогающая ориентироваться в современном ИИ-ландшафте.

Хамза Тахир,
соучредитель и технический директор
ZenML

Книга «LLM на практике» — это ценный ресурс для всех, кто стремится глубже понять устройство больших языковых моделей. Опираясь на подробный разбор LLM-двойника и множество наглядных примеров, авторы объясняют ключевые этапы создания и развертывания LLM-систем промышленного уровня.

Одной из отличительных особенностей книги является использование проекта создания LLM-двойника в качестве сквозного примера. Этот персонаж с искусственным интеллектом, имитирующий стиль письма конкретного человека, наглядно демонстрирует способ применения LLM для решения реальных задач.

Авторы знакомят читателей с основными инструментами и технологиями, необходимыми для разработки систем на основе LLM, начиная с Hugging Face, ZenML и Comet и заканчивая Opik, MongoDB и Qdrant. Каждый инструмент подробно и доступно описан, что помогает легко понять его назначение и место в пайплайне LLM.

Книга также охватывает широкий спектр тем, связанных с LLM-разработкой: сбор данных, обучение модели, оценку, оптимизацию инференса и принципы MLOps. Особенно стоит отметить главы, посвященные тонкой настройке, выравниванию по предпочтениям и генерации, дополненной поиском, которые дают глубокое и ясное понимание критически важных этапов разработки LLM.

Сильной стороной книги также является ее ориентированность на практику: авторы превосходно описали разные способы оптимизации пайплайнов инференса и эффективного внедрения модели в продакшен. За счет этого она становится ценным пособием как для исследователей, так и для практиков.

Я рекомендую книгу всем, кто интересуется большими языковыми моделями и их применением в реальном мире. Благодаря исчерпывающему обзору инструментов, методов разработки и лучших практик данное руководство способно стать настольной книгой для многих LLM-инженеров.

Антонио Гулли,
старший директор Google

Об авторах

Пол Юстин — ведущий ML- и MLOps-инженер с более чем семилетним опытом разработки решений в сферах генеративного ИИ, компьютерного зрения и MLOps. Его последним местом работы была компания Metaphysic, в которой он занимался внедрением в производство крупных нейронных сетей. Ранее он работал в CoreAI, Everseen и Continental. Пол также является основателем образовательного канала Decoding ML, посвященного практическому машинному обучению, в рамках которого он публикует статьи, посты и открытые курсы, помогающие инженерам создавать реальные ML-системы.

Максим Лабонн — руководитель направления постобучения в компании Liquid AI. Он защитил докторскую диссертацию по машинному обучению в Политехническом институте Парижа и признан Google экспертом в области искусственного интеллекта и машинного обучения. Будучи активным блогером, внес заметный вклад в сообщество открытого ПО. Среди его разработок — курс по LLM на GitHub, такие инструменты, как LLM AutoEval, и несколько передовых моделей, включая NeuralDaredevil. Максим также является автором бестселлера *Hands-On Graph Neural Networks Using Python*[1] (Packt, 2023).

Я хочу поблагодарить свою семью за их неизменную поддержку и терпение.

[1] *Лабонн М.* Графовые нейронные сети на Python. — СПб., 2024.

О научном редакторе

Рани Эль-Хусейни — архитектор ИИ-решений и руководитель инженерных команд с более чем двадцатилетним опытом работы в сферах искусственного интеллекта, обработки естественного языка (Natural Language Processing, NLP) и машинного обучения. На протяжении всей карьеры занимался разработкой и внедрением ИИ-моделей, а также публиковал статьи, посвященные архитектуре ИИ-систем и этичным практикам их внедрения. Он возглавлял работу над проектами в таких компаниях, как Microsoft, где руководил развитием технологий NLP и сервиса Language Understanding Intelligent Service (LUIS). В настоящее время занимает ведущую позицию в компании Clearwater Analytics, где отвечает за внедрение инноваций в сфере генеративного искусственного интеллекта и разработку ИИ-решений для управления финансами и инвестициями.

Я хочу поблагодарить компанию Clearwater Analytics за создание среды, способствующей росту и инновациям. Видение наших руководителей, всегда стремящихся быть на шаг впереди и следящих за новейшими технологиями, является для меня настоящим источником вдохновения. Благодаря их приверженности развитию ИИ мой опыт рецензирования этой книги оказался по-настоящему содержательным и обогащающим. Особую благодарность я выражаю своей семье, которая поддерживала меня на этом пути.

Введение

Область инженерии *больших языковых моделей* (large language model, LLM) быстро превратилась в ключевое направление в сфере искусственного интеллекта и машинного обучения. Активное внедрение LLM в области обработки и генерации естественного языка порождает спрос на специалистов, способных эффективно реализовывать, оптимизировать и применять эти модели на практике. Инженерия LLM охватывает широкий спектр дисциплин — от подготовки данных и тонкой настройки (fine-tuning) моделей до оптимизации инференса и развертывания в продакшене, требуя уникального сочетания навыков в сфере программной инженерии, машинного обучения и знаний предметной области.

Непрерывная разработка систем машинного обучения (Machine Learning Operations, MLOps) играет решающую роль в успешной реализации LLM в продакшене. Практика MLOps распространяет принципы DevOps на проекты ML и делает акцент на автоматизации и упрощении жизненного цикла моделей. В случае с большими языковыми моделями подход MLOps особенно важен ввиду сложности и масштабности больших языковых моделей. Он позволяет решать задачи, связанные с управлением крупными наборами данных (датасетами), версионированием моделей, обеспечением воспроизводимости результатов и поддержанием стабильного уровня производительности. Применение MLOps-практик повышает эффективность, надежность и масштабируемость LLM, что в конечном счете способствует их более успешному внедрению.

Книга «LLM на практике» поможет читателям овладеть лучшими практиками в области инженерии LLM. В ней приводятся упрощенные объяснения ключевых концепций, описания практических методов и советы экспертов, которые касаются различных этапов жизненного цикла LLM. Здесь также рассматриваются такие темы, как подготовка данных, тонкая настройка, оценка модели, оптимизация инференса и разработка пайплайна RAG (Retrieval-Augmented Generation — генерация, дополненная поиском).

Для иллюстрации описываемых концепций в книге будет пошагово реализован так называемый LLM-двойник — система, имитирующая стиль письма и лич-

ность конкретного человека. На его примере авторы покажут, как средствами LLM-инженерии и MLOps-практик можно создать минимально жизнеспособный продукт для решения практической задачи.

Вы узнаете о том, как собирать и подготавливать данные для LLM, настраивать модели под конкретные задачи, оптимизировать производительность инференса и реализовывать пайплайны RAG. Вы также научитесь оценивать эффективность работы LLM, согласовывать поведение моделей с человеческими предпочтениями и внедрять LLM-приложения. Кроме того, в книге подробно изложены ключевые принципы и практики MLOps, позволяющие создавать масштабируемые, воспроизводимые и устойчивые решения на основе больших языковых моделей.

Кому следует прочитать эту книгу

Книга ориентирована на широкий круг специалистов в области технологий, а также на энтузиастов, интересующихся практическими аспектами использования больших языковых моделей. Она будет особенно полезна разработчикам, планирующим начать работу над проектами в сфере искусственного интеллекта. Хотя базовые знания в области разработки ПО могут оказаться полезными, книга содержит пояснения многих ключевых концепций, что делает ее доступной даже для тех, кто только начинает знакомство с ИИ и машинным обучением.

Читателям, уже применяющим методы машинного обучения, книга позволит углубить навыки реализации и развертывания систем на основе LLM. В ней подробно рассматриваются основы MLOps, а также описывается пошаговый процесс создания минимально жизнеспособного продукта с использованием LLM с открытым исходным кодом.

Структура издания

Глава 1 «Концепция и архитектура LLM-двойника» знакомит читателей с проектом, который используется в книге в качестве сквозного примера разработки LLM-приложения, готового к развертыванию в производственной среде. В этой главе также описывается архитектура FTI (feature/training/inference), предназначенная для построения масштабируемых ML-систем, и демонстрируется ее применение в контексте разработки LLM-двойника.

Глава 2 «Инструменты и установка» посвящена языку Python, MLOps и облачным инструментам, которые используются для создания реальных LLM-приложений. Среди них — оркестраторы, системы отслеживания экспериментов,

инструменты мониторинга промптов и оценки LLM. В этой главе показано, как установить и использовать эти инструменты в локальной среде для тестирования и разработки.

Глава 3 «Инженерия данных» знакомит читателей с процессом реализации пайплайна, собирающего данные из нескольких источников, таких как Medium, GitHub и Substack, и помещающего их в хранилище. При этом акцент делается на сборе сырых данных из динамических источников, а не на использовании статических наборов данных, поскольку такая стратегия является более актуальной в контексте обсуждения реальных ML-приложений.

Глава 4 «RAG-пайплайн признаков» охватывает понятия, связанные с генерацией, дополненной поиском, включая эмбеддинги, базовую архитектуру RAG, векторные базы данных и способы оптимизации RAG-приложений. Далее рассматривается архитектура и процесс реализации RAG-пайплайна признаков для LLM-двойника с использованием передовых практик разработки ПО.

Глава 5 «Тонкая настройка с учителем» содержит описание процесса тонкой настройки предварительно обученных языковых моделей для решения конкретных задач с помощью пар «инструкция — ответ». В ней обсуждаются методы создания высококачественных наборов данных, такие техники, как полная тонкая настройка, LoRA и QLoRA, а также приводится пример настройки модели Llama-3.1-8B на пользовательском наборе данных.

Глава 6 «Тонкая настройка с учетом предпочтений человека» посвящена методам выравнивания языковых моделей на основе человеческих предпочтений с акцентом на *прямую оптимизацию предпочтений* (Direct Preference Optimization, DPO). В ней также описывается процесс создания пользовательских наборов данных о предпочтениях, реализации метода DPO и тонкой настройки модели TwinLlama-3.1-8B с помощью библиотеки Unsloth.

Глава 7 «Оценка LLM» содержит подробное описание различных методов оценки производительности языковых моделей и систем на их основе. В ней представлены универсальные и специализированные способы оценки, обсуждаются популярные бенчмарки и демонстрируется процесс оценки модели TwinLlama-3.1-8B по нескольким критериям.

Глава 8 «Оптимизация инференса» охватывает ключевые стратегии оптимизации, такие как спекулятивное декодирование, параллелизм и квантование весов модели. В ней обсуждаются способы ускорения инференса, уменьшения задержек и минимизации объема используемой памяти, а также описываются и сравниваются популярные движки для инференса.

Глава 9 «RAG-пайплайн инференса» посвящена таким продвинутым техникам RAG, как реализация самозапросов, повторное ранжирование и векторный по-

иск с фильтрацией. В ней описывается процесс проектирования и реализации RAG-пайплайна инференса для LLM-двойника и пользовательского модуля извлечения, аналогичного тем, что используются в популярных фреймворках, вроде LangChain.

Глава 10 «Развертывание пайплайна инференса» знакомит читателей с такими стратегиями развертывания моделей машинного обучения, как асинхронный, пакетный и онлайн-инференс, которые применяются при проектировании и развертывании дообученной модели LLM-двойника на платформе AWS SageMaker, а также при создании микросервиса на базе FastAPI для предоставления доступа к пайплайну RAG как к интерфейсу RESTful API.

Глава 11 «MLOps и LLMOps» посвящена практике LLMOps, начиная с ее истоков в DevOps и MLOps. В этой главе описывается процесс развертывания LLM-двойника в облаке, включая перенос пайплайнов машинного обучения на платформу AWS, а также способы контейнеризации кода с помощью Docker и построения пайплайна CI/CD/CT (continuous integration/deployment/training). Кроме того, в ней описан процесс добавления слоя мониторинга промптов поверх пайплайна инференса LLM-двойника.

Приложение «Принципы MLOps» содержит описание шести принципов MLOps, позволяющих создавать масштабируемые, воспроизводимые и надежные ML-приложения.

Как получить максимальную пользу от книги

Чтобы извлечь максимум из книги, желательно иметь хотя бы базовое понимание принципов и практик разработки ПО. Знание языка программирования Python будет особенно полезным, поскольку большинство примеров кода в книге написано на нем. Наличие опыта применения концепций машинного обучения приветствуется, но не является обязательным, так как многие ключевые идеи из области ИИ и машинного обучения подробно объясняются по ходу изложения. Вместе с тем вы должны уметь работать с базовыми структурами данных и алгоритмами, а также иметь некоторый опыт взаимодействия с API и облачными сервисами.

Предполагается, что читатель знаком с такими системами контроля версий, как Git, поскольку примеры кода размещены в репозитории на GitHub. Хотя книга рассчитана и на тех, кто только начинает знакомство с ИИ и LLM, наличие предварительной подготовки в этих областях облегчит освоение наиболее сложных концепций и приемов, которые в ней описываются.

Загрузка файлов с кодом из примеров

Код из примеров, приведенных в книге, опубликован на GitHub по адресу https://github.com/PacktPublishing/LLM-Engineers-Handbook. Вы также можете найти другие коллекции исходного кода из нашего обширного каталога книг и видеокурсов по адресу https://github.com/PacktPublishing/.

Загрузка цветных изображений

PDF-файл с цветными версиями снимков экрана и схем, приведенных в оригинальной книге, можно скачать по ссылке https://packt.link/gbp/9781836200079.

Условные обозначения

В книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины и важные понятия.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, типы данных, переменные окружения, операторы и ключевые слова, имена папок, каталогов, путей, файлов и их расширений. Например: «В функции `format_samples` мы применяем к промпту шаблон ведения диалога Alpaca».

Блок кода оформляется следующим образом:

```
def format_samples(example):
    example["prompt"] = alpaca_template.format(example["prompt"])
    example["chosen"] = example['chosen'] + EOS_TOKEN
    example["rejected"] = example['rejected'] + EOS_TOKEN
    return {"prompt": example["prompt"], "chosen": example["chosen"],
    ➥"rejected": example["rejected"]}
```

Символом переноса строки (➥) обозначается код, который не поместился на строке.

Когда требуется привлечь внимание читателя к определенной части блока кода, соответствующие строки или элементы выделяются серым цветом:

```
def format_samples(example):
    example["prompt"] = alpaca_template.format(example["prompt"])
    example["chosen"] = example['chosen'] + EOS_TOKEN
    example["rejected"] = example['rejected'] + EOS_TOKEN
    return {"prompt": example["prompt"], "chosen": example["chosen"],
    ➥"rejected": example["rejected"]}
```

Любой ввод или вывод командной строки оформляется следующим образом:

```
poetry install --without aws
```

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты и элементов интерфейса. Например: «Для этого перейдите на вкладку Settings (Настройки) в верхней части ответвленного репозитория на GitHub. В меню слева в разделе Security (Безопасность) раскройте пункт Secrets and variables (Секреты и переменные), а затем выберите Actions (Действия)».



Так оформляется примечание.

От издательства

Мы выражаем огромную благодарность клубу рецензентов ИТ-литературы ReadIT Club за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@sprintbook.kz (издательство SprintBook, компьютерная редакция).

Мы будем рады узнать ваше мнение!

О научном редакторе русскоязычного издания

Алексей Лашнёв — разработчик на Java/Kotlin в одном из крупнейших банков России. В ИТ 15 лет — в основном в финтехе. Заядлый путешественник, посетивший 77 стран, писатель на Хабре и просто неравнодушный к коду и архитектуре человек.

Книга дает хорошее представление о том, как дообучать нейросети и какие технологии в целом используются для работы в LLM-домене. В ней рассматривается простая, но релевантная задача, на которой можно показать все этапы тонкой настройки.

Примеры скриптов в книге написаны на Python, но они достаточно простые и содержат множественные комментарии, чтобы инженеры, не владеющие этим языком, могли понять логику.

Кое-где авторы углубляются в работу нейросетей на уровне математики. Надо отметить, что некоторые математические выкладки сложно понять, даже имея высшее физико-математическое образование.

На мой взгляд, книга подходит ИТ-инженерам, которые имеют опыт работы с нейросетями на профессиональном уровне.

1 Концепция и архитектура LLM-двойника

Мы твердо убеждены в том, что лучший способ понять работу *больших языковых моделей* (large language model, LLM) и принципы промышленного *машинного обучения* (machine learning, ML) — это не ограничиваться теорией, а создавать реальные системы. В книге мы покажем, как построить LLM-двойника — интеллектуального агента, способного писать в стиле конкретного человека, воспроизводя его манеру, голос и индивидуальность. На этом примере мы последовательно рассмотрим все стадии жизненного цикла ML-системы — от сбора данных до развертывания и мониторинга. Большинство концепций, которые вы освоите при реализации LLM-двойника, будут применимы и в других проектах, основанных на LLM или машинном обучении.

С инженерной точки зрения перед реализацией нового продукта необходимо пройти три этапа планирования. В первую очередь важно понять, какую задачу мы решаем и что именно собираемся создавать. В нашем случае следует определиться с тем, что представляет собой LLM-двойник и зачем он нужен. На этом шаге мы сосредотачиваемся на вопросе «Зачем?».

На втором этапе мы должны спроектировать первую версию продукта с минимальной функциональностью, чтобы приблизить процесс разработки к реальным условиям. Здесь нам нужно четко определить ключевые функции, необходимые для создания работоспособного и полезного продукта. Выбор зависит от сроков, доступных ресурсов и уровня подготовки команды. Таким образом, мы должны ответить на вопрос «Что именно мы собираемся построить?».

На третьем этапе мы проектируем систему, определяя ее архитектуру и принимая ключевые инженерные решения, лежащие в основе приложения на базе LLM. Обратите внимание: первые две стадии ориентированы на сам продукт, а третья имеет техническую направленность и отвечает на вопрос «Как?».

Эти три этапа естественным образом встраиваются в процесс создания реального продукта. Хотя первый и второй не имеют непосредственного отношения

к области машинного обучения, их необходимо пройти, чтобы разобраться в том, *как* строить продукт.

К концу главы у вас сложится ясное представление о том, что именно вы будете создавать при работе с книгой.

Понимание концепции LLM-двойника

Первым делом необходимо сформировать четкое видение того, что именно мы хотим создать и почему это ценно. Концепция LLM-двойника является новой, и, прежде чем переходить к техническим деталям, важно понять, что представляет собой этот двойник, чего от него следует ожидать и как он должен работать. Осознание конечной цели существенно облегчает усвоение теории, кода и инфраструктурных решений, представленных в книге.

Что такое LLM-двойник

Если в двух словах, то LLM-двойник — это интеллектуальный агент, встраивающий ваш стиль письма, голос и индивидуальность в большую языковую модель, то есть в сложную ИИ-модель. Можно сказать, что это цифровая *проекция* вашей личности. В отличие от универсальной модели, обученной на содержимом Интернета, LLM-двойник тонко настраивается на данных о вас самих. Поскольку любая ML-модель отражает те данные, на которых она обучалась, LLM-двойник будет воспроизводить ваш стиль письма, голос и особенности личности. Мы сознательно используем здесь слово «проекция». Любая проекция сопряжена с потерей значительной части информации, поэтому созданная модель не станет *вами*, но будет воспроизводить тот аспект вашей личности, который представлен в обучающих данных.

Важно понимать, что LLM отражает обучающие данные. Если вы «скормите» ей произведения Шекспира, она начнет писать в его манере. Если обучите модель на песнях Билли Айлиш, она будет создавать тексты, похожие на ее творчество. Эта концепция, известная как перенос стиля, широко используется и при генерации изображений, например, когда вы хотите изобразить кошку в стиле Ван Гога. Мы тоже применим эту стратегию, только стиль будет не заимствованным, а нашим собственным.

Для адаптации LLM к определенному стилю мы не только выполним тонкую настройку, но и задействуем различные продвинутые методы *генерации, дополненной поиском* (retrieval-augmented generation, RAG), используя эмбеддинги, чтобы направлять процесс авторегрессии сквозь данные о нас самих.

Методы тонкой настройки (доообучения) мы подробно рассмотрим в главе 5, а подходы RAG — в главах 4 и 9, а пока приведем несколько примеров, проясняющих изложенные выше идеи.

Вот на чем можно дообучить LLM, чтобы сделать из нее своего двойника:

- *публикации в социальных сетях* — настройка LLM на создание контента для социальных сетей;
- *переписка с друзьями и членами семьи* — адаптация LLM к вашему неформальному стилю общения;
- *научные статьи и публикации* — калибровка LLM для написания формального и образовательного контента;
- *программный код* — настройка LLM на генерацию кода в вашей манере.

Все эти сценарии сводятся к одной и той же базовой стратегии: собрать ваши цифровые данные и «скормить» их LLM с помощью соответствующих алгоритмов. В итоге модель будет отражать стиль, характерный для представленных данных. На первый взгляд все просто.

Но, к сожалению, здесь возникают как технические, так и этические вопросы. Как получить доступ к нужным данным? Достаточно ли у нас цифровых следов для создания LLM-двойника? Какие типы данных действительно полезны? Правильно ли вообще создавать такую копию себя? Хотим ли мы, чтобы кто-то — или что-то — подражал нам? Будет ли модель действительно использовать наш стиль или просто имитировать его?

Помните: цель этого раздела — ответить на вопрос «Зачем?», а не «Что?» и «Как?». Поэтому давайте обсудим, в чем ценность LLM-двойника и при каких условиях его создание может быть этически оправданно.

Причины для создания LLM-двойника

Для инженера (как и для любого другого специалиста) создание личного бренда зачастую оказывается более ценным, чем создание обычного резюме. Однако основная трудность заключается в том, что написание материалов для профильных площадок отнимает много времени. Даже если вам нравится писать и делиться идеями, рано или поздно вы столкнетесь с нехваткой времени или вдохновения и почувствуете, что вам требуется помощник. Мы не стремимся превратить этот раздел в рекламу, а лишь проясняем масштаб и задачи проекта, который нам предстоит реализовать.

Мы хотим создать LLM-двойника, который сможет писать персонализированный контент для социальных сетей, используя ваш уникальный стиль. Этот инструмент не предполагается использовать в неэтичных целях — он будет выполнять роль соавтора и помощника в рамках письменной коммуникации.

На основе описанных в книге техник вы сможете самостоятельно адаптировать модель для решения других задач, однако в рамках рассматриваемого примера мы сосредоточимся на генерации контента для социальных сетей и блогов. Благодаря этому вместо написания текста с нуля вы сможете передать LLM-двойнику основную идею и позволить ему сделать за вас всю рутинную работу.

В конечном счете нам все равно придется проверять, все ли работает корректно, и преобразовывать результаты в желаемый формат (подробнее об этом — в разделе «Планирование MVP для LLM-двойника» далее). Таким образом, мы проецируем себя в LLM-двойника — помощника, который будет автоматизировать наш процесс написания текстов. Эта конкретная LLM, скорее всего, окажется неэффективной в других сценариях, потому что мы специально адаптируем ее под одну задачу с помощью методов тонкой настройки, промпт-инжиниринга и RAG.

Итак, создание LLM-двойника поможет вам:

- выстраивать личный бренд;
- автоматизировать процесс написания текстов;
- генерировать новые творческие идеи.



Чем ассистент отличается от LLM-двойника

Цифровой ассистент и цифровой двойник — это разные понятия, сочетание которых образует по-настоящему мощное решение.

- Ассистент — это ИИ-инструмент, поддерживающий пользователя при выполнении задач, связанных с программированием, написанием текстов или созданием контента.
- Двойник — это цифровое представление реального объекта, часто использующее методы ИИ для преодоления разрыва между физическим и цифровым мирами. Например, LLM-двойник — это языковая модель, способная воспроизводить ваш голос, стиль письма и особенности личности.

С учетом этих определений можно сказать, что LLM-двойник выполняет роль ассистента, способного генерировать тексты в вашем стиле.

Важно подчеркнуть, что создание LLM-двойника вполне этично. Мы тонко настраиваем модель исключительно на собственных цифровых данных, не используя чужую информацию и не пытаясь имитировать чью-либо личность. Наша цель — создать персонализированную модель, подражающую нашему стилю письма. Таким образом, у каждого человека будет свой LLM-двойник с ограниченным доступом.

Конечно, существует множество вопросов, связанных с безопасностью, но мы не будем углубляться в них, так как эта тема заслуживает отдельной книги.

Почему не стоит использовать ChatGPT (и подобные чат-боты)



В этом разделе речь пойдет об использовании ChatGPT (или другого подобного чат-бота) исключительно в контексте генерации персонализированного контента.

Ответ на этот вопрос уже был дан: ChatGPT не *адаптирован* под ваш стиль письма и манеру изложения. Он генерирует текст, который кажется слишком общим, многословным и лишенным выразительности. Сохранение индивидуального стиля имеет решающее значение для долгосрочного успеха при формировании собственного бренда. Поэтому наличие ChatGPT или Gemini в вашем проекте не поспособствует достижению оптимального результата. Даже если вы готовы публиковать неперсонализированный контент, бездумное использование ChatGPT может привести к возникновению следующих проблем.

- *Искажение фактов из-за галлюцинаций.* Ручная проверка вывода или верификация текста с помощью сторонних инструментов утомительна и малоэффективна.
- *Монотонная ручная генерация промптов.* Самостоятельно формулировать запросы и внедрять в них внешнюю информацию тоже довольно утомительно, а получаемые ответы невозможно воспроизводить в точности, так как вы не контролируете содержимое промпта и структуру подаваемых данных. Частично эту проблему можно решить с помощью API и инструментов вроде LangChain[1], но это требует навыков программирования.

Наш опыт подсказывает, что если вы хотите создавать действительно качественный контент с реальной ценностью, то на проверку и правку сгенерированного текста вы потратите больше времени, чем на его написание с нуля.

Суть LLM-двойника определяется тем:

- какие данные мы собираем;
- как мы их обрабатываем;
- как подаем их на вход LLM;
- как выстраиваем цепочки промптов для получения нужного результата;
- как оцениваем сгенерированный контент.

---

[1] LangChain — это фреймворк для разработки приложений на базе LLM, предоставляющий инструменты для интеграции моделей с внешними источниками данных, API и последовательными пайплайнами обработки. — *Примеч. науч. ред.*

Разумеется, сама LLM тоже имеет значение. Однако стоит подчеркнуть, что использование веб-интерфейса ChatGPT крайне неудобно в плане управления данными, интеграции различных источников и оценки результатов. Решением этой проблемы является создание системы на базе LLM, которая инкапсулирует и автоматизирует все перечисленные ниже этапы работы (каждый раз выполнять их вручную — неэффективно и нецелесообразно в долгосрочной перспективе):

- сбор данных;
- предварительную обработку данных;
- хранение, версионирование и извлечение данных;
- тонкую настройку (дообучение) модели;
- генерацию, дополненную поиском (RAG);
- оценку сгенерированного контента.

При этом мы не утверждаем, что не стоит пользоваться API от OpenAI. Мы просто хотим подчеркнуть, что представленная в книге архитектура не будет зависеть от конкретной модели. Это означает, что, если модель можно использовать программно и она предоставляет интерфейс для тонкой настройки, значит, ее можно интегрировать в систему LLM-двойника. Ключ к успеху при создании ML-продуктов — быть ориентированными на данные и делать архитектуру независимой от конкретной модели. Тогда вы сможете легко экспериментировать с различными моделями на собственных данных.

Планирование MVP для LLM-двойника

Теперь, когда мы разобрались, что такое LLM-двойник и зачем его создавать, необходимо определиться с его функционалом. В книге мы сосредоточимся на первом этапе жизненного цикла, то есть на создании *минимально жизнеспособного продукта* (Minimum Viable Product, MVP). Основная задача здесь состоит в том, чтобы согласовать наше видение с реалистичными и достижимыми бизнес-целями, учитывая имеющиеся ресурсы. По мере роста инженеру необходимо проходить через этот этап, чтобы устранить разрыв между потребностями бизнеса и его техническими возможностями.

Что такое MVP

MVP — это версия продукта с минимальным набором функций, достаточным для привлечения первых пользователей и проверки жизнеспособности идеи на раннем этапе разработки. Обычно цель создания MVP — получить обратную связь от потенциальных потребителей с минимальными затратами.

Преимуществами стратегии разработки MVP являются:

- *быстрый выход на рынок* — максимально быстрый запуск продукта и привлечение первых пользователей;
- *проверка идеи* — тестирование продукта с участием реальных пользователей до начала его полномасштабной разработки;
- *исследование рынка* — получение данных о потребностях целевой аудитории;
- *снижение рисков* — минимизация затрат времени и ресурсов на разработку потенциально неуспешного продукта.

На этом этапе крайне важно создать именно *жизнеспособный* продукт, который предоставляет полноценный пользовательский опыт даже при реализации ограниченной функциональности. Он должен быть настолько работоспособным и удобным, чтобы людям захотелось продолжать им пользоваться и наблюдать за его развитием.

Определение MVP для LLM-двойника

Давайте проведем мысленный эксперимент и представим, что мы создаем не учебный проект из книги, а реальный продукт. Какие ресурсы есть в нашем распоряжении? Увы, их совсем немного:

- команда из трех человек — двух ML-инженеров и одного исследователя;
- наши ноутбуки;
- личные средства для оплаты вычислительных ресурсов (например, для обучения моделей);
- энтузиазм.

Как видите, наши ресурсы весьма ограниченны. И хотя это гипотетическая ситуация, она отражает реальность, с которой сталкивается большинство стартапов на начальном этапе своей деятельности. Поэтому мы должны стратегически подойти к определению MVP для LLM-двойника и к выбору функций, которые нам следует реализовать. Наша цель — максимально увеличить ценность продукта относительно вложенных усилий и ресурсов.

Чтобы упростить задачу, мы сосредоточимся на реализации таких функций LLM-двойника, как:

- сбор данных из пользовательских профилей в различных социальных сетях и на платформах вроде Substack и GitHub;
- тонкая настройка LLM с открытым исходным кодом на основе этих данных;

- заполнение векторной базы данных (БД) цифровыми материалами для их последующего использования в рамках RAG;
- создание публикаций в соцсетях с использованием:
  - пользовательских промптов;
  - механизма RAG (для повторного использования старого контента);
  - новых публикаций, статей и документов (в качестве дополнительного контекста для генерации ответов);
- наличие простого веб-интерфейса, позволяющего пользователю:
  - указать ссылки на свои соцсети и запустить процесс сбора данных;
  - отправлять промпты или ссылки на внешние ресурсы.

Это и будет MVP для нашего LLM-двойника. И пусть его функциональность ограничена, главное — чтобы система была экономически эффективной, масштабируемой и модульной.



Даже если мы сосредоточимся лишь на базовых функциях LLM-двойника, определенных в этом разделе, продукт будет разрабатываться с учетом современных исследований в области LLM, а также лучших практик программной инженерии и MLOps, поскольку наша цель заключается в демонстрации процесса проектирования экономически эффективного и масштабируемого приложения на базе LLM.

До этого момента мы рассматривали LLM-двойника с позиции пользователя и бизнеса. Теперь пришло время взглянуть на него с инженерной точки зрения и наметить план его разработки. Далее в книге мы сосредоточимся на технических аспектах реализации LLM-двойника.

Построение ML-систем с использованием пайплайнов признаков, обучения и инференса

Прежде чем переходить к обсуждению архитектуры LLM-двойника, необходимо рассмотреть лежащий в ее основе паттерн, называемый *FTI-архитектурой* (feature/training/inference — «признаки/обучение/инференс»). В этом разделе представлен общий обзор FTI-пайплайнов и методов их использования при построении ML-приложений.

Давайте посмотрим, как можно использовать FTI-пайплайны при разработке архитектуры LLM-двойника.

Проблема построения ML-систем

Создание ML-систем, готовых к развертыванию в продакшене, не ограничивается обучением модели. С технической точки зрения само обучение является одним из простейших этапов в большинстве случаев. Главная сложность при этом заключается в выборе правильной архитектуры и гиперпараметров[1] — но это уже исследовательская, а не инженерная проблема.

Мы же сосредоточимся на проектировании архитектуры, готовой к реальному использованию. Создание высокоточной модели, безусловно, важно, но ее обучения на статическом наборе данных недостаточно. Перед развертыванием системы нам необходимо подумать о том, как:

- загружать, очищать и осуществлять валидацию новых данных;
- разграничивать процессы обучения и инференса;
- вычислять признаки[2] и сохранять их в соответствующей среде;
- развертывать и обслуживать модель с минимальными затратами;
- версионировать, отслеживать и распространять наборы данных и модели;
- осуществлять мониторинг инфраструктуры и поведения моделей;
- развертывать модель в масштабируемой инфраструктуре;
- автоматизировать процессы обучения и развертывания.

Решение таких вопросов обычно возлагается на ML- или MLOps-инженера, в то время как за обучение модели чаще всего отвечает команда исследователей или специалистов по анализу данных.

На рис. 1.1 показаны основные компоненты, которые, по мнению команды Google Cloud, необходимы для создания полноценной ML- и MLOps-системы.

Помимо ML-кода (программного кода системы машинного обучения), данная схема должна включать в себя ряд других взаимосвязанных этапов, таких как конфигурации, автоматизация, сбор и проверка данных, тестирование и отладка, управление ресурсами, процессами и метаданными, анализ и мониторинг модели, а также инфраструктура ее обслуживания. Таким образом, при развертывании ML-модели в продакшене необходимо учитывать множество факторов.

---

[1] Гиперпараметры — это внешние настройки алгоритма машинного обучения, которые задаются до начала обучения и не изменяются самой моделью в процессе. Они управляют тем, как именно модель обучается, определяя ее структуру, скорость и качество работы. — *Примеч. науч. ред.*

[2] Признаки — это единицы информации, используемые для поиска подходящего контекста и генерации на его основе ответа от LLM. Например, для текста это может быть его абзац. — *Примеч. науч. ред.*



Рис. 1.1. Типовые компоненты ML-системы

В связи с этим возникает важный вопрос: как объединить все эти компоненты в согласованную систему? Чтобы на него ответить, нужно создать четкий шаблон для проектирования архитектуры ML-систем.

Аналогичный подход уже давно применяется в сфере разработки программного обеспечения. Большинство приложений можно разложить на три уровня: базу данных, бизнес-логику и пользовательский интерфейс. Каждый из этих уровней может быть сколь угодно сложным, но в обобщенном виде архитектура ПО сводится к этим трем компонентам.

Существует ли нечто подобное для ML-приложений? Прежде чем ответить на данный вопрос, рассмотрим существующие решения и выясним, почему они не подходят для построения масштабируемых ML-систем.

Проблема традиционных решений

На рис. 1.2 представлена типовая архитектура, используемая во многих ML-приложениях. Она монолитна, поскольку объединяет этапы создания признаков, обучения модели и инференса в один компонент. Такой подход позволяет быстро устранить одну из главных проблем машинного обучения — расхождение

между обучением и инференсом. Эта проблема возникает в тех случаях, когда признаки, подаваемые модели на стадии инференса, вычисляются иначе, чем на этапе ее обучения.



Рис. 1.2. Архитектура монолитного пакетного пайплайна

В рамках данной архитектуры признаки вычисляются одним и тем же кодом, что автоматически устраняет упомянутое расхождение. Такая схема вполне подходит для работы с небольшими объемами данных. Пайплайн запускается по расписанию в пакетном режиме, а предсказания используются сторонним приложением, например дашбордом.

Однако построение монолитной пакетной системы порождает ряд других проблем:

- признаки нельзя повторно использовать (ни внутри системы, ни в других проектах)[1];

[1] Например, для одной задачи определения вероятности покупки нужно через запрос рассчитать количество заходов на сайт покупателя. Через какое-то время появляется задача предсказать вероятность мошеннической покупки, на которую это значение тоже влияет, и в таком случае его придется заново рассчитывать из исходных данных. — *Примеч. науч. ред.*

- при росте объема данных приходится полностью перерабатывать код, чтобы обеспечить поддержку фреймворков PySpark или Ray;
- модуль предсказаний сложно переписать на более эффективном языке, например C++, Java или Rust;
- трудно разделить работу между командами, отвечающими за признаки, обучение и предсказания;
- невозможно переключиться на потоковую обработку для реализации обучения в реальном времени.

На рис. 1.3 показан аналогичный сценарий, но уже для системы с инференсом в режиме реального времени. Здесь к перечисленным проблемам добавляется еще одна: для генерации предсказаний все состояние приходится передавать в клиентском запросе — только так можно вычислить признаки и передать их модели.



Рис. 1.3. Архитектура системы с инференсом в режиме реального времени без сохранения состояния

Рассмотрим пример с рекомендацией фильмов для пользователей. Вместо того чтобы просто передать идентификатор пользователя, нам приходится отправлять полное его состояние, включая имя, возраст, пол, историю просмотров и т. п. Такой подход чреват ошибками, так как клиент должен знать, как получить данные сведения, и при этом он оказывается жестко связан с модельным сервисом.

Аналогичная ситуация возникает при реализации LLM с поддержкой RAG. Документы, добавляемые к запросу в качестве контекста, представляют собой внешнее состояние. Если бы мы не сохраняли эти записи в векторной базе данных, их пришлось бы передавать вместе с пользовательским запросом. Это потребовало бы от клиента умения запрашивать и извлекать документы, что весьма непрактично. Когда клиентское приложение вынуждено само вычислять или извлекать признаки, это считается антипаттерном. Если вы пока не знакомы с принципами работы механизма RAG, не беспокойтесь, мы подробно разберем их в главах 8 и 9.

Таким образом, наша главная задача состоит в том, чтобы понять, как получить доступ к признакам, необходимым для генерации предсказаний, не передавая их с клиентским запросом. Вернемся к примеру с рекомендациями фильмов: как предсказать предпочтения пользователя, опираясь исключительно на его идентификатор? Запомните эти вопросы, мы еще к ним вернемся.

На другом конце спектра находится архитектура от Google Cloud — готовое к использованию решение, представленное на рис. 1.4. Однако, несмотря на свою работоспособность, оно является довольно сложным и неинтуитивным. Без опыта в развертывании и сопровождении ML-моделей в продакшене разобраться в этой архитектуре будет непросто. Кроме того, не вполне ясно, как запустить такую систему в упрощенном виде и постепенно масштабировать ее.

Диаграмма на рис. 1.4 взята из документа, опубликованного компанией Google и распространяемого на условиях лицензии Creative Commons Attribution 4.0.

И здесь на сцену выходят архитектуры ML-систем на основе FTI-пайплайнов, которые позволяют решить указанные выше фундаментальные проблемы.

Решение: ML-пайплайны для построения ML-систем

Предлагаемое решение основано на построении четкой и простой ментальной модели, которой может следовать любая команда или инженер при создании признаков, обучении модели и генерации предсказаний. Эта структура, охватывающая три ключевых этапа работы любой ML-системы, известна как FTI-пайплайн. В чем же его отличие от ранее представленных подходов?

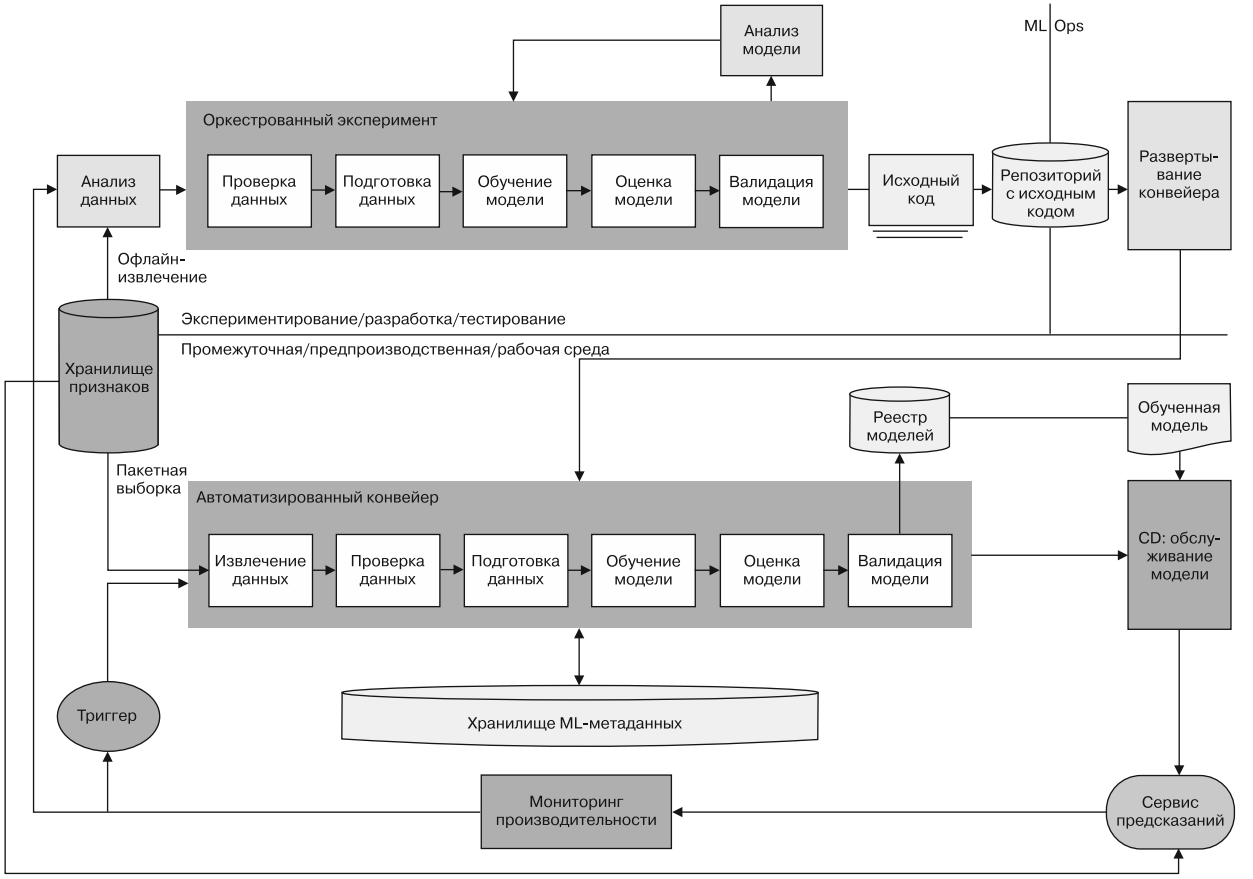


Рис. 1.4. Автоматизация ML-пайплайна для непрерывного обучения (источник: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning)

Данный паттерн предполагает, что любую ML-систему можно свести к трем базовым пайплайнам: признаков, обучения и инференса (по аналогии с такими уровнями традиционных систем, как база данных, бизнес-логика и пользовательский интерфейс). Этот подход позволяет четко определить границы и интерфейсы каждого пайплайна, а также понять, как они взаимодействуют друг с другом. В итоге вместо двадцати взаимосвязанных компонентов, представленных на рис. 1.4, у нас остается всего три, что существенно упрощает работу.

На рис. 1.5 изображена архитектура на основе FTI-пайплайнов. В следующих разделах мы подробно рассмотрим задачи и интерфейсы каждого из них.

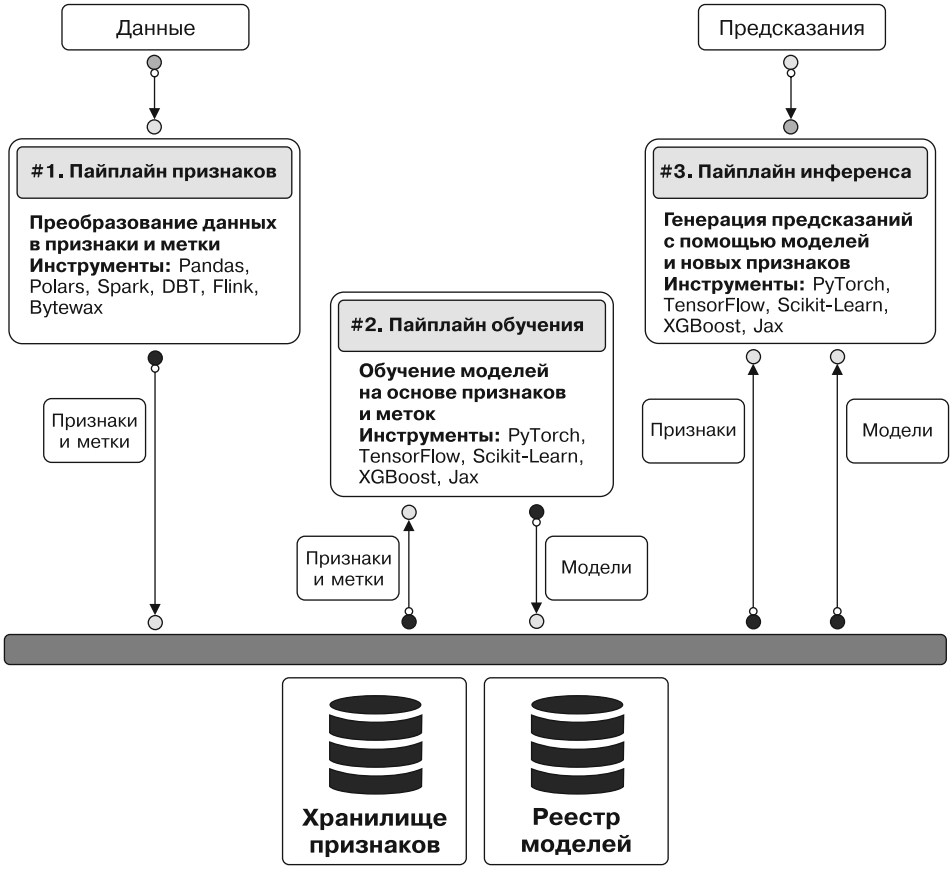


Рис. 1.5. Архитектура на основе FTI-пайплайнов

Прежде чем перейти к деталям, важно отметить, что каждый из этих пайплайнов представляет собой отдельный компонент, который может выполняться в отдельном процессе или даже на другом аппаратном обеспечении. Поэтому

любой такой пайплайн может быть реализован с использованием разных технологий, поддерживаться отдельной командой и масштабироваться независимо от остальных. Ключевая идея состоит в том, что архитектура остается максимально гибкой, что позволяет адаптировать ее под нужды вашей команды. Таким образом, данный паттерн служит мысленной моделью для структурирования ML-системы.

Пайплайн признаков

Пайплайн признаков принимает на вход сырые данные, обрабатывает их и формирует признаки и метки, необходимые модели для обучения или инференса. Однако эти признаки и метки не передаются непосредственно в модель, а отправляются в хранилище признаков, задача которого заключается в их хранении, версионировании, отслеживании и предоставлении. Благодаря этому у нас всегда есть сохраненное состояние признаков, и мы можем легко передавать их в пайплайны обучения и инференса.

Версионирование данных обеспечивает совпадение признаков на всех этапах работы модели и тем самым устраняет проблему расхождения между обучением и инференсом.

Пайплайн обучения

Пайплайн обучения получает на вход признаки и метки из хранилища признаков и дообучает одну или несколько моделей, которые сохраняются в реестре моделей (model registry). Этот реестр выполняет ту же функцию, что и хранилище признаков, но применительно к моделям. Таким образом, реестр моделей отвечает за их хранение, версионирование, отслеживание и передачу в пайплайн инференса.

Большинство современных реестров моделей поддерживают хранилище метаданных, где фиксируются основные сведения об обучении модели, в первую очередь использованные в ходе этого процесса признаки, метки и их версии, благодаря чему мы всегда знаем, на каких данных была обучена каждая конкретная модель.

Пайплайн инференса

Пайплайн инференса принимает на вход признаки и метки из хранилища признаков и обученную модель из реестра моделей. Этих двух компонентов достаточно для того, чтобы генерировать ответы — как в пакетном режиме, так и в режиме реального времени.

Поскольку данный паттерн достаточно универсален, вы сами определяете, как обрабатывать результаты предсказаний. В пакетной системе они, как правило, записываются в базу данных. В системе, работающей в режиме реального

времени, они возвращаются клиенту, инициировавшему запрос. Кроме того, все признаки, метки и модели версионируются, что упрощает обновление и откат развернутых решений. Например, мы знаем, что модель v1 использует признаки F1, F2 и F3, а модель v2 — F2, F3 и F4, благодаря чему мы можем быстро переключаться между разными конфигурациями модели и признаков.

Преимущества FTI-архитектуры

В завершение еще раз подчеркнем ключевые особенности интерфейсов FTI-пайплайнов:

- пайплайн признаков принимает на вход данные и сохраняет признаки и метки в хранилище признаков;
- пайплайн обучения извлекает признаки и метки из хранилища признаков и сохраняет обученную модель в реестре моделей;
- пайплайн инференса использует признаки из хранилища и модель из реестра для генерации предсказаний.

Какой бы сложной ни была ваша ML-система, эти интерфейсы всегда останутся неизменными.

Теперь, когда структура FTI-паттерна стала более ясной, сформулируем его основные преимущества:

- простота и наглядность — всего три ключевых компонента, которые легко понять и реализовать;
- гибкость в выборе технологий — каждый компонент может быть реализован на своей технологической платформе, что позволяет адаптировать систему под конкретные задачи, например, для обработки больших объемов данных или их потоковой обработки;
- разделение ответственности — благодаря четким границам интерфейсов каждый компонент может разрабатываться отдельной командой, что упрощает масштабирование и управление проектом;
- независимое развертывание — каждый компонент можно развертывать, масштабировать и мониторить независимо от остальных.

Важно понимать, что реальная система иногда включает в себя больше трех компонентов. Например, в состав пайплайна признаков могут входить отдельные сервисы для вычисления признаков и валидации данных, а в пайплайн обучения — модули обучения и оценки модели.

FTI-пайплайны следует воспринимать как логические уровни. Каждый из них может быть достаточно сложным и включать множество сервисов. Однако принципиально важно придерживаться единого интерфейса взаимодействия между

пайплайнами через хранилище признаков и реестр моделей. Это позволяет каждому элементу FTI-архитектуры развиваться независимо, без необходимости знать внутреннее устройство других компонентов и без риска нарушить работу системы при внесении изменений.



Если вы хотите глубже разобраться в FTI-архитектуре, рекомендуем ознакомиться со статьей From MLOps to ML Systems with Feature/Training/Inference Pipelines Джима Даулинга, главного исполнительного директора и сооснователя компании Hopsworks. Именно она вдохновила авторов книги на написание данного раздела (https://www.hopsworks.ai/post/mlops-to-ml-systems-with-fti-pipelines).

Теперь, когда мы разобрались с архитектурой FTI-пайплайнов, посмотрим, как ее можно применить для создания LLM-двойника.

Проектирование архитектуры системы LLM-двойника

В этом разделе мы перечислим конкретные технические требования к LLM-двойнику и разберем, как их можно реализовать с помощью FTI-архитектуры. Однако, прежде чем переходить к описанию пайплайнов, важно подчеркнуть, что на данном этапе мы не будем фокусироваться на инструментах или технологическом стеке. Сейчас наша цель — определить архитектуру системы на концептуальном уровне, не привязываясь к языкам программирования, фреймворкам, платформам и инфраструктуре. Здесь мы сосредоточимся на задачах каждого компонента, их интерфейсах и взаимосвязях, а в последующих главах подробно рассмотрим вопросы реализации и используемые технологии.

Перечень технических требований к архитектуре LLM-двойника

До сих пор мы говорили о том, какие функции должен поддерживать LLM-двойник с точки зрения пользователя. Теперь сформулируем требования к системе машинного обучения с сугубо технической позиции.

В рамках работы с данными мы должны иметь возможность делать следующее:

- автоматически и по расписанию собирать данные из соцсетей;
- стандартизировать собранные данные и сохранять их в хранилище;
- очищать исходные данные;

- создавать наборы данных с инструкциями (instruction dataset) для тонкой настройки LLM;
- разбивать очищенные данные на фрагменты и векторизовать их, а также сохранять векторные представления в векторную базу данных для последующей генерации, дополненной поиском.

В рамках обучения модели мы должны иметь возможность делать следующее:

- проводить тонкую настройку LLM разных размеров (7, 14, 30 или 70 млрд параметров);
- тонко настраивать модель на наборах данных с инструкциями разного объема;
- переключаться между различными типами LLM (например, Mistral, Llama и GPT);
- отслеживать и сравнивать результаты экспериментов;
- тестировать потенциальных кандидатов для развертывания в продакшене;
- автоматически запускать процесс обучения при появлении новых наборов данных с инструкциями.

Код, отвечающий за инференс, должен иметь возможность делать следующее:

- предоставлять REST API для взаимодействия с LLM-двойником;
- получать доступ к векторной базе данных в реальном времени для целей RAG;
- осуществлять инференс с использованием LLM разного размера;
- автоматически масштабироваться в зависимости от пользовательских запросов;
- автоматически развертывать модели, успешно прошедшие этап оценки.

Система должна поддерживать следующие возможности LLMOps:

- версионирование, отслеживание происхождения и повторное использование наборов данных с инструкциями;
- версионирование, отслеживание происхождения и повторное использование моделей;
- отслеживание экспериментов;
- непрерывное обучение, интеграция и доставка (continuous training/integration/delivery, CT/CI/CD);
- мониторинг промптов и состояния системы.



Если какие-то из требований пока кажутся неясными, не беспокойтесь. Мы рассмотрим их подробнее в соответствующих главах.

Представленный список уже довольно обширен. Мы могли бы дополнительно детализировать его, но сейчас важно сосредоточиться на ключевой функциональности. Нюансы мы обсудим, когда перейдем к реализации каждого из компонентов. А сейчас нам нужно ответить на фундаментальный вопрос о том, как применить FTI-архитектуру для выполнения перечисленных выше требований.

Проектирование архитектуры LLM-двойника с помощью FTI-пайплайнов

Мы разобьем нашу систему на четыре ключевых компонента. Возможно, вы спросите: «Почему на четыре, если FTI-пайплайнов всего три?» Дело в том, что, помимо пайплайнов признаков, обучения и инференса, нам также необходимо реализовать пайплайн данных. Согласно лучшим практикам:

- пайплайн данных находится в зоне ответственности команды дата-инженеров;
- за FTI-пайплайны отвечает команда ML-инженеров.

Поскольку наша цель заключается в том, чтобы создать MVP усилиями небольшой команды, нам предстоит реализовать все приложение целиком, включая создание всех четырех пайплайнов. Такая ситуация часто возникает в стартапах, которые не могут позволить себе выделить специализированные команды для решения отдельных задач. Поэтому инженерам часто приходится совмещать несколько ролей в процессе создания продукта. Тем не менее знание устройства ML-системы ценно для успешного взаимодействия с другими участниками команды.

На рис. 1.6 представлена архитектура системы LLM-двойника. Чтобы лучше понять ее устройство, рассмотрим каждый из четырех компонентов и обсудим их назначение.

Пайплайн сбора данных

Пайплайн сбора данных отвечает за автоматизированный сбор персональных данных с таких платформ, как Medium, Substack, LinkedIn и GitHub. Для реализации этого пайплайна мы будем использовать паттерн *ETL* (extract, transform, load — «извлечение, преобразование, загрузка»), обеспечивающий извлечение данных с указанных платформ, их стандартизацию и загрузку в хранилище.



Важно подчеркнуть, что пайплайн сбора данных предназначен исключительно для обработки вашей собственной информации, опубликованной в социальных сетях. Он не должен получать доступ к данным других пользователей. Авторы книги согласились предоставить свои данные для реализации описанного учебного примера. В остальных случаях несанкционированное использование чужих данных является неэтичным.

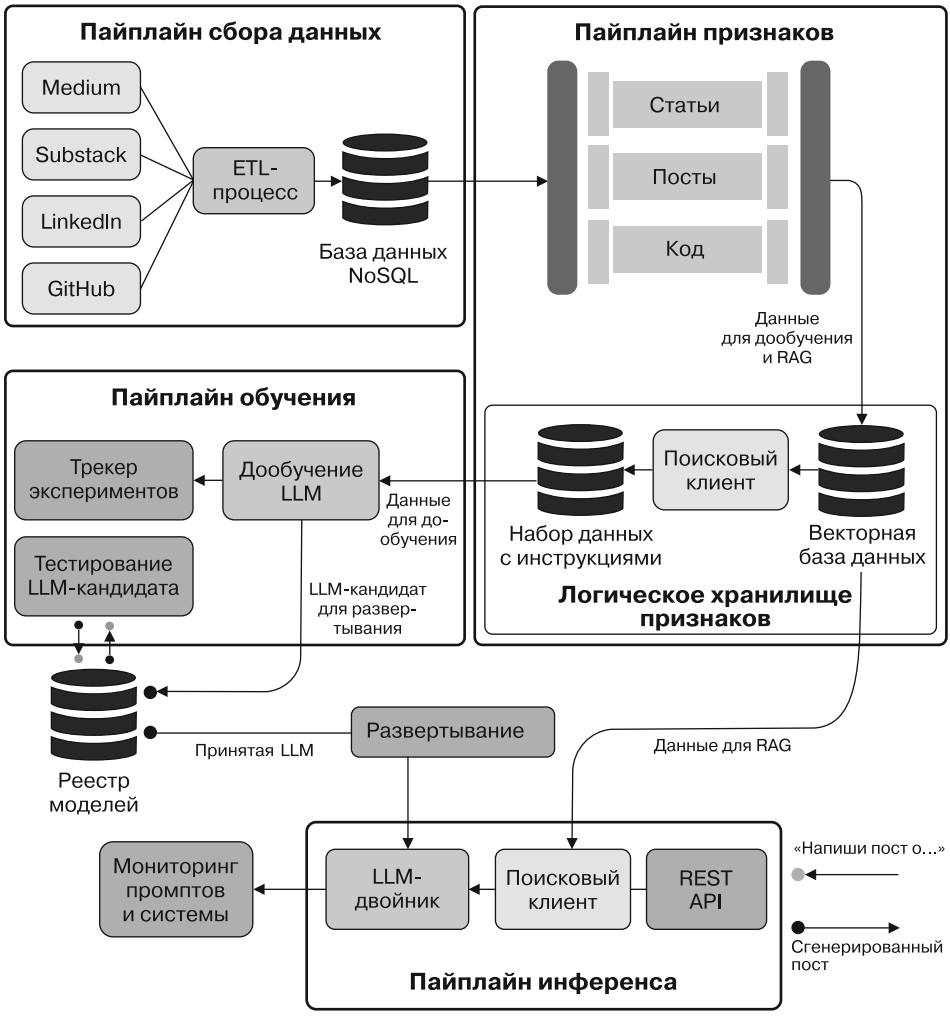


Рис. 1.6. Высокоуровневая архитектура LLM-двойника

Результат работы этого компонента — база данных NoSQL, которая выполняет роль хранилища. Поскольку мы работаем с текстовыми данными, неструктурированными по своей природе, NoSQL — отличный выбор.

Хотя базы NoSQL, такие как MongoDB, формально не считаются хранилищами данных, в нашем контексте они будут выполнять именно эту функцию, поскольку в них сохраняются стандартизированные исходные данные, собранные различными ETL-процессами и подготовленные к использованию в ML-системе.

Собранные цифровые данные делятся на три категории:

- статьи (Medium, Substack);
- посты (LinkedIn);
- код (GitHub).

Важно абстрагироваться от конкретной платформы, с которой были получены данные. Например, при передаче статьи в LLM не имеет значения, была она взята с Medium или Substack. Ссылку на источник можно сохранить в виде метаданных. Однако с точки зрения обработки, тонкой настройки и RAG крайне важно знать тип данных, поскольку каждая категория требует отдельного подхода. Так, стратегия разбиения на фрагменты для поста, статьи и кода будет различаться.

Кроме того, если группировать данные по категориям, а не по источникам, мы сможем легко подключать новые платформы, например, чтобы отнести данные с X к категории постов, а данные с GitLab — к категории кода. Благодаря модульной архитектуре системы для этого достаточно включить дополнительный ETL-процесс в пайплайн сбора данных, не внося изменений в остальной код.

Пайплайн признаков

Пайплайн признаков отвечает за извлечение сырых статей, постов и фрагментов кода из хранилища данных, их обработку и последующую загрузку в хранилище признаков.

Здесь уже проявляются характерные черты FTI-архитектуры, однако у пайплайна признаков LLM-двойника есть несколько особенностей:

- он по-разному обрабатывает три типа данных (статьи, посты и код);
- он предусматривает три основных этапа, необходимых для тонкой настройки модели и RAG: очистку, разбиение на фрагменты и векторизацию (создание эмбеддингов);
- он создает два снэпшота (snapshot) состояния цифровых данных: один после очистки (для тонкой настройки), другой после векторизации (для RAG);
- вместо специализированного хранилища признаков он использует логическое.

Рассмотрим логическое хранилище признаков подробнее. Как и в любой системе на базе RAG, центральным элементом инфраструктуры является векторная БД. Вместо того чтобы интегрировать еще одну базу данных — точнее, специализированное хранилище признаков, — мы использовали векторную БД, дополнив ее логикой, обеспечивающей все свойства, необходимые нашей системе.

Векторная БД не поддерживает концепцию обучающего набора данных, но может использоваться как NoSQL-хранилище, то есть допускает обращение к данным по их идентификаторам и названию коллекции. Это позволяет нам запрашивать новые точки данных[1] без применения механизмов векторного поиска. В дальнейшем мы будем оборачивать извлеченные данные в версионируемый, отслеживаемый и повторно используемый артефакт. Подробнее об артефактах мы поговорим во второй главе, а пока достаточно знать то, что это понятие из области MLOps, обозначающее упаковку данных с добавлением перечисленных выше свойств.

Как же другие компоненты системы будут обращаться к логическому хранилищу признаков? Пайплайн обучения воспользуется наборами данных с инструкциями в качестве артефактов, а пайплайн инференса запросит дополнительный контекст из векторной БД с помощью методов векторного поиска.

Такой подход полностью покрывает требования в нашем случае, и вот почему:

- артефакты отлично подходят для таких офлайн-сценариев, как обучение;
- векторная БД позволяет осуществлять онлайн-доступ, необходимый при выполнении инференса.

В следующих главах мы подробно рассмотрим процессы очистки, разбиения и векторизации статей, постов и фрагментов кода.

В заключение отметим, что в нашем случае на вход подаются сырые данные, которые обрабатываются и сохраняются в хранилище признаков для дальнейшего предоставления пайплайнам обучения и инференса. Обратите внимание, что устранение избыточной сложности и фокус на интерфейсе идеально воплощают принципы паттерна FTI. Здорово, правда?

Пайплайн обучения

Пайплайн обучения получает наборы данных с инструкциями из хранилища признаков, выполняет тонкую настройку LLM, а затем сохраняет полученные веса в реестре моделей. Проще говоря, когда в логическом хранилище признаков появляется новый набор данных с инструкциями, мы запускаем пайплайн обучения, извлекаем соответствующий артефакт и на его основе выполняем тонкую настройку LLM.

На начальном этапе этим процессом управляет команда специалистов по данным. Они проводят серию экспериментов, подбирая наиболее подходящую модель и гиперпараметры — как вручную, так и с помощью автоматизированных методов настройки. Для сравнения и выбора оптимальной конфигурации

---

[1] Точка данных — один пример из набора для обучения. — *Примеч. науч. ред.*

гиперпараметров используется система отслеживания (трекер) экспериментов, фиксирующая все значимые метрики. В итоге команда выбирает лучшую конфигурацию гиперпараметров и тонко настроенную модель и выдвигает ее в качестве LLM-кандидата для развертывания. Предложенная модель сохраняется в реестре моделей.

После завершения экспериментальной стадии подобранные гиперпараметры сохраняются для повторного использования, что устраняет необходимость в ручной настройке и позволяет полностью автоматизировать процесс обучения, то есть реализовать механизм непрерывного обучения (CT).

Затем запускается тестовый пайплайн, выполняющий более строгую проверку модели, чем на этапе тонкой настройки. Прежде чем новая модель будет развернута в продакшене, ее необходимо протестировать с использованием более жесткого набора критериев, чтобы убедиться в том, что она действительно превосходит текущую рабочую модель. Если модель проходит это тестирование, она считается принятой и развертывается в пайплайне инференса. Даже в полностью автоматизированной ML-системе рекомендуется предусмотреть этап ручной проверки перед внедрением новой модели в продакшен, который будет играть роль «красной кнопки», позволяющей отменить действие, чреватое серьезными последствиями. На этом этапе эксперт просматривает отчет, сгенерированный компонентом тестирования. Если все в порядке, модель утверждается и работа системы снова переводится в автоматический режим.

Особенности данного компонента будут определяться ответами на следующие вопросы.

- Как реализовать пайплайн, независимый от конкретной LLM?
- Какие методы тонкой настройки использовать?
- Как масштабировать алгоритм тонкой настройки для LLM и наборов данных разного размера?
- Как выбрать LLM-кандидата для развертывания из множества экспериментальных моделей?
- Как протестировать LLM на предмет ее пригодности к развертыванию в продакшене?

Прочитав книгу до конца, вы будете знать, как на них ответить.

Наконец, стоит отдельно отметить, как реализуется процесс *непрерывного обучения* (CT). Благодаря модульной архитектуре мы можем легко использовать ML-оркестратор для планирования и запуска различных компонентов системы. Например, пайплайн сбора данных можно запускать раз в неделю, пайплайн признаков — при появлении новых данных в хранилище, а пайплайн обучения — при поступлении очередных наборов данных с инструкциями.

Пайплайн инференса

Пайплайн инференса — завершающий элемент архитектуры. Он связан с реестром моделей и логическим хранилищем признаков: из реестра загружается тонко настроенная LLM, а из логического хранилища признаков осуществляется доступ к векторной базе данных для выполнения RAG. Пайплайн получает клиентские запросы через интерфейс REST API, а затем формирует ответ с использованием тонко настроенной LLM и содержимого векторной БД.

Все клиентские запросы, расширенные промпты и сгенерированные ответы передаются в систему мониторинга промптов, которая применяется для анализа, отладки и более глубокого понимания поведения модели. В зависимости от заданных условий система мониторинга может инициировать сигналы, требующие ручного вмешательства или автоматической реакции.

На уровне интерфейсов этот компонент полностью соответствует паттерну FTI, однако при более детальном рассмотрении проявляются уникальные черты, характерные для систем на базе LLM и RAG, в частности:

- использование поискового клиента для осуществления векторного поиска в рамках RAG;
- применение шаблонов промптов для сопоставления пользовательских запросов и внешней информации с входными данными, подаваемыми в LLM;
- наличие специализированных средств мониторинга промптов.

Некоторые соображения по поводу архитектуры LLM-двойника и паттерна FTI

Паттерн FTI не следует воспринимать как догму. Это инструмент, помогающий упростить процесс проектирования ML-систем. Например, вместо того, чтобы использовать специализированное хранилище признаков только потому, что так принято, в нашем случае удобнее и дешевле задействовать логическое хранилище признаков, основанное на векторной базе данных и артефактах. Главное — обеспечить необходимые свойства, такие как версионируемый и повторно используемый набор обучающих данных.

Давайте кратко перечислим вычислительные требования для каждого компонента. Пайплайны сбора данных и признаков в основном зависят от центрального процессора (CPU) и не требуют мощных машин. Пайплайн обучения, напротив, нуждается в машинах с высокопроизводительными графическими процессорами (GPU), способных загружать и тонко настраивать LLM. Пайплайн инференса занимает промежуточное положение: ему тоже нужна производительная машина, но нагрузка здесь меньше, чем при обучении. Однако тестировать его необходимо особенно тщательно, так как он напрямую взаи-

модействует с пользователем. То есть для обеспечения удовлетворительного пользовательского опыта задержка ответа должна укладываться в допустимые пределы. В этом отношении использование паттерна FTI не создает затруднений, так как для каждого компонента можно выбрать соответствующую конфигурацию оборудования.

Масштабирование этих пайплайнов тоже происходит немного по-разному. Пайплайны сбора данных и признаков масштабируются горизонтально в зависимости от нагрузки на CPU и оперативную память. Пайплайн обучения масштабируется вертикально путем увеличения числа GPU, а пайплайн инференса — горизонтально в зависимости от количества клиентских запросов.

Таким образом, представленная архитектура LLM-двойника полностью соответствует техническим требованиям, сформулированным в начале раздела. Система обрабатывает данные нужным образом, а пайплайн обучения обладает модульной структурой, что позволяет быстро адаптировать его под разные LLM, наборы данных и техники тонкой настройки. Пайплайн инференса поддерживает RAG и доступен через REST API. С точки зрения LLMOps LLM-система обеспечивает версионирование, отслеживание происхождения и повторное использование наборов данных и моделей. Она включает механизмы мониторинга и изначально спроектирована с учетом непрерывного обучения, интеграции и доставки (CT/CI/CD).

На этом высокоуровневый обзор архитектуры LLM-двойника завершен.

Резюме

Первая глава имеет ключевое значение для понимания цели книги. Поскольку эта книга ориентирована на построение полноценной ML-системы, нам было принципиально важно первым делом описать концепцию LLM-двойника. Затем мы поговорили о том, что такое минимально жизнеспособный продукт (MVP) и как спланировать его реализацию для LLM-двойника с учетом имеющихся ресурсов. Далее мы перешли от концепции к практическому техническому решению, сформулировав конкретные требования к создаваемой системе. В этом контексте были представлены паттерн FTI и его роль в проектировании модульных и масштабируемых систем. Затем мы применили архитектуру FTI для проектирования системы LLM-двойника, обеспечив ее соответствие всем заявленным техническим требованиям.

При создании систем важно иметь целостное видение. Понимание того, как отдельный компонент встраивается в общую систему, существенно повышает качество и осмысленность разработки. Мы начали с абстрактного представления архитектуры LLM-двойника, сосредоточив внимание на назначении и интерфейсах каждого из компонентов, а также на их взаимодействии друг с другом.

В следующих главах мы подробно рассмотрим процесс реализации и развертывания этих элементов. Вы узнаете, как использовать вычислительную платформу, оркестратор, реестр моделей, артефакты и другие инструменты и концепции, которые обеспечивают соответствие лучшим практикам MLOps.

Список используемых источников

- *Dowling J.* From MLOps to ML Systems with Feature/Training/Inference Pipelines. Hopsworks. https://www.hopsworks.ai/post/mlops-to-ml-systems-with-fti-pipelines.
- *Dowling J.* Modularity and Composability for AI Systems with AI Pipelines and Shared Storage. Hopsworks. https://www.hopsworks.ai/post/modularity-and-composability-for-ai-systems-with-ai-pipelines-and-shared-storage.
- *Joseph M.* The Taxonomy for Data Transformations in AI Systems. Hopsworks. https://www.hopsworks.ai/post/a-taxonomy-for-data-transformations-in-ai-systems.
- Google Cloud. MLOps: Continuous Delivery and Automation Pipelines in Machine Learning. https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning.
- Qwak. CI/CD for Machine Learning in 2024: Best Practices to Build, Test and Deploy. Medium. https://medium.com/infer-qwak/ci-cd-for-machine-learning-in-2024-best-practices-to-build-test-and-deploy-c4ad869824d2.
- Qwak. 5 Best Open Source Tools to Build End-to-End MLOps Pipeline in 2024. Medium. https://medium.com/infer-qwak/building-an-end-to-end-mlops-pipeline-with-open-source-tools-d8bacbf4184f.
- *Salama K., Kazmierczak J., Schut D.* Practitioners Guide to MLOps: A Framework for Continuous Delivery and Automation of Machine Learning. Google Cloud. https://services.google.com/fh/files/misc/practitioners_guide_to_mlops_whitepaper.pdf.

2 Инструменты и установка

В этой главе представлены все основные инструменты, которые будут использоваться в процессе реализации и развертывания LLM-двойника. На данном этапе мы не станем подробно рассматривать концепции LLM, RAG, MLOps или LLMOps. Вместо этого мы кратко ознакомим вас с выбранным стеком технологий и предварительными требованиями, чтобы в последующих главах не тратить время на объяснение процесса настройки того или иного инструмента и причин его выбора. В главе 3 мы приступим к практической работе над LLM-двойником, начав с реализации ETL-пайплайна сбора данных, который будет автоматически извлекать информацию из Интернета.

В первой части главы мы рассмотрим инструменты из экосистемы Python, предназначенные для управления несколькими версиями данного языка, создания виртуального окружения и установки зафиксированных зависимостей, необходимых для запуска нашего проекта. Мы также покажем, как работать локально с репозиторием `LLM-Engineers-Handbook`, на тот случай, если вы захотите воспроизвести код из примеров: https://github.com/PacktPublishing/LLM-Engineers-Handbook.

Затем перейдем к обзору инструментов MLOps и LLMOps, начав с более универсальных (реестр моделей) и перейдя к специализированным и ориентированным на LLM (инструменты для оценки моделей, мониторинга промптов и т. д.). Мы также поговорим о том, как управлять проектом с несколькими ML-пайплайнами с помощью ZenML — оркестратора, позволяющего преодолеть разрыв между ML и MLOps. Кроме того, мы кратко рассмотрим базы данных, которые нам понадобятся для создания NoSQL- и векторного хранилища, и покажем, как запускать все компоненты локально с использованием Docker. Наконец, обсудим облачную платформу AWS и расскажем, как создать пользователя AWS, получить ключи доступа, а также установить и настроить инструмент AWS CLI для программного управления облачными ресурсами. Кроме того, мы объясним, почему выбрали именно сервис Amazon SageMaker для обучения и развертывания своей LLM с открытым исходным кодом.

Если вы уже знакомы с перечисленными инструментами, можете пропустить эту главу. Все инструкции по установке проекта и настройке необходимых компонентов также содержатся в README-файле репозитория. Таким образом, вы можете воспользоваться им как более кратким справочником, если планируете запускать код параллельно с изучением книги.

К концу главы вы будете иметь полное представление обо всех инструментах, которые нам предстоит использовать. Кроме того, вы узнаете, как установить репозиторий `LLM-Engineers-Handbook`, настроить остальные компоненты и применять их, если решите запускать код во время чтения книги.

Экосистема Python и установка проекта

Любой Python-проект опирается на три основных инструмента: интерпретатор Python, менеджер зависимостей и инструмент автоматизации задач. Интерпретатор выполняет код вашего проекта ожидаемым образом. Вся кодовая база книги протестирована на Python 3.11.8. Скачать нужную версию интерпретатора можно здесь: https://www.python.org/downloads/. Мы рекомендуем воспользоваться именно версией Python 3.11.8, чтобы запустить проект LLM Twin через инструмент `pyenv`.

Вместо установки нескольких глобальных версий Python лучше применить `pyenv` — инструмент управления версиями Python, который позволяет легко переключаться между ними при работе над разными проектами. Скачать его можно по следующей ссылке: https://github.com/pyenv/pyenv#installation.

После этого вы можете установить нужную версию Python (3.11.8), выполнив команду:

```
pyenv install 3.11.8
```

Чтобы убедиться в том, что все прошло успешно, выполните команду:

```
pyenv versions
```

Вы должны увидеть примерно следующее:

```
# * system
# 3.11.8
```

Если вы хотите, чтобы версия Python 3.11.8 применялась по умолчанию (при открытии любого нового терминала), выполните команду:

```
pyenv global 3.11.8
```

Однако мы собираемся использовать Python 3.11.8 локально, только внутри проекта. Для этого сначала необходимо клонировать репозиторий и перейти в соответствующий каталог:

```
git clone https://github.com/PacktPublishing/LLM-Engineers-Handbook.git
cd LLM-Engineers-Handbook
```

Поскольку в репозитории есть файл `.python-version`, инструмент `pyenv` автоматически определит указанную в нем версию и будет использовать ее локально при работе в данном каталоге. Чтобы убедиться в этом, выполните следующую команду, находясь в каталоге репозитория:

```
python --version
```

Вы должны увидеть следующее:

```
# Python 3.11.8
```

Для создания файла `.python-version` достаточно один раз выполнить команду `pyenv local 3.11.8`, после чего инструмент `pyenv` всегда будет использовать указанную версию Python при работе в этом каталоге.

Теперь, когда нужная версия Python установлена, перейдем к Poetry — менеджеру зависимостей и виртуального окружения.

Poetry: управление зависимостями и виртуальным окружением

Poetry является одним из самых популярных инструментов для управления зависимостями и виртуальным окружением в экосистеме Python. Но прежде, чем переходить к его обсуждению, давайте уточним, что менеджер зависимостей — это инструмент, позволяющий указывать, устанавливать, обновлять и управлять внешними библиотеками и пакетами (зависимостями), используемыми в проекте. Например, ниже приведен простой файл зависимостей Poetry, в котором указана версия Python 3.11, а также библиотеки `requests` и `numpy`.

```
[tool.poetry.dependencies]
python = "^3.11"
requests = "^2.25.1"
numpy = "^1.19.5"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Применение Poetry для фиксации зависимостей гарантирует установку именно тех версий библиотек, с которыми работает ваш проект. По умолчанию Poetry сохраняет зависимости в файле `pyproject.toml`, размещенном в корне репозитория. Вы можете увидеть его, клонировав репозиторий `LLM-Engineers-Handbook`.

Еще одно важное преимущество Poetry состоит в том, что этот инструмент автоматически создает новое виртуальное окружение Python и устанавливает в него нужную версию языка и все указанные зависимости. Такое окружение позволяет изолировать зависимости проекта от глобальных зависимостей Python и других проектов, исключая тем самым конфликты версий. Например, если проект A использует `numpy == 1.19.5`, а проект B — `numpy == 1.26.0` и оба находятся в одной глобальной среде, то установка второй версии библиотеки нарушит работу проекта, использующего первую. С помощью Poetry мы можем изолировать каждый проект в отдельной среде с собственными зависимостями и избежать подобных проблем.

Скачать Poetry можно здесь: https://python-poetry.org/docs/. В книге используется версия Poetry 1.8.3. После установки Poetry перейдите в клонированный репозиторий `LLM-Engineers-Handbook` и выполните команду для установки всех необходимых зависимостей:

```
poetry install --without aws
```

Данная команда использует зависимости, указанные в файлах `pyproject.toml` и `poetry.lock`, находящихся в репозитории. После этого вы можете работать в нужном окружении Poetry, выполнив в терминале команду `poetry shell` или добавляя префикс `poetry run` ко всем консольным командам, например: `poetry run <`*`команда`*`>`.

Стоит также отметить, что Poetry фиксирует точные версии дерева зависимостей в файле `poetry.lock`, основываясь на определениях, заданных в файле `pyproject.toml`. Например, в файле `pyproject.toml` может быть указан диапазон версий (`requests = "^2.25.1"`), а в `poetry.lock` — зафиксирована конкретная установленная версия (`requests = "2.25.1"`). Кроме того, в `poetry.lock` сохраняются точные версии вложенных зависимостей (то есть зависимостей зависимостей), даже если они явно не указаны в файле `pyproject.toml`. Фиксация всех зависимостей в файле `poetry.lock` гарантирует, что при установке проекта всегда будут использоваться одни и те же версии зависимостей (библиотек). Такая согласованность обеспечивает предсказуемое поведение и снижает вероятность появления проблем из серии «на моем компьютере все работает».

Для создания виртуальных окружений предусмотрены и другие инструменты, такие как Venv и Conda, однако они не позволяют управлять зависимостями. В случае их выбора вам придется использовать стандартные файлы `requirements.txt`, менее выразительные и надежные по сравнению с `poetry.lock`.

Еще одной альтернативой является инструмент Pipenv, который по функциональности близок к Poetry, но работает медленнее. Кроме того, существует инструмент uv — более быстрый аналог Poetry, написанный на Rust. Он обладает большим потенциалом и со временем может заменить Poetry, поэтому стоит обратить на него внимание: https://github.com/astral-sh/uv.

В следующем разделе мы рассмотрим инструмент для выполнения консольных команд, который будет управлять задачами в рамках нашего проекта.

Poe the Poet: инструмент для выполнения команд

Poe the Poet — это плагин для Poetry, предназначенный для управления консольными командами, необходимыми для взаимодействия с проектом. Он позволяет определять и выполнять задачи внутри Python-проекта, упрощая автоматизацию и запуск сценариев. Другими популярными вариантами являются Makefile, Invoke или shell-сценарии (сценарии командной оболочки), однако Poe the Poet избавляет от необходимости писать отдельные shell-сценарии или файлы Makefile для управления задачами проекта, позволяя применять для этого файл конфигурации, который Poetry уже использует для зависимостей.

При работе с Poe the Poet, вместо того чтобы документировать все команды в файле README или другом документе, допускается добавить их непосредственно в `pyproject.toml` и затем вызывать в командной строке с помощью заданных псевдонимов. Например, с помощью Poe the Poet можно определить следующие задачи в файле `pyproject.toml`:

```
[tool.poe.tasks]
test = "pytest"
format = "black ."
start = "python main.py"
```

После этого вы сможете запускать их с помощью команды `poe`:

```
poetry poe test
poetry poe format
poetry poe start
```

Установить Poe the Poet в качестве плагина для Poetry можно следующим образом:

```
poetry self add 'poethepoet[poetry_plugin]'
```

Использование подобного инструмента в качестве встроенной документации для управления консольными командами значительно упрощает структуру проекта и облегчает совместную работу над ним.

Если у вас уже установлены `pyenv` и Poetry, то вот набор команд, который позволит вам клонировать репозиторий, установить зависимости и подключить Poe the Poet как плагин Poetry:

```
git clone https://github.com/PacktPublishing/LLM-Engineers-Handbook.gitcd
LLM-Engineers-Handbook
poetry install --without aws
poetry self add 'poethepoet[poetry_plugin]'
```

Чтобы проект стал полностью рабочим, осталось выполнить несколько дополнительных шагов, в частности заполнить файл `.env` своими учетными данными и получить токены от OpenAI и Hugging Face. Если вы планируете самостоятельно реализовать описанный в книге проект, соответствующие инструкции вы найдете в файле README репозитория: https://github.com/PacktPublishing/LLM-Engineers-Handbook.

Теперь, когда установка Python-проекта завершена, обсудим инструменты MLOps, которые будут использоваться в книге. Если вы уже знакомы с ними, можете сразу перейти к разделу «Базы данных для хранения неструктурированных и векторных данных».

Инструменты MLOps и LLMOps

В этом разделе мы кратко обсудим все инструменты MLOps и LLMOps, которые встречаются в книге, а также их роль в построении ML-систем с соблюдением лучших практик MLOps. На данном этапе мы не будем подробно останавливаться на всех компонентах, используемых в архитектуре LLM-двойника, таких как реестры моделей и оркестраторы. Наша задача — дать общее представление об этих инструментах и способах их применения. В последующих главах мы рассмотрим практические примеры их использования в ходе разработки проекта. В главе 11 мы подробно обсудим основные концепции и принципы MLOps и LLMOps и свяжем воедино все, о чем говорилось ранее. Поскольку эти области имеют прикладной характер, теорию будет легче усвоить после погружения в практику.

Стоит также отметить, что данный раздел не посвящен настройке инструментов. Мы лишь объясним, для чего нужен каждый из них, и выделим основные функции, используемые в книге.

Тем не менее с помощью Docker вы сможете запустить всю инфраструктуру локально. Если вы хотите пройти все описанные в книге этапы самостоятельно, то для локального запуска приложения вам будет достаточно выполнить три простых шага.

1. Установите Docker версии 27.1.1 или выше.
2. Заполните файл `.env` всеми необходимыми учетными данными (см. инструкцию в README-файле репозитория).
3. Выполните команду `poetry poe local-infrastructure-up`, чтобы локально запустить ZenML (`http://127.0.0.1:8237/`), а также базы данных MongoDB и Qdrant.

Подробную инструкцию по локальному запуску можно найти в README-файле репозитория, доступного по ссылке https://github.com/PacktPublishing/LLM-Engineers-Handbook. В книге также будет показано, как развернуть каждый компонент в облаке.

Hugging Face: реестр моделей

Реестр моделей — это централизованное хранилище, предназначенное для управления ML-моделями на всех этапах их жизненного цикла. В нем сохраняются сами модели, а также связанные с ними метаданные, история версий и показатели производительности. Такой реестр служит единым источником истины, который позволяет отслеживать, документировать и распространять версии моделей, а также упрощает командную работу над ними. В MLOps-архитектурах реестр моделей является ключевым элементом процесса развертывания, поскольку легко интегрируется с пайплайнами *непрерывной интеграции и развертывания* (CI/CD).

В качестве реестра моделей для нашего проекта мы использовали Hugging Face, что позволяет нам публиковать тонко настроенные версии модели LLM-двойника и делиться ими с читателями. Кроме того, интерфейс Hugging Face дает возможность интегрировать модель со всеми фреймворками LLM-экосистемы, например использовать Python-фреймворк Unsloth для тонкой настройки и облачную платформу SageMaker для инференса (рис. 2.1).

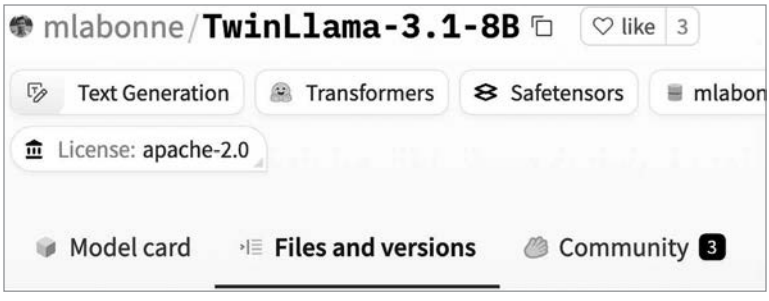


Рис. 2.1. Пример интерфейса реестра моделей Hugging Face

Наши дообученные LLM[1] опубликованы на сайте Hugging Face:

- *TwinLlama-3.1-8B* (после тонкой настройки): https://huggingface.co/mlabonne/TwinLlama-3.1-8B;
- *TwinLlama-3.1-8B DPO* (после согласования предпочтений): https://huggingface.co/mlabonne/TwinLlama-3.1-8B-DPO.

Для быстрого ознакомления с моделями мы также опубликовали их демоверсии:

- *TwinLlama-3.1-8B:* https://huggingface.co/spaces/mlabonne/TwinLlama-3.1-8B;
- *TwinLlama-3.1-8B DPO:* https://huggingface.co/spaces/mlabonne/TwinLlama-3.1-8B-DPO.

Большинство современных инструментов машинного обучения поддерживают функциональность реестра моделей. Например, такие платформы, как ZenML, Comet и SageMaker (которые будут рассмотрены позже), также предоставляют встроенные реестры моделей. Все это приемлемые варианты, однако мы выбрали Hugging Face из-за его обширной экосистемы, обеспечивающей простоту распространения и интеграцию с открытым программным обеспечением. Таким образом, выбор реестра моделей обычно зависит от того, насколько хорошо он интегрируется с инструментами и требованиями вашего проекта.

ZenML: оркестратор, артефакты и метаданные

ZenML служит связующим звеном между ML и MLOps. Этот инструмент предоставляет широкий набор функций, которые упрощают процессы отслеживания, воспроизведения, развертывания и сопровождения ML-пайплайнов. В основе ZenML лежит идея построения воспроизводимых рабочих процессов (workflows) машинного обучения. Он облегчает переход от исследовательских экспериментов в Jupyter-блокнотах к развертыванию модели в продакшене. ZenML помогает справляться с типичными трудностями с версионированием, воспроизведением экспериментов, организацией сложных ML-процессов, устранением разрыва между обучением и развертыванием, а также отслеживанием метаданных. К числу его основных возможностей относятся оркестрация ML-пайплайнов, хранение и версионирование результатов их работы, а также добавление метаданных к артефактам для лучшей наблюдаемости.

ZenML не просто очередная ML-платформа: он вводит концепцию *стека*, которая позволяет запускать ZenML на различных вариантах инфраструктуры. Стек дает возможность подключить ZenML к различным облачным сервисам, таким как:

- оркестратор и вычислительная платформа (например, AWS SageMaker или Vertex AI);

---

[1] Для этой книги авторы специально разработали учебную модель TwinLlama-3.1-8B на базе Llama 3.1. Примеры и подходы переносятся на современные модели.

- удаленное хранилище (например, AWS S3 или Google Cloud Storage);
- реестр контейнеров (например, Docker Registry или AWS ECR).

Таким образом, ZenML выступает в роли «клея», объединяющего всю вашу инфраструктуру и инструменты с помощью *стека*, позволяющего быстро осуществлять итеративную разработку и с легкостью наблюдать за всей ML-системой. При этом ZenML не привязывает вас к конкретному облачному провайдеру: он полностью изолирует Python-код от инфраструктуры, на которой тот выполняется. Например, при создании LLM-двойника мы использовали стек на базе AWS:

- SageMaker — в качестве оркестратора и вычислительной платформы;
- S3 — в качестве удаленного хранилища для хранения и отслеживания артефактов;
- ECR — в качестве реестра контейнеров.

Однако Python-код не содержит привязки к S3 или ECR, так как ZenML абстрагирует все детали. Это означает, что при необходимости мы можем легко перейти, например, на Google Cloud или Azure.



В книге мы сосредоточимся на трех основных составляющих ZenML: оркестраторе, артефактах и метаданных.

Локальная версия сервера ZenML устанавливается в качестве Python-пакета. То есть выполнение команды `poetry install` приводит к автоматической установке отладочного сервера ZenML, который можно использовать локально. В главе 11 мы покажем, как развернуть ML-пайплайны на платформе AWS с помощью облачной версии ZenML в бессерверном режиме.

Оркестратор

Оркестратор — это система, которая автоматизирует, планирует и координирует работу всех ML-пайплайнов. Он гарантирует, что каждый этап — будь то загрузка данных, предварительная обработка, обучение модели или ее развертывание — выполняется в нужном порядке с учетом всех зависимостей. Управление этими процессами позволяет эффективно использовать ресурсы, корректно обрабатывать сбои и повышать масштабируемость, благодаря чему даже сложные ML-пайплайны становятся более надежными и удобными в обслуживании.

Как именно ZenML работает в качестве оркестратора? ZenML оперирует такими понятиями, как *пайплайн* и *шаг*. Пайплайн — это высокоуровневый объект, состоящий из нескольких шагов. Функция становится ZenML-пайплайном, если она аннотирована с помощью `@pipeline`, и шагом, если к ней применяется декоратор `@step`. Это стандартная схема при работе с оркестраторами: у нас есть

функция верхнего уровня (часто называемая пайплайном), которая вызывает отдельные единицы обработки — шаги или задачи.

Мы реализовали один из таких ML-пайплайнов в рамках разработки LLM-двойника. В следующем фрагменте кода показан пример ZenML-пайплайна, который ищет пользователя в базе данных по его полному имени и просматривает все связанные с ним ссылки:

```
from zenml import pipeline
from steps.etl import crawl_links, get_or_create_user

@pipeline
def digital_data_etl(user_full_name: str, links: list[str]) -> None:
    user = get_or_create_user(user_full_name)
    crawl_links(user=user, links=links)
```

Запустить этот пайплайн можно с помощью команды `poetry poe run-digital-data-etl`. Для визуализации процесса выполнения шагов пайплайна перейдите к дашборду ZenML по адресу `http://127.0.0.1:8237/`, откройте вкладку Pipelines (Пайплайны) в левой части интерфейса и выберите пайплайн `digital_data_etl`, как показано на рис. 2.2.

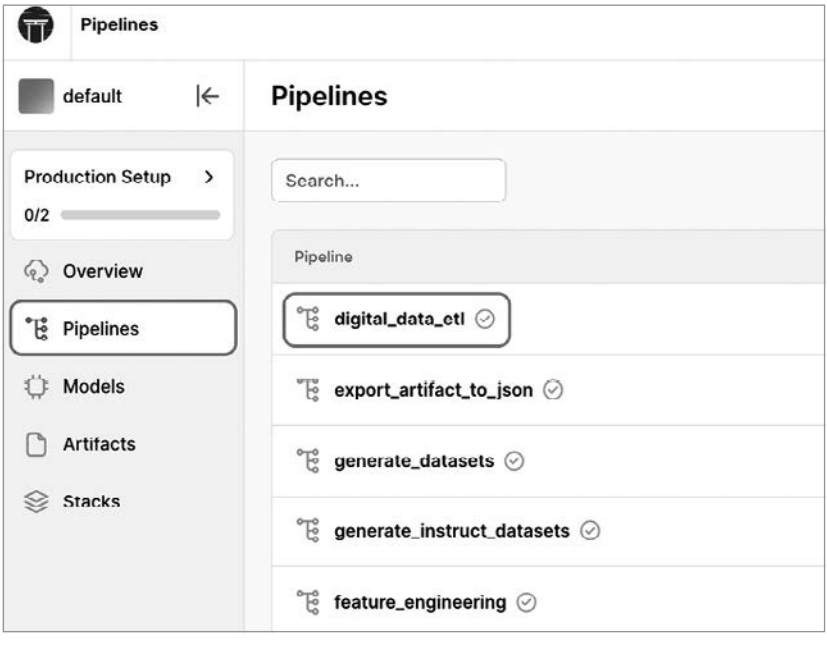


Рис. 2.2. Дашборд ZenML со списком пайплайнов

После этого вы сможете просмотреть все текущие и завершенные запуски пайплайна `digital_data_etl` (рис. 2.3). Здесь отображаются статусы выполнения

(успешно завершен, ошибка, в процессе), а также используемый стек (по умолчанию это локальный стек, применяемый при запуске ML-пайплайнов на вашей машине).



Рис. 2.3. Дашборд ZenML с информацией о запусках пайплайна digital_data_etl

Теперь, щелкнув на строке, соответствующей последнему запуску пайплайна `digital_data_etl` (или любому другому запуску, который уже успешно завершился или находится в процессе выполнения), вы увидите все шаги пайплайна, их выходные данные и дополнительную информацию (рис. 2.4). Такая структура часто называется *ориентированным или направленным ациклическим графом* (directed acyclic graph, DAG). Более подробно об этих графах мы поговорим в главе 11.

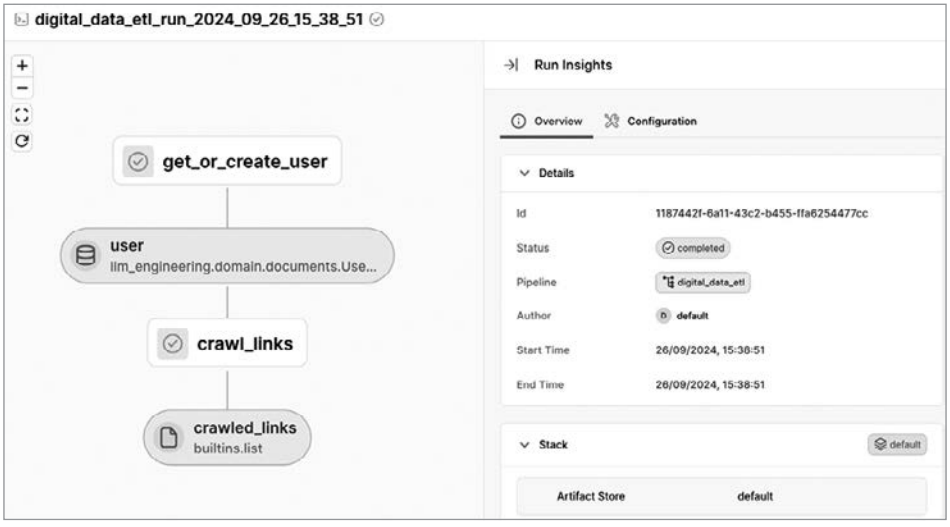


Рис. 2.4. Дашборд ZenML с информацией об отдельном запуске пайплайна digital_data_etl

Щелкнув на конкретном шаге пайплайна, вы получите дополнительную информацию о соответствующем коде и конфигурации. Кроме того, ZenML отображает агрегированные логи (рис. 2.5), сгенерированные на этом шаге, что избавляет от необходимости переключаться на другие инструменты.

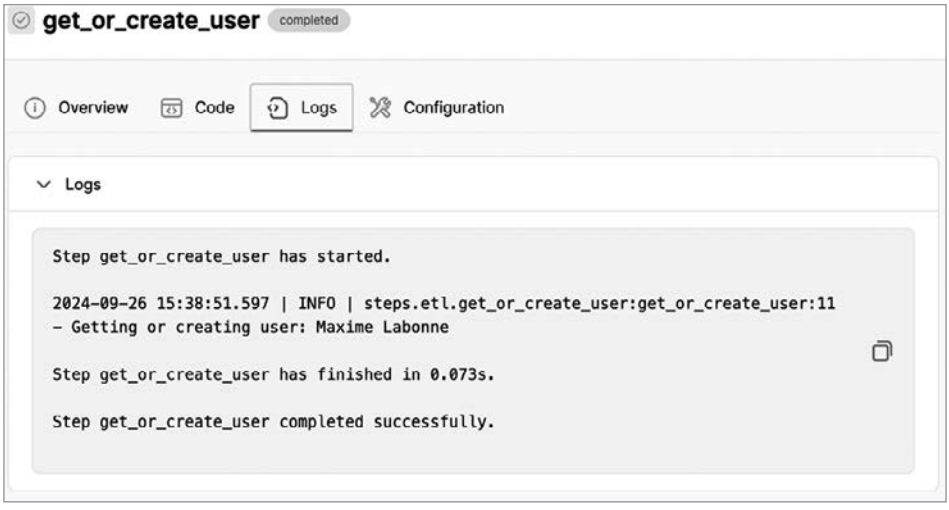


Рис. 2.5. Данные, полученные при выполнении конкретного шага пайплайна digital_data_etl

Теперь, когда мы знаем, как запустить ZenML-пайплайн и где можно найти информацию о нем, давайте разберемся с определением ZenML-шага. В приведенном ниже фрагменте кода содержится определение шага `get_or_create_user()`, который представляет собой обычную функцию Python, обернутую декоратором `@step`. Мы не будем вдаваться в детали ее реализации, так как логика ETL будет подробно рассмотрена в главе 3. Сейчас нас интересует только функциональность ZenML.

```
from loguru import logger
from typing_extensions import Annotated
from zenml import get_step_context, step

from llm_engineering.application import utils
from llm_engineering.domain.documents import UserDocument

@step
def get_or_create_user(user_full_name: str) -> Annotated[UserDocument, "user"]:
    logger.info(f"Getting or creating user: {user_full_name}")
```

```
    first_name, last_name = utils.split_user_full_name(user_full_name)

    user = UserDocument.get_or_create(first_name=first_name,
           ➥last_name=last_name)

    return user
```

В рамках шага ZenML можно реализовать любую Python-логику, соответствующую вашему сценарию. В приведенном примере мы просто создаем или получаем пользователя, но этот код можно заменить любой другой задачей — от сбора данных до подготовки признаков и обучения модели. Для интеграции ZenML необходимо писать модульный код, где каждая функция отвечает за выполнение одной конкретной задачи. Такая модульность позволяет легко аннотировать функции с помощью `@step`, а затем объединять их в пайплайн, обернутый декоратором `@pipeline`. Одно из главных архитектурных решений при работе с ZenML заключается в выборе уровня детализации шагов, поскольку при развертывании в облаке каждый шаг выполняется как отдельная задача на выделенном узле.

Чтобы отвязать свой код от ZenML, мы выделили всю бизнес-логику и логику приложения в Python-модуль `llm_engineering`. Отдельно были созданы папки `pipelines` и `steps` с логикой, специфичной для ZenML. Модуль `steps` импортирует из `llm_engineering` только необходимое (как при подключении обычного Python-пакета). В модуле `pipelines` шаги ZenML объединяются в общий пайплайн. Благодаря такой структуре можно легко заменить ZenML любым оркестратором или повторно использовать логику приложения в других сценариях, например в REST API. Для этого потребуется изменить только ZenML-код, не затрагивая модуль `llm_engineering`, где сосредоточена вся логика.

Описанную выше структуру папок можно увидеть в корневом каталоге репозитория `LLM-Engineers-Handbook` (рис. 2.6).

При написании ZenML-шагов следует учитывать еще один момент: если шаг возвращает значение, оно должно быть сериализуемым. ZenML способен сериализовать большинство объектов, которые можно свести к примитивным типам данных, но есть и исключения. Например, в нашем коде в качестве идентификаторов используются объекты UUID, однако их нативная поддержка в ZenML пока не реализована. Поэтому нам пришлось добавить сериализатор для ZenML. Мы сообщили об этой проблеме разработчикам ZenML, так что в будущих версиях можно ожидать поддержку типов UUID. Тем не менее данный пример прекрасно продемонстрировал важность сериализации при преобразовании выходных значений функций в артефакты.

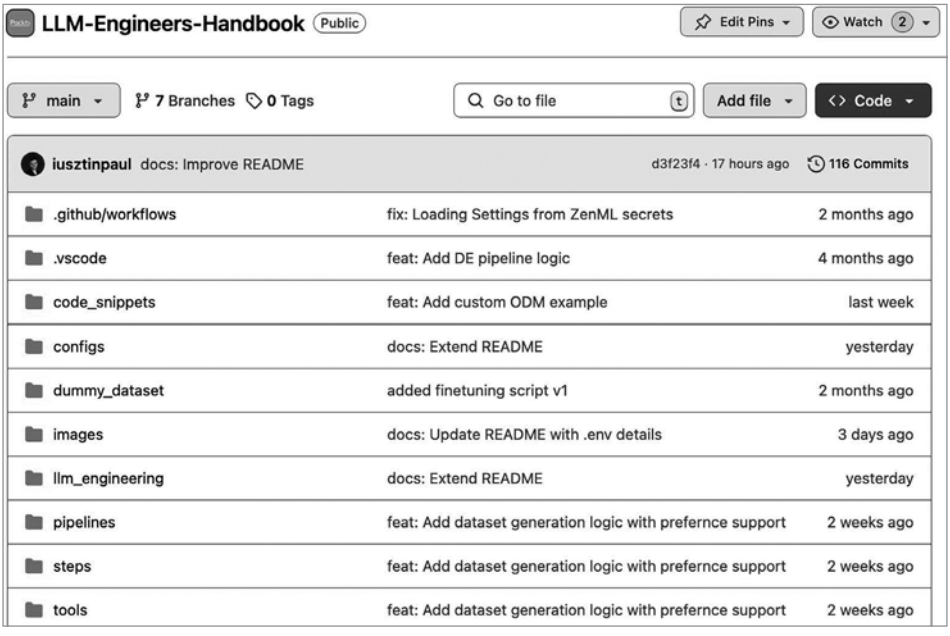


Рис. 2.6. Структура папок репозитория LLM-Engineers-Handbook

Артефакты и метаданные

Как упоминалось в предыдущем разделе, ZenML преобразует вывод каждого шага в артефакт. Прежде чем продолжить, кратко поясним, что это такое. В контексте MLOps *артефактом* называется любой файл (или набор файлов), создаваемый в ходе жизненного цикла ML-системы: это могут быть наборы данных, обученные модели, контрольные точки или логи. Артефакты крайне важны для воспроизводимости экспериментов и развертывания моделей. В артефакт можно преобразовать практически все. Так, частным случаем использования артефакта является реестр моделей. Артефакты обладают следующими уникальными свойствами: они версионируются, могут передаваться другим пользователям и сопровождаются метаданными, предоставляющими общие сведения об их содержимом. Например, обернув набор данных в артефакт, вы можете указать в его метаданных размер набора, соотношение между обучающей и тестовой выборками, размер и типы меток, а также любую другую информацию, позволяющую получить представление о содержимом набора без его загрузки.

Вернемся к примеру с пайплайном `digital_data_etl`, выходом одного из шагов которого стал артефакт в виде списка ссылок (рис. 2.7).

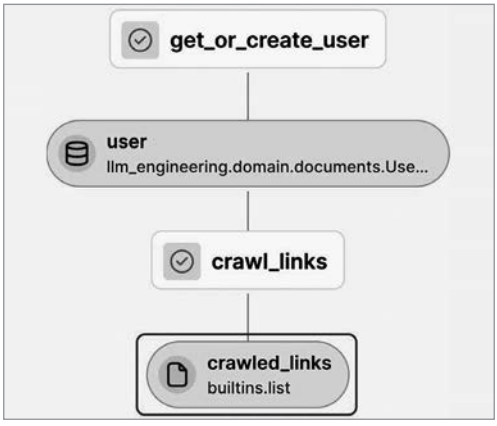


Рис. 2.7. Артефакт ZenML на примере пайплайна digital_data_etl

Если щелкнуть на артефакте `crawled_links` и перейти на вкладку Metadata (Метаданные), можно увидеть все домены, с которых производился сбор данных конкретного автора, число ссылок, просмотренных в каждом домене, а также количество успешно завершенных проходов (рис. 2.8).

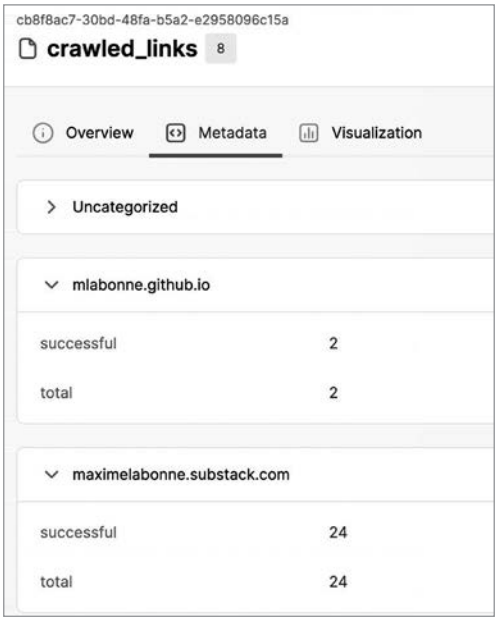


Рис. 2.8. Метаданные ZenML на примере пайплайна digital_data_etl

Более интересным примером артефакта является сгенерированный набор данных. На рис. 2.9 показаны метаданные артефакта `instruct_datasets`, который был создан автоматически для последующей тонкой настройки модели LLM-двойника. Подробно о наборах данных с инструкциями мы поговорим в главе 5. Сейчас важно отметить лишь то, что в метаданных этого набора содержатся такие ключевые сведения, как число категорий данных, занимаемый объем памяти, а также количество примеров в обучающей и тестовой выборках.

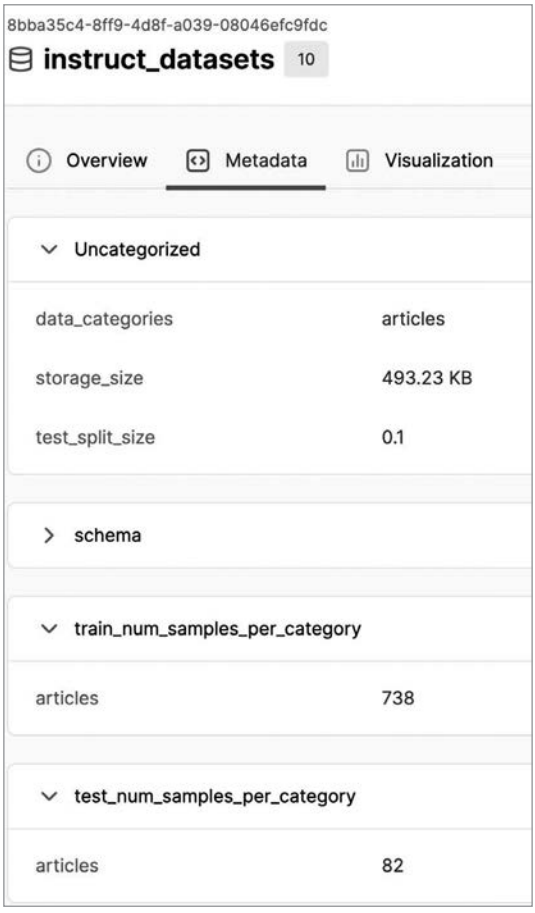


Рис. 2.9. Метаданные ZenML на примере артефакта instruct_datasets

Метаданные добавляются в артефакт вручную, как показано в приведенном ниже фрагменте кода. Таким образом, вы можете заранее вычислить и включить в метаданные любую информацию, которая будет полезна для поиска и по-

вторного использования наборов данных при реализации различных проектов и решении бизнес-задач.

```
... # Дополнительные инструкции импорта
from zenml import ArtifactConfig, get_step_context, step
@step

def generate_intruction_dataset(
    prompts: Annotated[dict[DataCategory,
    ➥list[GenerateDatasetSamplesPrompt]], "prompts"]) -> Annotated[
    InstructTrainTestSplit,
    ArtifactConfig(
        name="instruct_datasets",
        tags=["dataset", "instruct", "cleaned"],
    ),
]:
    datasets = ... # Генерация наборов данных

    step_context = get_step_context()
    step_context.add_output_metadata(output_name="instruct_datasets",
    ➥metadata=_get_metadata_instruct_dataset(datasets))

    return datasets

def _get_metadata_instruct_dataset(datasets: InstructTrainTestSplit) ->
➥dict[str, Any]:
    instruct_dataset_categories = list(datasets.train.keys())
    train_num_samples = {
        category: instruct_dataset.num_samples for category,
        ➥instruct_dataset in datasets.train.items()
    }
    test_num_samples = {category: instruct_dataset.num_samples for category,
    ➥ instruct_dataset in datasets.test.items()}

    return {
        "data_categories": instruct_dataset_categories,
        "test_split_size": datasets.test_split_size,
        "train_num_samples_per_category": train_num_samples,
        "test_num_samples_per_category": test_num_samples,
    }
```

Кроме того, вы можете легко загрузить и получить доступ к конкретной версии набора данных, используя его *универсальный уникальный идентификатор*, который можно найти на дашборде ZenML или с помощью инструмента CLI (Command Line Interface — «интерфейс командной строки»):

```
from zenml.client import Client

artifact = Client().get_artifact_version('8bba35c4-8ff9-4d8f-a039-08046efc9fdc')
loaded_artifact = artifact.load()
```

Теперь рассмотрим процесс запуска и настройки пайплайна ZenML.

Запуск и настройка пайплайна ZenML

Все пайплайны ZenML можно запускать из файла `run.py`, расположенного в каталоге `tools/run.py` репозитория на GitHub. В этом файле реализован простой интерфейс командной строки, позволяющий указать, какой именно пайплайн следует запустить.

Например, чтобы запустить пайплайн `digital_data_etl` для сбора информации о Максиме, достаточно выполнить команду:

```
python -m tools.run --run-etl --no-cache --etl-config-filename
➥ digital_data_etl_maxime_labonne.yaml
```

Для Пола:

```
python -m tools.run --run-etl --no-cache --etl-config-filename
➥ digital_data_etl_paul_iusztin.yaml
```

Как уже упоминалось в разделе, посвященном инструменту Poe the Poet, все консольные команды, используемые для взаимодействия с проектом, исполняются через Poe, чтобы упростить и стандартизировать это взаимодействие. По этой причине приведенные выше вызовы были инкапсулированы в следующие команды:

```
poetry poe run-digital-data-etl-maxime
poetry poe run-digital-data-etl-paul
```

Единственное, что меняется при запуске команд для сбора данных разных авторов, — это имя конфигурационного файла. ZenML позволяет динамически передавать такие файлы во время выполнения:

```
config_path = root_dir / "configs" / etl_config_filename
assert config_path.exists(), f"Config file not found: { config_path }"
run_args_etl = {
    "config_path": config_path,
    "run_name": f"digital_data_etl_run_{dt.now().
    ➥strftime('%Y_%m_%d_%H_%M_%S')}"
}
digital_data_etl.with_options()(**run_args_etl)
```

В конфигурационном файле указываются все параметры, передаваемые в пайплайн. Например, файл `configs/digital_data_etl_maxime_labonne.yaml` выглядит так:

```
parameters:
    user_full_name: Maxime Labonne # [Имя (имена)] [Фамилия]
    links:
      # Личный блог
      - https://mlabonne.github.io/blog/posts/2024-07-29_Finetune_Llama31.html
```

```
    - https://mlabonne.github.io/blog/posts/2024-07-
    ➥15_The_Rise_of_Agentic_Data_Generation.html
    # Substack
    - https://maximelabonne.substack.com/p/uncensor-any-llm-
    ➥withabliteration-d30148b7d43e
    ... # Прочие ссылки
```

Сигнатура функции `digital_data_etl` выглядит следующим образом:

```
@pipeline
def digital_data_etl(user_full_name: str, links: list[str]) -> str:
```

Такой подход позволяет настраивать каждый пайплайн во время выполнения без изменения исходного кода, а также четко отслеживать входные данные, обеспечивая тем самым воспроизводимость. Как показано на рис. 2.10, для каждого пайплайна может быть предусмотрен один или несколько конфигурационных файлов.

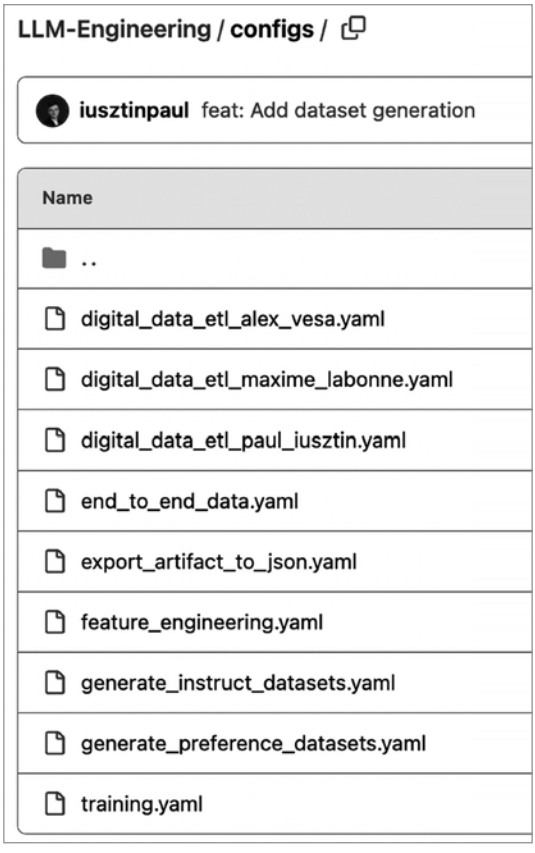


Рис. 2.10. Конфигурационные файлы пайплайнов ZenML

Среди других популярных оркестраторов, которые мы протестировали и сочли довольно мощными, можно отметить Airflow, Prefect, Metaflow и Dagster. Если вы активно пользуетесь Kubernetes, возможно, вам подойдут Argo Workflows или Kubeflow (последняя система работает исключительно поверх Kubernetes). Однако ZenML, на наш взгляд, представляет собой оптимальный компромисс между удобством, функциональностью и стоимостью.

В главе 11 мы подробнее рассмотрим использование оркестраторов для реализации передовых практик MLOps. А сейчас давайте поговорим о системе отслеживания экспериментов.

Comet ML: система отслеживания экспериментов

Обучение ML-моделей — это итеративный процесс. В отличие от традиционной разработки ПО он предполагает проведение множества параллельных экспериментов, сравнение их результатов с помощью заранее определенных метрик и выбор наилучшего кандидата для развертывания в продакшене. Инструмент для отслеживания экспериментов (или трекер) позволяет зафиксировать всю необходимую информацию — от метрик до визуальных представлений предсказаний модели, что облегчает сравнение и выбор лучшего варианта. Наш LLM-проект не является исключением.

Как показано на рис. 2.11, мы использовали систему Comet для отслеживания таких метрик, как функции потерь и значения нормы градиента в ходе всех экспериментов.

С помощью системы отслеживания экспериментов можно записывать в журнал не только метрики обучения и оценки, но и гиперпараметры, чтобы отслеживать различия в конфигурациях между экспериментами.

Кроме того, Comet автоматически фиксирует такие системные метрики, как уровень загрузки GPU, CPU и оперативной памяти, что дает целостное представление о ресурсах, необходимых для обучения модели, и помогает выявлять узкие места, замедляющие данный процесс (рис. 2.12).

При этом устанавливать Comet локально нет необходимости. В книге мы будем работать с онлайн-версией, которую можно использовать бесплатно и без ограничений. Кроме того, если вы хотите подробнее изучить функциональность системы Comet ML, вы можете получить доступ к экспериментам, которые мы отслеживали с ее помощью во время тонкой настройки моделей LLM-двойника, по следующей ссылке: https://www.comet.com/mlabonne/llm-twin-training/view/new/panels.

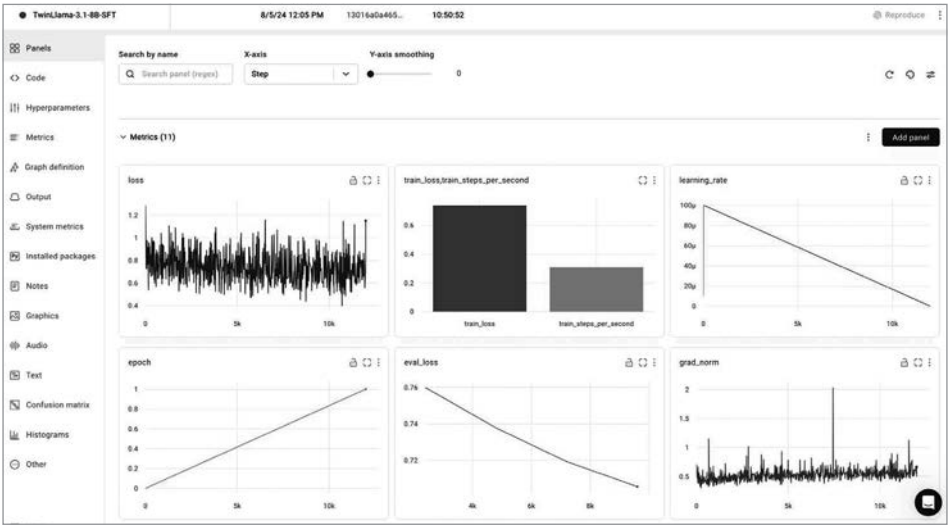


Рис. 2.11. Пример метрик обучения в Comet ML

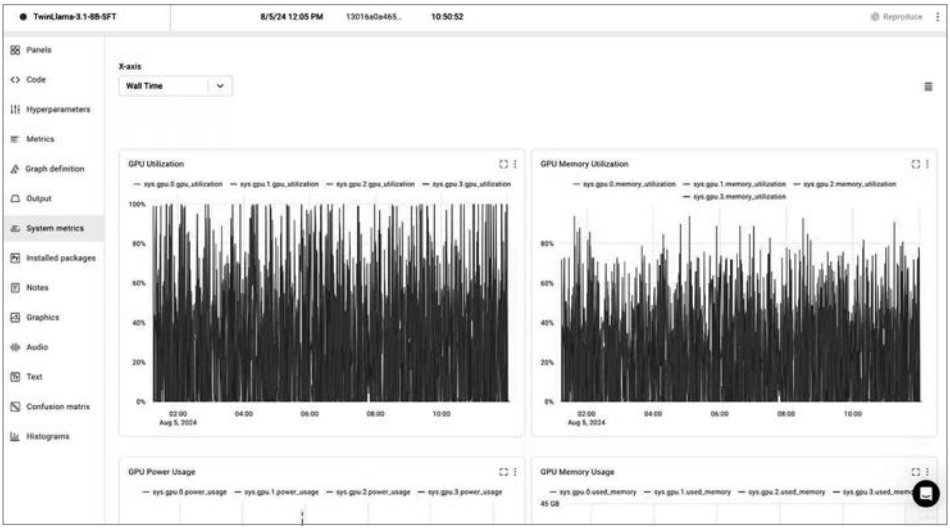


Рис. 2.12. Пример системных метрик в Comet ML

Среди других популярных средств отслеживания экспериментов выделяются W&B, MLflow и Neptune. Мы работали со всеми этими инструментами и можем сказать, что в целом их возможности сопоставимы, однако Comet

ML проще в использовании и обладает более понятным интерфейсом. Теперь обсудим последний компонент MLOps-платформы — систему Opik для мониторинга промптов.

Opik: мониторинг промптов

Для регистрации и мониторинга промптов нельзя использовать стандартные инструменты и методы. Причину этого мы подробно рассмотрим в главе 11, а пока лишь отметим, что промпты представляют собой сложные неструктурированные цепочки. При взаимодействии с приложением на основе LLM несколько входных промптов и сгенерированные ответы объединяются в последовательность (trace), в которой каждый следующий промпт зависит от предыдущих.

Поэтому вместо простых текстовых логов необходим инструмент, позволяющий группировать такие последовательности интуитивно понятным образом и отображать их на специализированном дашборде, упрощающем отладку и мониторинг промптов.

В качестве системы мониторинга промптов мы использовали Opik — инструмент с открытым исходным кодом, созданный компанией Comet. Он соответствует общей философии Comet, основу которой составляют простота и удобство использования, что по-прежнему является редкостью в сфере LLM. Существуют и другие инструменты с аналогичным функционалом, в частности свободно распространяемый Langfuse (https://langfuse.com) и проприетарные Galileo (rungalileo.io) и LangSmith (https://www.langchain.com/langsmith), однако на практике они оказались более громоздкими в использовании и реализации. Opik также предлагает облачную версию и полностью открытое решение, которое вы можете развернуть и контролировать самостоятельно. Подробнее о проекте Opik можно прочитать здесь: https://github.com/comet-ml/opik.

Базы данных для хранения неструктурированных и векторных данных

Теперь обсудим NoSQL и векторные БД, которые задействованы в наших примерах. При локальной работе они интегрируются с помощью Docker. Таким образом, при выполнении команды `poetry poe local-infrastructure-up`, как описано в одном из предыдущих разделов, локальные Docker-образы с обеими базами данных будут автоматически загружены и запущены на вашей машине. Кроме того, при развертывании проекта мы покажем, как использовать облачные версии этих БД и интегрировать их с остальными компонентами LLM-двойника.

MongoDB: NoSQL-база данных

MongoDB — это одна из самых популярных, производительных и функциональных NoSQL-баз данных на сегодняшний день. Она хорошо интегрируется с большинством облачных экосистем, включая AWS, Google Cloud, Azure и Databricks. Поэтому в качестве нашей NoSQL-базы данных мы выбрали именно ее.

MongoDB активно используется такими крупными компаниями, как Novo Nordisk, Delivery Hero, Okta и Volvo. Столь широкая распространенность говорит о том, что MongoDB еще долго будет оставаться одной из ведущих NoSQL-систем.

В рамках нашего проекта мы используем MongoDB для хранения исходных данных, собираемых из Интернета, вплоть до их обработки и передачи в векторную БД. Поскольку мы работаем с неструктурированными текстовыми данными, гибкость NoSQL-архитектуры идеально подходит под наши задачи.

Qdrant: векторная база данных

Qdrant (https://qdrant.tech/) — это одна из самых популярных, устойчивых и функциональных векторных баз данных. Для создания нашего небольшого MVP мы могли бы выбрать практически любую векторную БД, однако стремились применить легкое и перспективное решение, которое, скорее всего, будет востребовано в индустрии в течение многих лет.

В рамках этого проекта мы будем использовать Qdrant для хранения данных, поступающих из MongoDB после их обработки и преобразования для задач генеративного ИИ.

Среди компаний, активно работающих с Qdrant, — Disney, Microsoft, Discord и Johnson & Johnson. Это говорит о высокой вероятности того, что Qdrant надолго сохранит свои позиции в экосистеме векторных БД.

Помимо Qdrant, популярными решениями являются Milvus, Redis, Weaviate, Pinecone, Chroma и pgvector (плагин PostgreSQL для поддержки векторных индексов). Однако, по нашему опыту, Qdrant обеспечивает наилучший баланс между количеством запросов в секунду (RPS), задержкой и временем построения индексов, что делает его оптимальным выбором для большинства приложений на базе генеративного ИИ.

Мы не будем подробно сравнивать все векторные БД, потому что это потребовало бы написания отдельной главы. Однако, если вам интересно, рекомендуем посетить ресурс Vector DB Comparison от Superlinked (https://superlinked.com/vector-db-comparison), где вы можете сравнить ведущие векторные БД по всем ключевым аспектам: от лицензии и года выпуска до функциональных возможностей, поддерживаемых моделей векторизации и фреймворков.

Подготовка к работе с AWS

В последнем разделе мы сосредоточимся на настройке аккаунта AWS (если у вас его еще нет), создании ключа доступа и установке интерфейса командной строки (CLI). Мы также обсудим сервис SageMaker и расскажем, почему используем именно его.

В качестве облачного провайдера мы выбрали AWS, поскольку это наиболее популярная платформа и мы имеем богатый опыт работы с ней. Другие крупные облачные провайдеры, такие как GCP или Azure, предлагают схожие сервисы. Выбор подходящего варианта сводится к поиску компромисса между временем разработки (и наличием соответствующего опыта), набором функций и стоимостью. Для создания нашего MVP платформа AWS стала оптимальным решением, поскольку она предоставляет все необходимые компоненты, включая S3 (объектное хранилище), ECR (реестр контейнеров) и SageMaker (вычислительные ресурсы для обучения моделей и инференса).

Настройка аккаунта AWS, ключ доступа и интерфейс командной строки

Поскольку интерфейс AWS может со временем меняться, мы рекомендуем вам воспользоваться официальной инструкцией по созданию аккаунта: https://docs.aws.amazon.com/accounts/latest/reference/manage-acct-creating.html.

После успешной регистрации вы сможете войти в консоль AWS по адресу http://console.aws.amazon.com. На экране входа нажмите кнопку Sign in using root user email (которая находится под кнопкой Sign in), затем введите адрес электронной почты и пароль от вашего аккаунта.

Далее необходимо сгенерировать ключи доступа для программного взаимодействия с сервисами AWS. Наилучший способ сделать это — создать IAM-пользователя, обладающего правами администратора. Подробности приведены в официальной инструкции AWS: https://docs.aws.amazon.com/streams/latest/dev/setting-up.html.

В продакшене рекомендуется придерживаться принципа минимальных привилегий, то есть предоставлять каждому пользователю только те разрешения, которые требуются для выполнения его задач. Однако для упрощения настройки тестового аккаунта мы воспользуемся политикой `AdministratorAccess`, предоставляющей полный доступ, как показано на рис. 2.13.

Теперь необходимо сгенерировать ключ доступа для вновь созданного IAM-пользователя с помощью официальной инструкции: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html.

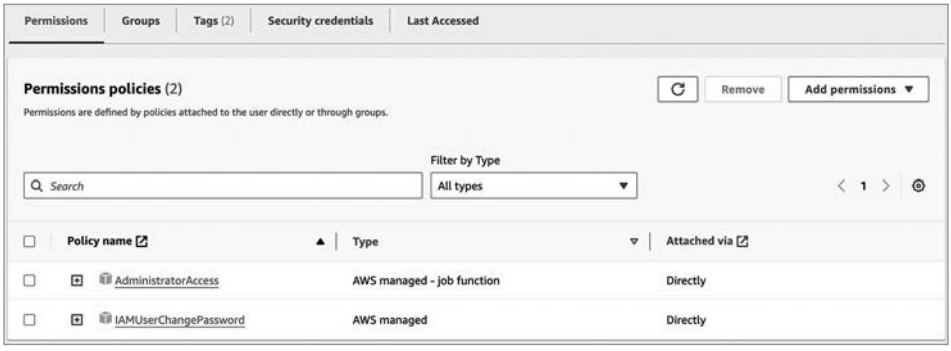


Рис. 2.13. Пример политик доступа для IAM-пользователя

Эти ключи будут выглядеть примерно так:

```
aws_access_key_id = <идентификатор_ключа_доступа>
aws_secret_access_key = <секретный_ключ_доступа>
```

Обязательно сохраните их в безопасном месте, так как после создания вы больше не сможете их просмотреть. Кроме того, будьте осторожны с тем, кому вы передаете эти ключи, поскольку они предоставляют доступ к вашему аккаунту AWS и позволяют управлять различными ресурсами.

Теперь осталось установить AWS CLI по ссылке https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html и настроить его с помощью созданных ключей.

После установки выполните команду `aws configure` для настройки интерфейса. Ниже приведен пример нашей конфигурации AWS CLI:

```
[default]
aws_access_key_id = *************
aws_secret_access_key = ************
region = eu-central-1
output = json
```

Дополнительные сведения о настройке AWS CLI доступны в официальной документации: https://docs.aws.amazon.com/cli/v1/userguide/cli-configure-files.html.

Кроме того, для настройки проекта с использованием ваших учетных данных AWS вам необходимо задать следующие переменные в файле `.env`:

```
AWS_REGION="eu-central-1" # Замените своим регионом AWS.
                          # По умолчанию "eu-central-1"
AWS_ACCESS_KEY="<ключ_доступа_aws>"
AWS_SECRET_KEY="<секретный_ключ_aws>"
```



Важное замечание о стоимости выполнения практических заданий

Все облачные сервисы, используемые в книге, предоставляют бесплатный доступ к базовой функциональности, за исключением AWS. Поэтому при реализации описанного проекта вам придется самостоятельно нести все связанные с этим расходы. Какие-то из сервисов AWS могут быть доступны бесплатно (AWS Free Tier), но это касается не всех. Мы настоятельно рекомендуем регулярно проверять консоль выставления счетов.

Бóльшая часть расходов будет связана с сервисом SageMaker для обучения модели и инференса. Ориентировочная стоимость услуг AWS при использовании конфигураций, приведенных в книге и репозитории, может составить от 50 до 100 долларов США.

Советуем ознакомиться с документацией по настройке уведомлений о расходах: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/monitor_estimated_charges_with_cloudwatch.html.

SageMaker: вычислительная платформа для обучения и инференса

Наконец, обсудим сервис SageMaker и причины, по которым мы выбрали именно его. SageMaker — это ML-платформа для обучения и развертывания моделей. Согласно официальному определению, AWS SageMaker — это полностью управляемый ML-сервис от AWS, позволяющий разработчикам и специалистам по работе с данными создавать, обучать и развертывать масштабируемые ML-модели. AWS SageMaker берет на себя все инфраструктурные задачи и позволяет пользователям сосредоточиться на разработке высококачественных моделей.

В рамках нашего проекта мы будем использовать SageMaker для тонкой настройки модели и развертывания пайплайна обучения в кластерах графических процессоров, а также для публикации нашей модели LLM-двойника в виде REST API, доступного в реальном времени из любой точки мира.

Почему именно AWS SageMaker. Итак, давайте обсудим, почему мы выбрали AWS SageMaker, а не более простые и экономичные решения, такие как Amazon Bedrock. Начнем с краткого обзора Bedrock и его преимуществ.

Amazon Bedrock — это бессерверное решение для развертывания больших языковых моделей. Бессерверная архитектура означает, что разработчику не нужно управлять серверами или инфраструктурой. Bedrock предоставляет доступ к заранее обученным моделям напрямую через вызовы API. На момент написания книги данный сервис поддерживал довольно ограниченный набор моделей: Mistral, Flan, Llama 2 и Llama 3. Вы можете отправлять входные данные и получать предсказания от модели, не заботясь о конфигурации программного или аппаратного обеспечения. Такой подход значительно упрощает и ускоряет

процесс интеграции ИИ в приложения, что особенно ценно для разработчиков, не имеющих глубоких знаний в области машинного обучения. Однако подобная простота достигается за счет ограниченных возможностей в плане настройки, поскольку вы вынуждены довольствоваться набором моделей и API, предоставляемым сервисом Amazon Bedrock. Что касается ценообразования, то Bedrock использует модель оплаты по количеству API-вызовов, благодаря которой легче прогнозировать и контролировать расходы.

В свою очередь, SageMaker представляет собой полнофункциональную платформу для создания, обучения и развертывания ML-моделей. Она позволяет полностью контролировать процесс обучения или использовать платформу для проведения исследований. Именно поэтому SageMaker востребована среди специалистов по работе с данными и экспертов в области машинного обучения, которые умеют программировать, разбираются в концепциях ML и уверенно обращаются с такими облачными платформами, как AWS. SageMaker, как и большинство сервисов AWS, использует модель оплаты «по факту», то есть вы платите за вычислительные ресурсы, хранилище и другие сервисы, задействованные в процессе разработки приложения.

В отличие от Bedrock, даже если конечная точка SageMaker фактически не используется, вы будете вынуждены оплачивать ресурсы AWS, например активные экземпляры EC2. В связи с этим крайне важно проектировать системы с автомасштабированием, которые самостоятельно удаляют неиспользуемые ресурсы. В заключение следует отметить, что Bedrock — это готовое решение, позволяющее быстро развернуть API на базе одной из поддерживаемых моделей, тогда как SageMaker представляет собой полнофункциональную платформу, предоставляющую полный контроль над логикой машинного обучения.

Почему же мы выбрали SageMaker, а не Bedrock? Сервис Bedrock был бы отличным выбором для быстрого прототипирования, однако наша книга посвящена подробному рассмотрению инженерных аспектов работы с LLM, от которых Bedrock, напротив, стремится абстрагироваться. Именно поэтому мы выбрали SageMaker как платформу с высокой степенью настраиваемости, позволяющую продемонстрировать весь процесс, связанный с развертыванием модели.

Тем не менее на практике даже сервис SageMaker нельзя назвать полностью настраиваемым. Если вы хотите получить абсолютный контроль над развертыванием, используйте EKS — управляемый сервис Kubernetes от AWS. В этом случае вы получите прямой доступ к виртуальным машинам и сможете в полной мере контролировать построение ML-пайплайнов, их взаимодействие и использование ресурсов. Аналогичных целей можно достичь и с помощью ECS — альтернативы Kubernetes от AWS. Кроме того, работа с EKS или ECS может оказаться более выгодной, поскольку услуги этих сервисов стоят существенно дешевле.

Таким образом, выбор SageMaker представляет собой разумный компромисс между возможностями настройки, с одной стороны, и удобством использования управляемого сервиса, абстрагирующего инфраструктурную сложность, — с другой. Такой подход позволяет получить необходимый уровень контроля и при этом воспользоваться преимуществами готовой платформы.

Резюме

В этой главе мы рассмотрели основные инструменты, используемые в книге. Сначала мы обсудили установку корректной версии Python, поддерживаемой нашим репозиторием, далее создали виртуальное окружение и установили все зависимости с помощью Poetry. После этого вы познакомились с Poe the Poet — инструментом для агрегации и запуска задач, необходимых для работы приложения.

Затем мы рассмотрели инструменты, обеспечивающие реализацию передовых практик MLOps: реестр моделей для обмена результатами обучения, средство отслеживания экспериментов, оркестратор для управления ML-пайплайнами и артефактами, а также метаданные для управления файлами и наборами данных. Мы также определили, какие базы данных нам необходимы для реализации LLM-двойника, и обсудили процесс настройки аккаунта AWS, создания ключа доступа и конфигурирования интерфейса AWS CLI для программного доступа к облаку. Наконец, мы подробно рассмотрели сервис AWS SageMaker и объяснили причины, по которым он был выбран в качестве вычислительной платформы для обучения и развертывания модели LLM-двойника.

В следующей главе мы перейдем к практической реализации нашего проекта, начав с ETL-пайплайна, который извлекает посты, статьи и репозитории из Интернета и сохраняет их в хранилище данных.

Список используемых источников

- *Acsany P.* Dependency Management With Python Poetry. Real Python. https://realpython.com/dependency-management-python-poetry/.
- Comet.ml. comet-ml/opik: Open-source end-to-end LLM Development Platform. GitHub. https://github.com/comet-ml/opik.
- *Czakon J.* ML Experiment Tracking: What It Is, Why It Matters and How to Implement It. neptune.ai. https://neptune.ai/blog/ml-experiment-tracking.
- Hopsworks. ML Artifacts (ML Assets)? Hopsworks. https://www.hopsworks.ai/dictionary/ml-artifacts.
- Poetry. Documentation | Poetry — Python dependency management and packaging made easy. https://python-poetry.org/docs.

- *Jones L.* Managing Multiple Python Versions With pyenv. Real Python. https://realpython.com/intro-to-pyenv/.
- *Kaewsanmua K.* Best Machine Learning Workflow and Pipeline Orchestration Tools. neptune.ai. https://neptune.ai/blog/best-workflow-and-pipeline-orchestration-tools.
- MongoDB. What is NoSQL? NoSQL databases explained. https://www.mongodb.com/resources/basics/databases/nosql-explained.
- Nat-N. nat-n/poethepoet: A task runner that works well with poetry. GitHub. https://github.com/nat-n/poethepoet.
- *Oladele S.* ML Model Registry: The Ultimate Guide. neptune.ai. https://neptune.ai/blog/ml-model-registry.
- *Schwaber-Cohen R.* What is a Vector Database & How Does it Work? Use Cases + Examples. Pinecone. https://www.pinecone.io/learn/vector-database/.
- Superlinked. Vector DB Comparison. https://superlinked.com/vector-db-comparison.

3 Инженерия данных

В этой главе мы более подробно обсудим проект LLM-двойника. Мы научимся проектировать и реализовывать пайплайн сбора данных, необходимый для всех вариантов использования LLM, таких как тонкая настройка и инференс. Поскольку книга не посвящена инженерии данных, в этой главе мы поговорим только о тех аспектах, которые действительно необходимы для сбора исходной информации. В главе 4 перейдем к рассмотрению теоретических основ и деталей практической реализации больших языковых моделей и генеративного ИИ.

Во многих учебных проектах и исследовательских работах обычно используется уже готовый статический набор данных. Однако мы стремимся смоделировать реальную ситуацию, в которой нам необходимо собрать и подготовить данные самостоятельно. Таким образом, реализация собственного пайплайна данных позволит на практике понять, как выглядит полный цикл машинного обучения. В этой главе мы рассмотрим процесс проектирования и реализации *пайплайна ETL* (extract, transform, load — «извлечение, преобразование, загрузка»), который анализирует несколько онлайн-платформ, таких как Medium, Substack и GitHub, и сохраняет собранные данные в хранилище MongoDB. Мы покажем, как реализовать различные методы просмотра, привести данные к стандартному формату и загрузить их в хранилище.

Начнем с проектирования пайплайна сбора данных для LLM-двойника и рассмотрим архитектуру ETL-системы. Затем приступим к реализации, начиная с ZenML, который будет отвечать за оркестрацию всего процесса. Мы разберем устройство краулера и напишем код, отвечающий за выбор нужного экземпляра краулера в зависимости от домена переданной ссылки, соблюдая при этом лучшие практики разработки ПО. После этого мы поочередно реализуем каждый краулер и добавим слой работы с данными, чтобы структурировать документы и взаимодействовать с MongoDB.

Наконец, мы покажем, как запускать пайплайн сбора данных с помощью ZenML и извлекать собранные данные из MongoDB.

К концу главы вы будете знать, как спроектировать и реализовать ETL-пайплайн, который извлекает, преобразует и загружает исходные данные, пригодные для последующего использования в ML-приложении.

Проектирование пайплайна сбора данных для LLM-двойника

Прежде чем переходить к реализации, необходимо разобраться в архитектуре ETL-пайплайна для LLM-двойника, представленной на рис. 3.1. Для этого нам предстоит определить, с каких платформ будут извлекаться данные, а также продумать их структуру и организацию процессов обработки. Однако первым делом нам нужно понять, как наш пайплайн сбора данных соотносится с классическим ETL-процессом.

Пайплайн ETL включает три этапа.

1. *Извлечение* данных из различных источников (в нашем случае — Medium, Substack и GitHub).
2. *Преобразование* данных, то есть очистка и стандартизация с приведением к единому формату, пригодному для хранения и анализа.
3. *Загрузка* преобразованных данных в хранилище или базу данных.

В рамках нашего проекта в качестве NoSQL-хранилища данных используется MongoDB. Этот подход не является стандартным, но причины такого решения мы объясним чуть позже.

Наша задача — спроектировать ETL-пайплайн, на вход которого подается имя пользователя и список ссылок. Далее система просматривает и анализирует каждую ссылку по отдельности, стандартизирует собранный контент и сохраняет его в MongoDB под именем соответствующего автора.

В результате сигнатура пайплайна сбора данных будет выглядеть следующим образом:

- *входные данные* — список ссылок и связанный с ними пользователь (автор);
- *выходные данные* — список исходных документов, помещенных в хранилище NoSQL.

В дальнейшем мы будем использовать термины «пользователь» и «автор» как синонимы, поскольку в большинстве случаев пользователь является автором извлекаемого контента. Однако в самой базе данных (MongoDB) вся информация сохраняется в единственной коллекции — `user`.

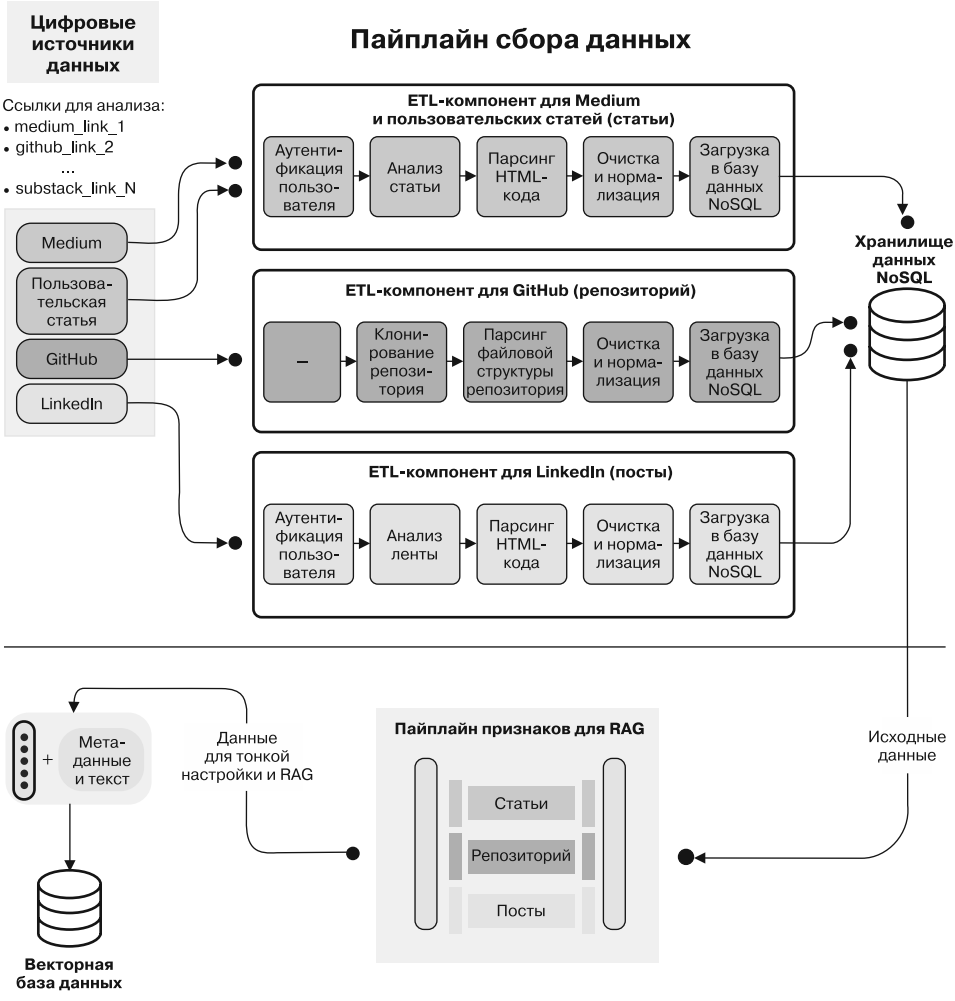


Рис. 3.1. Архитектура ETL-пайплайна сбора данных для LLM-двойника

ETL-пайплайн будет определять домен каждой ссылки и вызывать специализированный краулер. Мы реализовали четыре краулера для сбора трех категорий данных, таких как статья, репозиторий (или код) и пост. Нас интересует не источник, а формат документа, от которого зависит способ его обработки. Поэтому каждая категория представлена в системе как отдельная сущность со своим классом и собственной коллекцией в MongoDB (рис. 3.2). Поскольку исходная ссылка сохраняется в метаданных документа, источник можно отследить и использовать в задачах генеративного ИИ.

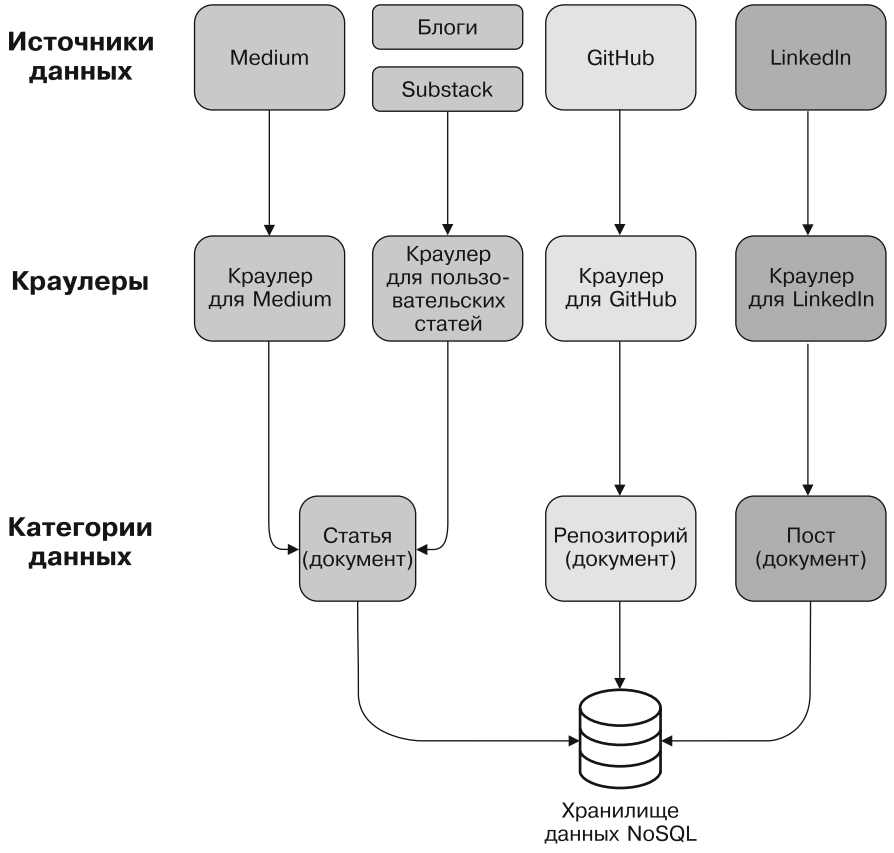


Рис. 3.2. Связь между краулерами и категориями данных

В кодовой базе нашего проекта реализованы четыре краулера.

- *Краулер для Medium* — используется для сбора данных с платформы Medium и возвращает документ типа «статья». При этом он авторизуется на платформе, просматривает HTML-страницу по указанной ссылке, извлекает, очищает и нормализует текст, после чего сохраняет его стандартизированную версию в хранилище NoSQL.
- *Краулер для пользовательских статей* — выполняет те же шаги, что и краулер для Medium, но реализован в более общем виде и не учитывает особенности конкретной платформы. Он не выполняет аутентификацию, а просто извлекает весь HTML-код по ссылке. Этого достаточно для сбора статей, опубликованных, например, на Substack или в персональных блогах. Данный краулер служит резервным вариантом на случай, если домен ссылки

не обрабатывается ни одним из специализированных краулеров. Например, при передаче ссылки на Substack будет использован именно краулер для пользовательских статей, а при передаче ссылки на Medium — специализированный краулер для Medium.

- *Краулер для GitHub* — применяется для сбора данных с платформы GitHub и возвращает документ типа «репозиторий». При этом он клонирует репозиторий, анализирует его файловую структуру, очищает и нормализует содержимое файлов, а затем сохраняет его в БД.
- *Краулер для LinkedIn* — используется для сбора данных с платформы LinkedIn и возвращает несколько документов типа «пост». При этом он авторизуется на платформе, переходит к ленте пользователя и извлекает все его последние публикации. Для каждого поста он анализирует HTML-код, очищает и нормализует содержимое, после чего сохраняет его в MongoDB.

В следующем разделе мы подробно рассмотрим реализацию каждого из краулеров. А пока просто имейте в виду, что каждый краулер по-своему получает доступ к определенной платформе или сайту и извлекает оттуда HTML-код. Затем этот код разбирается, из него извлекается текст, который очищается и нормализуется, после чего помещается в хранилище данных через единый интерфейс.

Благодаря тому что все собранные данные приводятся к одной из трех категорий и мы не создаем новую категорию для каждого источника, эту архитектуру можно легко масштабировать, добавляя новые источники с минимальными усилиями. Например, если мы решим собирать данные из конкретной соцсети, нам будет достаточно реализовать новый краулер, возвращающий документ типа «пост». Остальной код системы останется нетронутым. В противном случае, если бы мы добавили тип источника в структуру классов и документов, нам пришлось бы вносить изменения во все нижележащие уровни системы, например создавать новый класс документа для каждого источника и адаптировать пайплайн признаков для его поддержки.

Для создаваемой нами пробной реализации достаточно собрать несколько сотен документов. Однако если мы захотим масштабировать проект до полноценного продукта, нам потребуется значительно больше источников данных. Вместо нескольких сотен документов желательно иметь тысячи, чтобы получить оптимальные результаты. При реализации большинства проектов имеет смысл начать с разработки базовой версии системы, пусть и не обеспечивающей максимальную точность, а затем поэтапно улучшать ее. Благодаря такой архитектуре вы без труда сможете добавлять новые источники данных в рамках будущих итераций, формируя все более крупные наборы данных. О тонкой настройке LLM и выборе подходящего размера для набора данных мы поговорим в следующей главе.

Как ETL-пайплайн связан с пайплайном признаков? Пайплайн признаков получает исходные данные из хранилища MongoDB, выполняет дополнительную очистку, преобразует данные в признаки и сохраняет их в векторной базе данных Qdrant, откуда они могут подаваться в LLM-пайплайны обучения и инференса. Более подробно пайплайн признаков мы обсудим в главе 4. ETL-пайплайн работает независимо от пайплайна признаков. Взаимодействие между ними происходит исключительно через хранилище MongoDB. Таким образом, пайплайн сбора данных может записывать данные в MongoDB, а пайплайн признаков — считывать их оттуда независимо и по собственному расписанию.

Почему мы выбрали MongoDB в качестве хранилища данных? Использование транзакционной БД, такой как MongoDB, в качестве хранилища данных — это довольно нетипичное решение. Однако в рамках нашего проекта мы работаем с небольшими объемами данных, с которыми MongoDB справляется без особых проблем. Даже если мы захотим вычислять статистические показатели на основе коллекций MongoDB, эта БД будет достаточно эффективно обрабатывать сотни документов, используемых нашим LLM-двойником. Мы остановились на MongoDB главным образом из-за характера наших данных — неструктурированного текста, собранного из Интернета. Поэтому выбор базы данных NoSQL, не требующей заранее заданной схемы, упростил и ускорил процесс разработки. Кроме того, система MongoDB надежна и удобна в использовании. Она имеет интуитивно понятный Python SDK, Docker-образ для локального запуска, а также бесплатный облачный тариф, идеально подходящий для создания прототипов вроде нашего LLM-двойника. Таким образом, мы можем свободно работать с MongoDB как локально, так и в облаке. Однако при взаимодействии с большими объемами данных (миллионы документов и более) предпочтительнее использовать специализированные хранилища данных, такие как Snowflake или BigQuery.

Теперь, когда мы разобрались в архитектуре пайплайна сбора данных для LLM-двойника, перейдем к его реализации.

Реализация пайплайна сбора данных для LLM-двойника

Как уже упоминалось в главе 2, точкой входа в любой пайплайн проекта LLM-двойника служит пайплайн ZenML. Его можно настраивать во время выполнения с помощью YAML-файлов и запускать в экосистеме ZenML. Мы начнем с анализа пайплайна `digital_data_etl`, который приводили в качестве примера в главе 2, чтобы продемонстрировать работу ZenML. Однако теперь углубимся в детали и объясним, как устроен процесс сбора данных. После изучения структуры самого пайплайна мы рассмотрим реализацию отдельных

краулеров, применяемых для сбора данных с различных сайтов, а также структуру MongoDB-документов, используемых для хранения собранных данных и их извлечения.

Пайплайн ZenML и его шаги

Ниже приведен фрагмент кода, демонстрирующий реализацию ZenML-пайплайна `digital_data_etl`. На вход он получает полное имя пользователя и список ссылок для извлечения контента, автором которого, как предполагается, является пользователь. Внутри функции вызываются два шага: сначала наша система ищет пользователя в БД по полному имени, а затем последовательно проходит по всем переданным ссылкам. Реализация этого пайплайна доступна в репозитории проекта, в файле `pipelines/digital_data_etl.py`.

```
from zenml import pipeline

from steps.etl import crawl_links, get_or_create_user

@pipeline
def digital_data_etl(user_full_name: str, links: list[str]) -> str:
    user = get_or_create_user(user_full_name)
    last_step = crawl_links(user=user, links=links)

    return last_step.invocation_id
```

На рис. 3.3 показан дашборд ZenML с информацией о запуске пайплайна `digital_data_etl`. Далее мы подробно рассмотрим два шага этого пайплайна — `get_or_create_user` и `crawl_links`. Их реализацию можно найти в папке репозитория `steps/etl`.

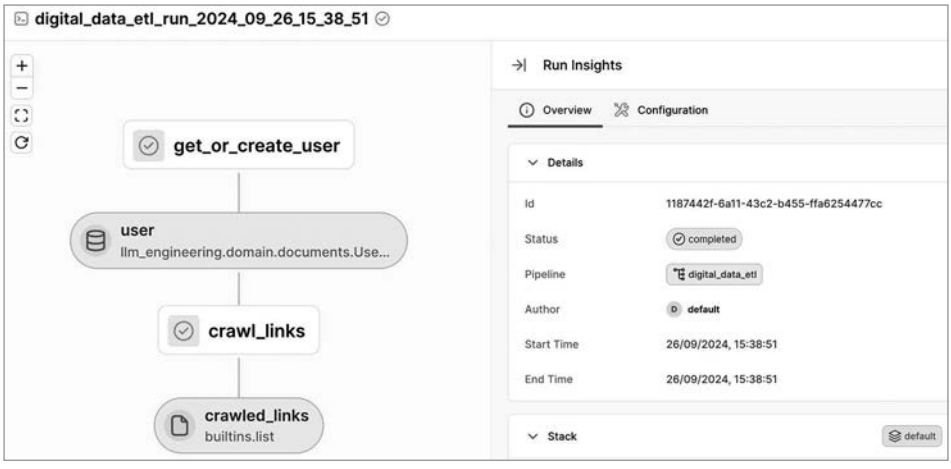


Рис. 3.3. Информация о запуске пайплайна digital_data_etl на дашборде ZenML

Начнем с шага `get_or_create_user`. Сначала импортируются все необходимые модули и функции, используемые в скрипте.

```
from loguru import logger
from typing_extensions import Annotated
from zenml import get_step_context, step

from llm_engineering.application import utils
from llm_engineering.domain.documents import UserDocument
```

Далее определяется сигнатура функции, которая принимает на вход полное имя пользователя и либо извлекает его из MongoDB, либо создает нового, если он отсутствует в базе:

```
@step
def get_or_create_user(user_full_name: str) -> Annotated[UserDocument, "user"]:
```

С помощью вспомогательной функции мы разделяем полное имя пользователя на имя и фамилию. Затем пытаемся извлечь пользователя из БД или создаем нового, если он отсутствует. Мы также извлекаем контекст текущего шага и добавляем к выходным данным метаданные о пользователе, которые будут отражены в метаданных выходного ZenML-артефакта `user`:

```
    logger.info(f"Getting or creating user: {user_full_name}")

    first_name, last_name = utils.split_user_full_name(user_full_name)

    user = UserDocument.get_or_create(first_name=first_name,
    ➥last_name=last_name)

    step_context = get_step_context()
    step_context.add_output_metadata(output_name="user",
    ➥metadata=_get_metadata(user_full_name, user))

    return user
```

Кроме того, мы определяем вспомогательную функцию `_get_metadata()`, которая формирует словарь с параметрами запроса и извлеченной информацией о пользователе (все это будет включено в метаданные артефакта `user`):

```
def _get_metadata(user_full_name: str, user: UserDocument) -> dict:
    return {
        "query": {
            "user_full_name": user_full_name,
        },
        "retrieved": {
            "user_id": str(user.id),
            "first_name": user.first_name,
            "last_name": user.last_name,
        },
    }
```

Теперь перейдем к шагу `crawl_links`, который отвечает за сбор данных, доступных по предоставленным ссылкам. Начинаем с импорта основных модулей и библиотек, необходимых для анализа веб-страниц:

```
from urllib.parse import urlparse

from loguru import logger
from tqdm import tqdm
from typing_extensions import Annotated
from zenml import get_step_context, step

from llm_engineering.application.crawlers.dispatcher import CrawlerDispatcher
from llm_engineering.domain.documents import UserDocument
```

Главная функция этого шага получает на вход список ссылок, относящихся к конкретному автору. Внутри данной функции инициализируется диспетчер краулеров, настроенный на работу с такими доменами, как LinkedIn, Medium и GitHub:

```
@step
def crawl_links(user: UserDocument, links: list[str]) ->
Annotated[list[str], "crawled_links"]:
    dispatcher = CrawlerDispatcher.build().register_linkedin().register_
    ➥medium().register_github()

    logger.info(f"Starting to crawl {len(links)} link(s).")
```

Затем инициализируются переменные для накопления метаданных и подсчета успешно обработанных ссылок, после чего функция перебирает ссылки, пытаясь извлечь данные по каждой из них, обновляя счетчик успешных попыток и пополняя словарь метаданных:

```
    metadata = {}
    successfull_crawls = 0
    for link in tqdm(links):
        successfull_crawl, crawled_domain = _crawl_link(dispatcher, link, user)
        successfull_crawls += successfull_crawl

        metadata = _add_to_metadata(metadata, crawled_domain, successfull_crawl)
```

После прохода по всем ссылкам собранные метаданные прикрепляются к выходному артефакту:

```
    step_context = get_step_context()
    step_context.add_output_metadata(output_name="crawled_links",
    ➥metadata=metadata)

    logger.info(f"Successfully crawled {successfull_crawls} / {len(links)}
    ➥ links.")

    return links
```

В коде также реализована вспомогательная функция `_crawl_link()`, которая отвечает за извлечение информации по каждой ссылке с помощью соответствующего краулера (выбираемого на основе домена). Эта функция обрабатывает исключения и возвращает кортеж с результатом анализа и доменом ссылки:

```
def _crawl_link(dispatcher: CrawlerDispatcher, link: str, user:
➥UserDocument) -> tuple[bool, str]:
    crawler = dispatcher.get_crawler(link)
    crawler_domain = urlparse(link).netloc

    try:
        crawler.extract(link=link, user=user)

        return (True, crawler_domain)
    except Exception as e:
        logger.error(f"An error occurred while crawling: {e!s}")

        return (False, crawler_domain)
```

Еще одна вспомогательная функция предназначена для обновления словаря метаданных с учетом результатов просмотра очередной ссылки:

```
def _add_to_metadata(metadata: dict, domain: str, successfull_crawl: bool)
➥ -> dict:
    if domain not in metadata:
        metadata[domain] = {}
    metadata[domain]["successful"] = metadata.get(domain, {}).get
    ➥("successful", 0) + successfull_crawl
    metadata[domain]["total"] = metadata.get(domain, {}).get("total", 0) + 1

    return metadata
```

Как видно из кода функции `_crawl_link()`, класс `CrawlerDispatcher` определяет нужный краулер на основе домена ссылки. Остальная логика инкапсулирована внутри метода `extract()` соответствующего краулера. В следующем разделе мы подробно рассмотрим реализацию класса `CrawlerDispatcher`, чтобы понять, как он работает.

Диспетчер: инициализация нужного краулера

Точкой входа в логику анализа является класс `CrawlerDispatcher`. Как показано на рис. 3.4, диспетчер выступает в роли промежуточного звена между переданными ссылками и соответствующими краулерами. Он знает, какой краулер следует использовать для просмотра каждой ссылки.

Класс `CrawlerDispatcher` умеет извлекать домен из каждой ссылки и инициализировать краулер, способный собирать данные с указанного сайта. Например,

если в переданной ссылке содержится домен `https://medium.com`, диспетчер создаст экземпляр краулера `MediumCrawler`, предназначенного для анализа этой платформы. Имея это в виду, давайте рассмотрим реализацию класса `CrawlerDispatcher`.



Вся связанная с просмотром и анализом логика содержится в папке репозитория GitHub: llm_engineering/application/crawlers.

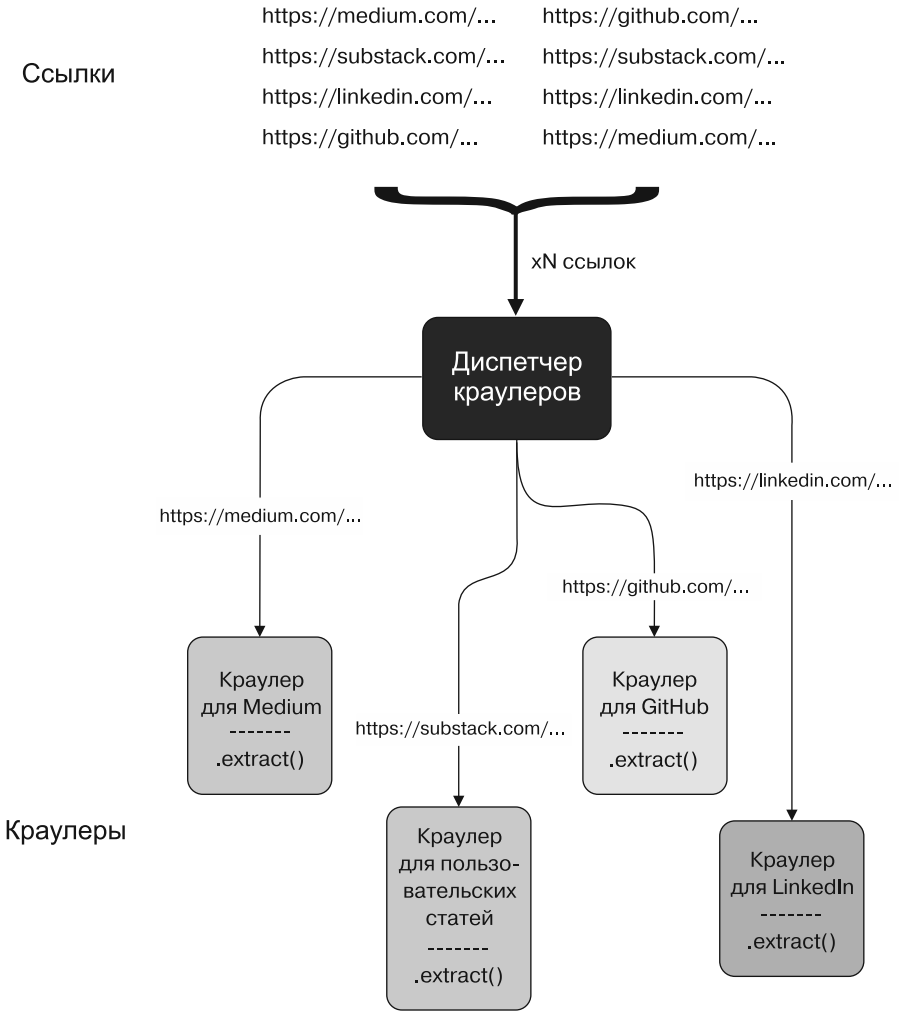


Рис. 3.4. Взаимосвязи между переданными ссылками, диспетчером CrawlerDispatcher и краулерами

Мы начинаем с импорта модулей Python для обработки URL-адресов и регулярных выражений, а также самих классов краулеров:

```
import re
from urllib.parse import urlparse

from loguru import logger

from .base import BaseCrawler
from .custom_article import CustomArticleCrawler
from .github import GithubCrawler
from .linkedin import LinkedInCrawler
from .medium import MediumCrawler
```

Класс `CrawlerDispatcher` предназначен для управления краулерами и создания соответствующих экземпляров в зависимости от домена переданной ссылки. В конструкторе этого класса создается реестр для хранения зарегистрированных краулеров:

```
class CrawlerDispatcher:
    def __init__(self) -> None:
        self._crawlers = {}
```

Поскольку для инстанцирования и настройки диспетчера используется порождающий паттерн проектирования Строитель, мы определяем метод класса `build()`, который возвращает экземпляр этого диспетчера:

```
    @classmethod
    def build(cls) -> "CrawlerDispatcher":
        dispatcher = cls()

        return dispatcher
```

Для регистрации краулеров, соответствующих отдельным платформам (например, Medium, LinkedIn или GitHub), реализованы специальные методы, использующие обобщенный метод `register()`, который добавляет каждый краулер в реестр. Возвращая объект `self`, мы следуем принципам паттерна Строитель (подробнее о котором можно прочитать здесь: https://refactoring.guru/design-patterns/builder). Мы можем объединить несколько вызовов методов `register_*()` в цепочку при создании экземпляра диспетчера следующим образом: `CrawlerDispatcher.build().register_linkedin().register_medium()`:

```
    def register_medium(self) -> "CrawlerDispatcher":
        self.register("https://medium.com", MediumCrawler)

        return self

    def register_linkedin(self) -> "CrawlerDispatcher":
        self.register("https://linkedin.com", LinkedInCrawler)

        return self
```

```
    def register_github(self) -> "CrawlerDispatcher":
        self.register("https://github.com", GithubCrawler)

        return self
```

Обобщенный метод `register()` нормализует доменное имя, чтобы обеспечить единообразие формата, и добавляет его в словарь `self._crawlers`, где домены используются в качестве ключей, по которым будет производиться сопоставление ссылок с краулерами:

```
    def register(self, domain: str, crawler: type[BaseCrawler]) -> None:
        parsed_domain = urlparse(domain)
        domain = parsed_domain.netloc

        self._crawlers[r"https://(www\.)?{}/*".format(re.escape(domain))]
        ➥ = crawler
```

Наконец, метод `get_crawler()` определяет необходимый краулер, сопоставляя ссылку с зарегистрированными доменами. Если соответствие не обнаружено, система выводит предупреждение и по умолчанию использует краулер `CustomArticleCrawler`.

```
    def get_crawler(self, url: str) -> BaseCrawler:
        for pattern, crawler in self._crawlers.items():
            if re.match(pattern, url):
                return crawler()
            else:
                logger.warning(f"No crawler found for {url}. Defaulting to
                ➥CustomArticleCrawler.")

                return CustomArticleCrawler()
```

Следующим шагом в понимании работы пайплайна сбора данных является анализ реализации каждого из краулеров.

Краулеры

Прежде чем перейти к реализации каждого из краулеров, нужно рассмотреть их базовый класс, который задает унифицированный интерфейс. Как показано на рис. 3.4, мы можем создать диспетчерский слой именно потому, что все краулеры имеют одну и ту же сигнатуру. Каждый класс выполняет метод `extract()`, что позволяет нам задействовать такие принципы объектно-ориентированного программирования, как полиморфизм, то есть работать с абстрактными объектами, не зная, к какому конкретному подклассу они относятся. Например, во вспомогательной функции `_crawl_link()` из ZenML-шага использовался следующий фрагмент кода:

```
crawler = dispatcher.get_crawler(link)
crawler.extract(link=link, user=user)
```

Обратите внимание, что мы вызвали метод `extract()`, не заботясь о конкретном типе инстанцированного краулера. Таким образом, применение абстрактных интерфейсов обеспечивает повторное использование базовой логики и упрощает расширение системы.

Базовые классы

Теперь рассмотрим интерфейс `BaseCrawler`, который можно найти в репозитории по адресу https://github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/llm_engineering/application/crawlers/base.py:

```
from abc import ABC, abstractmethod

class BaseCrawler(ABC):
    model: type[NoSQLBaseDocument]

    @abstractmethod
    def extract(self, link: str, **kwargs) -> None: ...
```

Как уже упоминалось, этот интерфейс определяет метод `extract()`, который принимает на вход ссылку. Кроме того, на уровне класса задается атрибут `model`, указывающий на тип документа, соответствующий одной из категорий данных, и используемый для сохранения извлеченного контента в хранилище MongoDB. Такой подход позволяет настраивать каждый подкласс под конкретную категорию данных, сохраняя при этом общие атрибуты на уровне класса. Класс `NoSQLBaseDocument` мы рассмотрим чуть позже, когда перейдем к сущностям, описывающим документы.

Мы также расширяем класс `BaseCrawler` с помощью класса `BaseSeleniumCrawler`, который реализует повторно используемую функциональность на базе Selenium для анализа различных сайтов, таких как Medium или LinkedIn. *Selenium* — это инструмент для автоматизированного управления веб-браузерами, позволяющий программно взаимодействовать с веб-страницами (например, входить в аккаунт LinkedIn, переходить по разделам профиля и т. д.).

Selenium предоставляет возможность программно управлять браузерами вроде Chrome, Firefox или Brave. На некоторых платформах нам необходимо использовать этот инструмент, чтобы автоматизировать такие действия, как вход в систему или прокрутка ленты новостей или статьи для извлечения полного HTML-кода с нужной страницы. С других сайтов, где вход в систему не требуется или вся страница доступна сразу, HTML-код можно извлекать более простыми средствами без использования Selenium.



Для работы краулеров на базе Selenium необходимо установить браузер Chrome (или его альтернативу на базе Chromium, например Brave).

Этот код начинается с подключения необходимых модулей и настройки окружения для обхода веб-страниц с помощью Selenium и инициализатора ChromeDriver. Библиотека `chromedriver_autoinstaller` автоматически устанавливает подходящую версию ChromeDriver и добавляет ее в системный путь, обеспечивая совместимость с установленной на машине версией браузера на базе Chromium. Selenium через ChromeDriver взаимодействует с браузером и запускает его в фоновом режиме (headless), что позволяет управлять браузером программно: переходить по URL-адресам, использовать различные элементы управления, вроде кнопок, прокручивать страницу и т. д. Благодаря `chromedriver_autoinstaller` мы можем быть уверены, что у нас установлена корректная версия ChromeDriver, соответствующая версии браузера Chrome на нашем компьютере.

```
import time
from tempfile import mkdtemp

import chromedriver_autoinstaller
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

from llm_engineering.domain.documents import NoSQLBaseDocument

# Проверяем факт наличия текущей версии chromedriver.
# Если она не установлена, автоматически загружаем ее
# и добавляем chromedriver в системный путь
chromedriver_autoinstaller.install()
```

Затем мы определяем класс `BaseSeleniumCrawler`, предназначенный для ситуаций, когда сбор данных без Selenium невозможен, как, например, в случае с платформами Medium или LinkedIn.

В конструкторе этого класса настраиваются различные параметры Chrome, оптимизирующие производительность, повышающие безопасность и обеспечивающие работу браузера в фоновом режиме. Эти настройки отключают неиспользуемые функции, такие как отрисовка графики с помощью GPU, расширения и уведомления, которые могут помешать автоматизации процесса обхода. Стандартная конфигурация для работы в фоновом режиме выглядит так:

```
class BaseSeleniumCrawler(BaseCrawler, ABC):
    def __init__(self, scroll_limit: int = 5) -> None:
        options = webdriver.ChromeOptions()

        options.add_argument("--no-sandbox")
        options.add_argument("--headless=new")
        options.add_argument("--disable-dev-shm-usage")
        options.add_argument("--log-level=3")
        options.add_argument("--disable-popup-blocking")
        options.add_argument("--disable-notifications")
        options.add_argument("--disable-extensions")
        options.add_argument("--disable-background-networking")
```

```
        options.add_argument("--ignore-certificate-errors")
        options.add_argument(f"--user-data-dir={mkdtemp()}")
        options.add_argument(f"--data-path={mkdtemp()}")
        options.add_argument(f"--disk-cache-dir={mkdtemp()}")
        options.add_argument("--remote-debugging-port=9226")
```

После настройки браузера дочерним классам разрешается указать дополнительные параметры драйвера путем вызова метода `set_extra_driver_options()`. Затем устанавливается предел прокрутки и создается экземпляр драйвера Chrome с заданными опциями:

```
        self.set_extra_driver_options(options)

        self.scroll_limit = scroll_limit
        self.driver = webdriver.Chrome(
            options=options,
        )
```

Класс `BaseSeleniumCrawler` содержит методы-заглушки `set_extra_driver_options()` и `login()`, которые могут быть переопределены подклассами для реализации специфичного поведения. Это обеспечивает модульность, поскольку каждая платформа имеет особую страницу входа и HTML-структуру:

```
    def set_extra_driver_options(self, options: Options) -> None:
        pass

    def login(self) -> None:
        pass
```

Наконец, метод `scroll_page()` реализует механизм прокрутки страницы до заданного ограничения, что необходимо, например, при анализе ленты LinkedIn. Этот метод прокручивает страницу вниз, дожидается загрузки нового контента и повторяет процесс, пока не достигнет конца страницы или установленного предела:

```
    def scroll_page(self) -> None:
        """Прокручивать страницу LinkedIn в пределах заданного лимита прокрутки."""
        current_scroll = 0
        last_height = self.driver.execute_script("return
        ➥document.body.scrollHeight")
        while True:
            self.driver.execute_script("window.scrollTo(0,
            ➥ document.body.scrollHeight);")
            time.sleep(5)
            new_height = self.driver.execute_script("return
            ➥ document.body.scrollHeight")
            if new_height == last_height or (self.scroll_limit and
            ➥ current_scroll >= self.scroll_limit):
            break
            last_height = new_height
            current_scroll += 1
```

Теперь, когда мы познакомились с базовыми классами краулеров, рассмотрим реализации конкретных краулеров:

- `GitHubCrawler(BaseCrawler)`;
- `CustomArticleCrawler(BaseCrawler)`;
- `MediumCrawler(BaseSeleniumCrawler)`.



Реализацию этих краулеров вы можете найти в репозитории GitHub по адресу https://github.com/PacktPublishing/LLM-Engineers-Handbook/tree/main/llm_engineering/application/crawlers.

Класс GitHubCrawler

Класс `GitHubCrawler` предназначен для сбора данных из репозиториев GitHub и расширяет функциональность базового класса `BaseCrawler`. В нашем случае не требуется входить в систему через браузер, поскольку мы можем использовать предусмотренную в Git функцию клонирования репозиториев. Следовательно, нет необходимости задействовать Selenium. При инициализации краулер задает список шаблонов, позволяющих исключить из обработки стандартные файлы и каталоги, часто встречающиеся в репозиториях GitHub, такие как `.git`, `.toml`, `.lock`, `.png` и др. Это позволяет избежать включения ненужных данных в результирующий документ:

```
class GithubCrawler(BaseCrawler):
    model = RepositoryDocument

    def __init__(self, ignore=(".git", ".toml", ".lock", ".png")) -> None:
        super().__init__()
        self._ignore = ignore
```

Затем мы реализуем метод `extract()`. В его рамках краулер проверяет, был ли указанный репозиторий уже обработан и сохранен в БД. Если такой документ уже существует, выполнение метода прекращается во избежание дублирования:

```
   def extract(self, link: str, **kwargs) -> None:
       old_model = self.model.find(link=link)
       if old_model is not None:
           logger.info(f"Repository already exists in the database: {link}")

           return
```

Если же репозиторий новый, краулер извлекает из ссылки его имя и создает временный каталог для его клонирования. Это делается для того, чтобы все локальные данные могли быть автоматически удалены после завершения обработки:

```
        logger.info(f"Starting scrapping GitHub repository: {link}")

        repo_name = link.rstrip("/").split("/")[-1]

        local_temp = tempfile.mkdtemp()
```

Внутри блока `try` краулер изменяет текущий рабочий каталог на временный и выполняет команду `git clone` в отдельном процессе:

```
    try:
        os.chdir(local_temp)
        subprocess.run(["git", "clone", link])
```

После успешного клонирования формируется путь к локальной копии репозитория. Затем инициализируется пустой словарь для сборки содержимого файлов в стандартизированной форме. Краулер проходит по файловой структуре, пропуская те каталоги и файлы, имена которых соответствуют шаблонам исключения. В случае с релевантными файлами он считывает содержимое, удаляет символы пробела и помещает данные в словарь, где ключом служит путь к файлу:

```
repo_path = os.path.join(local_temp, os.listdir(local_temp)[0]) #
tree = {}
for root, _, files in os.walk(repo_path):
    dir = root.replace(repo_path, "").lstrip("/")
    if dir.startswith(self._ignore):
        continue

    for file in files:
        if file.endswith(self._ignore):
            continue
        file_path = os.path.join(dir, file)
        with open(os.path.join(root, file), "r", errors="ignore") as f:
            tree[file_path] = f.read().replace(" ", "")
```

После этого создается экземпляр модели `RepositoryDocument`, в который включается информация о содержимом репозитория, его имя, ссылка, название платформы и сведения об авторе. Полученный экземпляр сохраняется в MongoDB:

```
        user = kwargs["user"]
            instance = self.model(
            content=tree,
            name=repo_name,
            link=link,
            platform="github",
            author_id=user.id,
            author_full_name=user.full_name,
        )
        instance.save()
```

Независимо от того, завершился ли процесс извлечения данных успешно или возникло исключение, краулер гарантирует удаление временного каталога, тем самым освобождая использованные ресурсы:

```
        except Exception:
            raise
        finally:
            shutil.rmtree(local_temp)

    logger.info(f"Finished scrapping GitHub repository: {link}")
```

Класс CustomArticleCrawler

Класс `CustomArticleCrawler` реализует несколько иной подход к сбору данных из Интернета. Он использует класс `AsyncHtmlLoader` для загрузки всего HTML-кода по предоставленной ссылке и `Html2TextTransformer` для извлечения текста из этого кода. Оба класса предоставляются импортируемым пакетом `langchain_community`:

```
from urllib.parse import urlparse

from langchain_community.document_loaders import AsyncHtmlLoader
from langchain_community.document_transformers.html2text
➥import Html2TextTransformer
from loguru import logger

from llm_engineering.domain.documents import ArticleDocument

from .base import BaseCrawler
```

Затем мы определяем класс `CustomArticleCrawler`, который наследует от `BaseCrawler`. В данном случае нам тоже нет необходимости входить в систему или прокручивать страницу, поэтому мы не задействуем Selenium. В рамках метода `extract()` краулер первым делом проверяет наличие обрабатываемой статьи в БД, чтобы избежать дублирования контента:

```
class CustomArticleCrawler(BaseCrawler):
    model = ArticleDocument

    def extract(self, link: str, **kwargs) -> None:
        old_model = self.model.find(link=link)
        if old_model is not None:
            logger.info(f"Article already exists in the database: {link}")

            return
```

Если статья отсутствует в БД, начинается процесс ее извлечения. Для этого используется класс `AsyncHtmlLoader`, который загружает HTML-код по предоставленной ссылке. Затем код преобразуется в обычный текст с помощью класса

`Html2TextTransformer`, возвращающего список документов. Нас интересует только первый из них. Поскольку вся логика извлечения и преобразования контента делегируется этим двум классам, мы не контролируем процесс обработки данных. Поэтому мы применяем `CustomArticleCrawler` в качестве резервного варианта для доменов, где у нас не реализован специализированный краулер. Оба упомянутых класса следуют подходу LangChain, суть которого сводится к предоставлению высокоуровневой функциональности, достаточно хорошо работающей в большинстве типовых сценариев. Такой подход позволяет быстро создать рабочее решение, но затрудняет его адаптацию. Это одна из причин, по которой многие специалисты избегают использования LangChain.

```
logger.info(f"Starting scrapping article: {link}")

loader = AsyncHtmlLoader([link])
docs = loader.load()

html2text = Html2TextTransformer()
docs_transformed = html2text.transform_documents(docs)
doc_transformed = docs_transformed[0]
```

Из полученного документа извлекается основной текст статьи и такие метаданные страницы, как заголовок, подзаголовок и язык статьи:

```
content = {
    "Title": doc_transformed.metadata.get("title"),
    "Subtitle": doc_transformed.metadata.get("description"),
    "Content": doc_transformed.page_content,
    "language": doc_transformed.metadata.get("language"),
}
```

Далее мы разбираем URL-адрес для определения платформы (домена), с которой была получена статья:

```
parsed_url = urlparse(link)
platform = parsed_url.netloc
```

Затем создается новый экземпляр модели, заполняемый извлеченным контентом. Наконец, мы помещаем этот экземпляр в хранилище данных MongoDB:

```
user = kwargs["user"]
instance = self.model(
    content=content,
    link=link,
    platform=platform,
    author_id=user.id,
    author_full_name=user.full_name,
)
instance.save()

logger.info(f"Finished scrapping custom article: {link}")
```

Итак, мы выяснили, как организован сбор данных из репозиториев GitHub и других сайтов. Осталось обсудить краулер, использующий Selenium для программного управления браузером. Поэтому в следующем разделе мы рассмотрим реализацию `MediumCrawler`.

Класс MediumCrawler

Реализация начинается с подключения необходимых библиотек и определения класса `MediumCrawler`, который наследует от `BaseSeleniumCrawler`:

```
from bs4 import BeautifulSoup
from loguru import logger

from llm_engineering.domain.documents import ArticleDocument

from .base import BaseSeleniumCrawler

class MediumCrawler(BaseSeleniumCrawler):
    model = ArticleDocument
```

Внутри класса `MediumCrawler` мы используем метод `set_extra_driver_options()`, позволяющий дополнить стандартные параметры драйвера, которые задействует Selenium:

```
    def set_extra_driver_options(self, options) -> None:
        options.add_argument(r"--profile-directory=Profile 2")
```

Метод `extract()` реализует основную функциональность. Сначала он проверяет, существует ли статья в БД, чтобы избежать дублирования.

Если статья новая, метод переходит по указанной ссылке и выполняет прокрутку страницы, чтобы обеспечить полную загрузку содержимого:

```
    def extract(self, link: str, **kwargs) -> None:
        old_model = self.model.find(link=link)
        if old_model is not None:
            logger.info(f"Article already exists in the database: {link}")

            return

        logger.info(f"Starting scrapping Medium article: {link}")

        self.driver.get(link)
        self.scroll_page()
```

После полной загрузки страницы метод использует библиотеку `BeautifulSoup` для разбора HTML-кода и извлечения нужных элементов: заголовка, подзаголовка и основного текста статьи. `BeautifulSoup` — это популярная библиотека Python, предназначенная для парсинга документов HTML и XML, и именно

ее мы использовали для извлечения необходимых элементов из HTML-кода, полученного с помощью Selenium. Далее все данные агрегируются в словарь:

```
soup = BeautifulSoup(self.driver.page_source, "html.parser")
title = soup.find_all("h1", class_="pw-post-title")
subtitle = soup.find_all("h2", class_="pw-subtitle-paragraph")
data = {
    "Title": title[0].string if title else None,
    "Subtitle": subtitle[0].string if subtitle else None,
    "Content": soup.get_text(),
}
```

Наконец, метод завершает сеанс работы с WebDriver, освобождая ресурсы. Затем создается экземпляр `ArticleDocument`, который заполняется извлеченным содержимым и данными о пользователе (переданными через `kwargs`), после чего сохраняется в БД:

```
self.driver.close()

user = kwargs["user"]
instance = self.model(
    platform="medium",
    content=data,
    link=link,
    author_id=user.id,
    author_full_name=user.full_name,
)
instance.save()

logger.info(f"Successfully scraped and saved article: {link}")
```

На этом реализация `MediumCrawler` завершена. Краулер для LinkedIn построен по аналогичной схеме: он задействует инструмент Selenium для входа в систему и доступа к ленте с последними публикациями пользователя, затем извлекает посты и последовательно прокручивает ленту, подгружая новый контент, вплоть до достижения заданного предела. С полной реализацией краулера для LinkedIn можно ознакомиться в репозитории по ссылке https://github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/llm_engineering/application/crawlers/linkedin.py.

Из-за роста популярности LLM сбор данных из Интернета стал ключевым этапом работы многих реальных ИИ-приложений. В связи с этим в экосистеме Python появились такие высокоуровневые инструменты, как Scrapy (https://github.com/scrapy/scrapy) — универсальный фреймворк извлечения структурированных данных с веб-сайтов, и Crawl4AI (https://github.com/unclecode/crawl4ai) — специализированный инструмент для сбора данных именно для LLM и ИИ-приложений.

В этом разделе мы рассмотрели реализацию трех типов краулеров: один из них использует утилиту Git в отдельном процессе для клонирования репозиториев,

второй применяет утилиты LangChain для извлечения HTML-кода с отдельной веб-страницы, а третий задействует инструмент Selenium и предназначен для более сложных сценариев, требующих прохождения аутентификации, прокрутки страницы и преобразования HTML-кода в текст. Далее обсудим принцип работы используемых в этой главе классов документов, таких как `ArticleDocument`.

Документы в хранилище данных NoSQL

Для структурирования категорий данных нам пришлось реализовать три класса документов. Они определяют конкретные атрибуты каждого типа документа, такие как содержимое, имя автора и ссылка на источник. Хорошей практикой считается структурировать данные в виде классов, а не словарей, поскольку это позволяет явно задать ожидаемые атрибуты и избежать многих ошибок во время выполнения. Например, при обращении к значению в словаре Python мы не можем быть уверены, что оно существует или соответствует нужному типу. Обернув элементы данных в классы, мы гарантируем наличие всех необходимых атрибутов и их соответствие ожидаемым типам.

Благодаря библиотекам вроде Pydantic мы получаем доступ к встроенным функциям проверки типов, обеспечивающим согласованность наших наборов данных. Таким образом, наши категории данных были смоделированы в виде следующих классов документов, которые уже использовались в приведенном ранее коде:

- `ArticleDocument`;
- `PostDocument`;
- `RepositoryDocument`.

Однако речь идет не просто о классах данных Python или моделях Pydantic. Эти классы поддерживают чтение данных из хранилища MongoDB и их запись в него. Чтобы внедрить такую функциональность сразу во все классы документов и избежать дублирования кода, мы использовали паттерн проектирования *ODM* (Object-Document Mapping — «объектно-документное отображение»), основанный на паттерне *ORM* (Object-Relational Mapping — «объектно-реляционное отображение»). Прежде чем перейти к рассмотрению нашего собственного ODM-решения и структуры классов документов, давайте кратко познакомимся с принципами ORM и ODM.

Паттерны проектирования ORM и ODM

Сперва разберемся в самой сути ORM. Эта технология позволяет обращаться к данным, хранящимся в БД, и изменять их с помощью объектно-ориентированного подхода. Вместо использования SQL-запросов или специфичных API мы инкапсулируем всю сложность в ORM-классе, который умеет выполнять

типовые операции с БД — прежде всего операции CRUD (create, read, update, delete — «создание, чтение, обновление, удаление»). Таким образом, технология ORM избавляет нас от необходимости вручную делать запросы к БД и сокращает объем шаблонного кода. ORM применяется при работе с базами данных SQL, такими как PostgreSQL или MySQL.

ORM используется при взаимодействии с БД в большинстве современных Python-приложений. Хотя язык SQL по-прежнему популярен в сфере обработки данных, к сырым SQL-запросам редко прибегают в бэкендах Python-систем. Наиболее распространенной ORM-библиотекой Python является SQLAlchemy (https://www.sqlalchemy.org/). Кроме того, благодаря растущей популярности FastAPI все чаще используется SQLModel (https://github.com/fastapi/sqlmodel) — обертка над SQLAlchemy, упрощающая интеграцию с этим фреймворком.

Например, с помощью SQLAlchemy можно определить ORM-класс `User` с полями `ID` и `name`. Данный класс будет сопоставлен с таблицей `users` в БД SQL. При создании нового пользователя и его сохранении в базе соответствующая запись автоматически добавляется в таблицу `users`. Аналогичным образом реализуются и другие CRUD-операции.

```
from sqlalchemy import Column, Integer, String, create_engine
from sqlalchemy.orm import declarative_base, sessionmaker

    Base = declarative_base()

# Определение класса, который сопоставляется с таблицей users
class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    name = Column(String)
```

Используя ORM-класс `User`, мы можем легко добавлять и извлекать данные пользователей непосредственно из Python-кода, обходясь без написания SQL-запросов. Как правило, ORM поддерживает все операции CRUD. Ниже приведен фрагмент кода, показывающий, как сохранить экземпляр класса `User` в базе SQLite:

```
    engine = create_engine("sqlite:///:memory:")
    Base.metadata.create_all(engine)

    # Создание сеанса взаимодействия с базой данных
    Session = sessionmaker(bind=engine)
    session = Session()

    # Добавление нового пользователя
    new_user = User(name="Alice")
    session.add(new_user)
    session.commit()
```

А вот как можно извлечь данные пользователя из таблицы `users`:

```
user = session.query(User).first()
if user:
    print(f"User ID: {user.id}")
    print(f"User name: {user.name}")
```



Весь сценарий и инструкции по его запуску можно найти в репозитории GitHub, в файле code_snippets/03_orm.py.

Паттерн ODM очень похож на ORM, но взаимодействует не с SQL-базами и таблицами, а с базами данных NoSQL (например, MongoDB) и неструктурированными коллекциями. Поскольку мы работаем с NoSQL, в нашем случае структура данных опирается на коллекции, в которых хранятся JSON-подобные документы, а не строки таблиц.

Таким образом, паттерн ODM упрощает работу с документо-ориентированными базами данных NoSQL и обеспечивает отображение объектно-ориентированного кода на JSON-подобные документы. Чтобы лучше понять принцип работы этого паттерна, мы реализуем облегченный вариант ODM-модуля поверх MongoDB.

Реализация ODM-класса

Здесь мы покажем, как реализовать ODM-класс с нуля. Это отличное упражнение, которое позволит разобраться в принципах работы ODM и попрактиковаться в написании модульного и повторно используемого кода на языке Python. Мы создадим базовый ODM-класс под названием `NoSQLBaseDocument`, от которого будут наследовать все остальные классы документов для взаимодействия с хранилищем данных MongoDB.



Исходный код этого класса находится в файле репозитория: llm_engineering/domain/base/nosql.py.

Реализация ODM-класса начинается с импорта необходимых модулей и установки соединения с БД. Через переменную `_database` создается подключение к БД, имя которой задается в конфигурационных настройках проекта (по умолчанию — `twin`):

```
import uuid
from abc import ABC
from typing import Generic, Type, TypeVar

from loguru import logger
from pydantic import UUID4, BaseModel, Field
from pymongo import errors
```

```
from llm_engineering.domain.exceptions import ImproperlyConfigured
from llm_engineering.infrastructure.db.mongo import connection
from llm_engineering.settings import settings

_database = connection.get_database(settings.DATABASE_NAME)
```

Далее мы определяем типовую переменную `T`, связанную с классом `NoSQLBaseDocument`. Она использует механизм Python, позволяющий обобщать типы внутри класса. Например, если мы реализуем класс `ArticleDocument`, наследующий от `NoSQLBaseDocument`, то при анализе сигнатур функций переменная `T` везде будет трактоваться как `ArticleDocument` (подробнее об обобщениях Python можно узнать здесь: https://realpython.com/python312-typing). Затем объявляется класс `NoSQLBaseDocument`. Это абстрактный базовый класс, который наследуется одновременно от трех классов: `BaseModel` из библиотеки Pydantic, `Generic` из Python (который, как мы уже видели, отвечает за обобщенное программирование) и `ABC`, делающего класс абстрактным. Этот класс служит основой для построения ODM-класса.

```
T = TypeVar("T", bound="NoSQLBaseDocument")

class NoSQLBaseDocument(BaseModel, Generic[T], ABC):
```

Внутри класса `NoSQLBaseDocument` определяется поле `id` — объект UUID версии 4, значение которого автоматически генерируется с помощью функции `default_factory`. Реализованы также методы `__eq__` и `__hash__`, позволяющие сравнивать экземпляры и использовать их в хешируемых коллекциях, таких как множества или словари, на основе уникального атрибута `id`:

```
    id: UUID4 = Field(default_factory=uuid.uuid4)

    def __eq__(self, value: object) -> bool:
        if not isinstance(value, self.__class__):
            return False

        return self.id == value.id

    def __hash__(self) -> int:
        return hash(self.id)
```

Кроме того, класс содержит методы для выполнения преобразований между документами MongoDB и экземплярами класса. Метод класса `from_mongo()` превращает словарь, полученный из MongoDB, в экземпляр класса, а метод `to_mongo()` конвертирует экземпляр модели в словарь, пригодный для сохранения в базе MongoDB:

```
    @classmethod
    def from_mongo(cls: Type[T], data: dict) -> T:
        if not data:
            raise ValueError("Data is empty.")
```

```
        id = data.pop("_id")

        return cls(**dict(data, id=id))

    def to_mongo(self: T, **kwargs) -> dict:
        exclude_unset = kwargs.pop("exclude_unset", False)
        by_alias = kwargs.pop("by_alias", True)

        parsed = self.model_dump(exclude_unset=exclude_unset,
        ➥ by_alias=by_alias, **kwargs)

        if "_id" not in parsed and "id" in parsed:
            parsed["_id"] = str(parsed.pop("id"))

        for key, value in parsed.items():
            if isinstance(value, uuid.UUID):
                parsed[key] = str(value)

        return parsed
```

Метод `save()` позволяет сохранить экземпляр модели в соответствующей коллекции MongoDB. Он получает эту коллекцию, преобразует экземпляр в формат MongoDB (с помощью вышеупомянутого метода `to_mongo()`) и пытается поместить его в базу данных, при необходимости обрабатывая возникающие ошибки записи:

```
    def save(self: T, **kwargs) -> T | None:
        collection = _database[self.get_collection_name()]
        try:
            collection.insert_one(self.to_mongo(**kwargs))

            return self
        except errors.WriteError:
            logger.exception("Failed to insert document.")

            return None
```

Метод класса `get_or_create()` ищет документ в БД по заданным фильтрам. Если такой документ найден, то он преобразуется в экземпляр модели. А если нет, то создается новый экземпляр с указанными параметрами фильтра и сохраняется в базе:

```
    @classmethod
    def get_or_create(cls: Type[T], **filter_options) -> T:
        collection = _database[cls.get_collection_name()]
        try:
            instance = collection.find_one(filter_options)
            if instance:
                return cls.from_mongo(instance)

            new_instance = cls(**filter_options)
            new_instance = new_instance.save()
```

```
            return new_instance
        except errors.OperationFailure:
            logger.exception(f"Failed to retrieve document with filter
            ➥ options: {filter_options}")

            raise
```

Метод `bulk_insert()` позволяет помещать в БД сразу несколько документов:

```
@classmethod
def bulk_insert(cls: Type[T], documents: list[T], **kwargs) -> bool:
    collection = _database[cls.get_collection_name()]
    try:
        collection.insert_many([doc.to_mongo(**kwargs) for
        ➥ doc in documents])

        return True
    except (errors.WriteError, errors.BulkWriteError):
        logger.error(f"Failed to insert documents of type {cls.__name__}")

        return False
```

Метод `find()` ищет в базе один документ, соответствующий указанным параметрам фильтра:

```
@classmethod
def find(cls: Type[T], **filter_options) -> T | None:
    collection = _database[cls.get_collection_name()]
    try:
        instance = collection.find_one(filter_options)
        if instance:
            return cls.from_mongo(instance)

        return None
    except errors.OperationFailure:
        logger.error("Failed to retrieve document.")

        return None
```

Метод `bulk_find()` возвращает список документов, соответствующих параметрам фильтра, преобразуя каждый из них в экземпляр модели.

```
@classmethod
def bulk_find(cls: Type[T], **filter_options) -> list[T]:
    collection = _database[cls.get_collection_name()]
    try:
        instances = collection.find(filter_options)
        return [document for instance in instances if (document :=
        ➥ cls.from_mongo (instance)) is not None]
    except errors.OperationFailure:
        logger.error("Failed to retrieve document.")

        return []
```

Наконец, метод `get_collection_name()` возвращает имя MongoDB-коллекции, связанной с классом. Для этого в каждом классе должна присутствовать вложенная структура `Settings` с атрибутом `name`, задающим имя коллекции. Если такая структура отсутствует, выбрасывается исключение `ImproperlyConfigured`, которое указывает на необходимость определить вложенный класс `Settings`:

```
@classmethod
def get_collection_name(cls: Type[T]) -> str:
    if not hasattr(cls, "Settings") or not hasattr(cls.Settings, "name"):
        raise ImproperlyConfigured(
            "Document should define an Settings configuration class with
            ➥ the name of the collection."
        )

    return cls.Settings.name
```

Каждый подкласс можно конфигурировать с помощью вложенного класса `Settings`, где задается имя коллекции и другие специфичные для этого подкласса параметры. В экосистеме Python существует готовая ODM-реализация для MongoDB — библиотека `mongoengine` (доступная на GitHub), которая следует аналогичному паттерну, но является более избыточной. В нашем проекте мы реализовали собственный легковесный вариант, чтобы отточить навыки написания модульного и обобщенного кода на основе принципов ООП, что крайне важно для создания ПО, готового к развертыванию в продакшене.

Категории данных и пользовательские классы документов

Последним этапом является реализация подклассов, наследующих от базового класса `NoSQLBaseDocument`. Именно эти конкретные классы определяют типы данных. Мы использовали их на протяжении всей главы при работе со статьями, репозиториями и постами с помощью краулеров.

Мы начинаем с импорта необходимых модулей Python и базового ODM-класса:

```
from abc import ABC
from typing import Optional

from pydantic import UUID4, Field

from .base import NoSQLBaseDocument
from .types import DataCategory
```

Затем определяется класс `enum` (перечисление), в котором централизованно задаются все используемые в проекте типы данных. Эти переменные выступают в роли констант при конфигурировании всех ODM-классов в книге.



Код класса enum находится в файле репозитория: llm_engineering/domain/types.py.

```
from enum import StrEnum

class DataCategory(StrEnum):
    PROMPT = "prompt"
    QUERIES = "queries"

    INSTRUCT_DATASET_SAMPLES = "instruct_dataset_samples"
    INSTRUCT_DATASET = "instruct_dataset"
    PREFERENCE_DATASET_SAMPLES = "preference_dataset_samples"
    PREFERENCE_DATASET = "preference_dataset"

    POSTS = "posts"
    ARTICLES = "articles"
    REPOSITORIES = "repositories"
```

Класс `Document` вводится как абстрактная базовая модель для других документов, реализуемая поверх ODM-класса `NoSQLBaseDocument`. Он определяет такие общие атрибуты, как содержимое, название платформы и сведения об авторе, и обеспечивает унифицированную структуру для всех наследующих от него документов:

```
class Document(NoSQLBaseDocument, ABC):
    content: dict
    platform: str
    author_id: UUID4 = Field(alias="author_id")
    author_full_name: str = Field(alias="author_full_name")
```

Наконец, создаются конкретные типы документов путем расширения класса `Document`. Классы `RepositoryDocument`, `PostDocument` и `ArticleDocument` представляют разные категории данных, каждая из которых предусматривает специфичные поля и настройки, определяющие соответствующее имя коллекции в базе данных:

```
class RepositoryDocument(Document):
    name: str
    link: str

    class Settings:
        name = DataCategory.REPOSITORIES

class PostDocument(Document):
    image: Optional[str] = None
    link: str | None = None

    class Settings:
        name = DataCategory.POSTS

class ArticleDocument(Document):
    link: str

    class Settings:
        name = DataCategory.ARTICLES
```

Наконец, мы реализуем класс `UserDocument`, предназначенный для хранения и получения имен всех пользователей проекта LLM-двойника:

```
class UserDocument(NoSQLBaseDocument):
    first_name: str
    last_name: str

    class Settings:
        name = "users"

    @property
    def full_name(self):
        return f"{self.first_name} {self.last_name}"
```

Благодаря реализации базового ODM-класса `NoSQLBaseDocument` мы смогли сосредоточиться исключительно на полях и специфичном поведении каждой сущности, делегировав всю CRUD-функциональность родительскому классу. Кроме того, благодаря использованию библиотеки Pydantic для определения полей мы получили встроенную проверку типов. Например, если при создании экземпляра класса `ArticleDocument` значение поля `link` будет `None` или отличным от строки, возникнет исключение, говорящее о невалидных данных.

На этом реализация пайплайна сбора данных завершена. Мы начали с компонентов ZenML, затем перешли к краулерам и закончили реализацией ODM-класса и документов, описывающих типы данных. Теперь остается запустить этот пайплайн и загрузить исходные данные в хранилище MongoDB.

Сбор исходных данных в хранилище

Оркестрация пайплайна сбора данных осуществляется с помощью ZenML. Это означает, что запуск пайплайна может выполняться вручную, по расписанию или при наступлении определенных событий. В текущем разделе мы покажем, как запустить его вручную. Остальные сценарии будут рассмотрены в главе 11, где речь пойдет о MLOps.

Для каждого автора мы настроили отдельный способ запуска пайплайна. В репозитории проекта предусмотрены конфигурационные файлы ZenML для сбора данных Пола Юстина и Максима Лабонна. Например, для сбора данных Максима необходимо выполнить следующую команду:

```
poetry poe run-digital-data-etl-maxime
```

Эта команда запустит ZenML-пайплайн с использованием следующего YAML-файла конфигурации:

```
parameters:
    user_full_name: Maxime Labonne # [Имя (имена)] [Фамилия]
```

```
links:
    # Личный блог
    - https://mlabonne.github.io/blog/posts/2024-07-29_Finetune_
    ➥Llama31.html
    - https://mlabonne.github.io/blog/posts/2024-07-15_The_Rise_
    ➥of_Agentic_Data_Generation.html
    # Substack
    - https://maximelabonne.substack.com/p/uncensor-any-llm-
    ➥withabliteration-d30148b7d43e
    - https://maximelabonne.substack.com/p/create-mixtures-of-
    ➥expertswith-mergekit-11b318c99562
    - https://maximelabonne.substack.com/p/merge-large-language-
    ➥modelswith-mergekit-2118fb392b54
    ... # Дополнительные ссылки Substack
```

На рис. 3.3 была показана структура выполнения (в виде ориентированного ациклического графа) и информация о запуске пайплайна на дашборде ZenML. На рис. 3.5 представлен выходной артефакт `user`, сгенерированный пайплайном сбора данных. Здесь можно просмотреть значение параметра `user_full_name`, использованного при запросе, а также извлеченные из базы MongoDB данные о пользователе, контент которого собирался в рамках этого запуска.

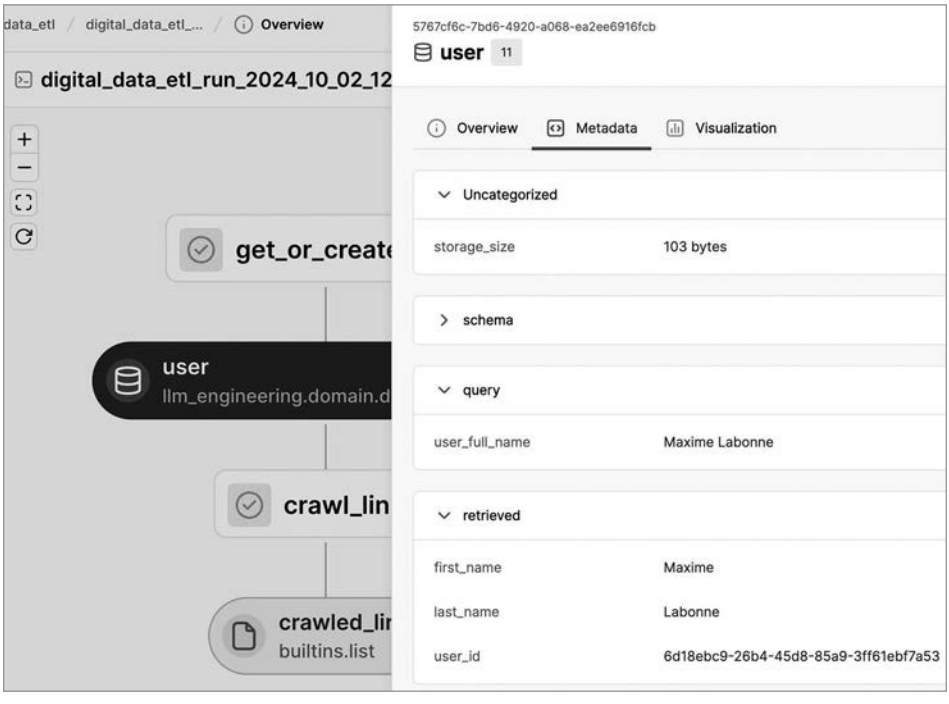


Рис. 3.5. Пример выходного артефакта user, полученного в результате запуска пайплайна сбора данных с использованием файла конфигурации Максима

На рис. 3.6 показан выходной артефакт `crawled_links`, содержащий список доменов, с которых были собраны данные, общее число обработанных ссылок на каждом домене и количество успешно полученных материалов.

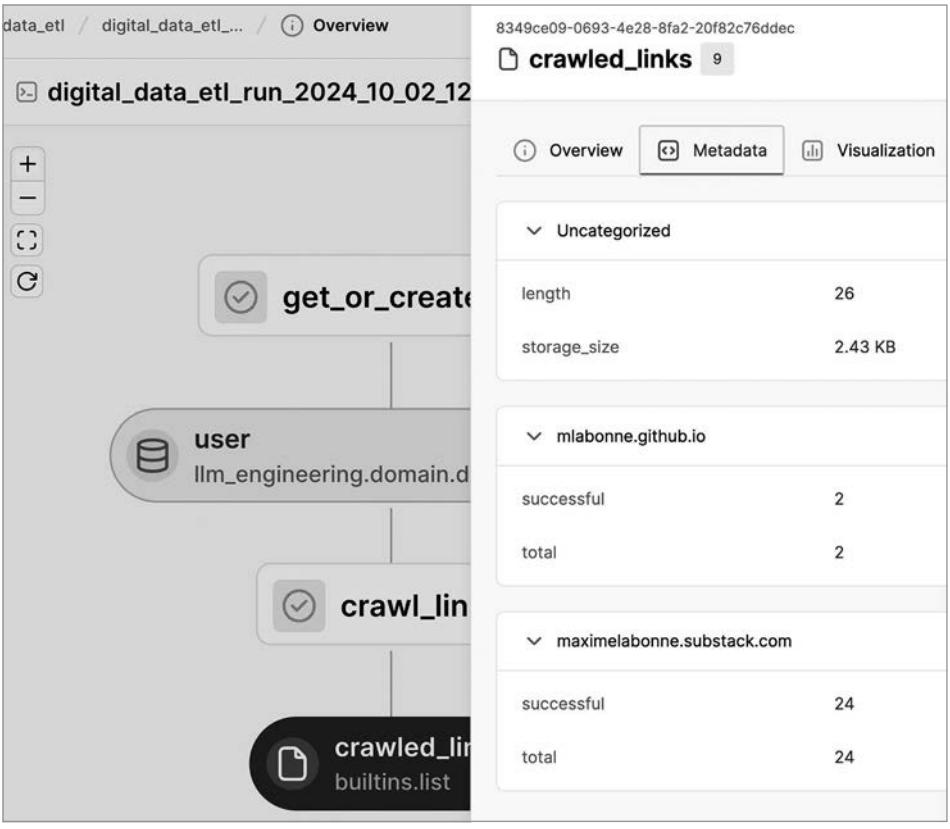


Рис. 3.6. Пример выходного артефакта crawled_links, полученного в результате запуска пайплайна сбора данных с использованием файла конфигурации Максима

Эти артефакты позволяют отслеживать результаты и метаданные каждого отдельного запуска пайплайна, что значительно упрощает отладку и мониторинг.

Мы можем загрузить артефакт `crawled_links` из любого места в коде, используя его уникальный идентификатор, который можно найти в ZenML:

```
from zenml.client import Client

artifact = Client().get_artifact_version('8349ce09-0693-4e28-8fa2-
➥20f82c76ddec')
loaded_artifact = artifact.load()
```

Аналогичным образом можно запустить сбор данных Пола, использовав его конфигурационный YAML-файл:

```
parameters:
    user_full_name: Paul Iusztin # [Имя (имена)] [Фамилия]
    links:
        # Medium
        - https://medium.com/decodingml/an-end-to-end-framework-
        ➥forproduction-ready-llm-systems-by-building-your-llm-twin-2cc6bb01141f
        - https://medium.com/decodingml/a-real-time-retrieval-system-for-
        ➥ragon-social-media-data-9cc01d50a2a0
        - https://medium.com/decodingml/sota-python-streaming-pipelines-
        ➥forfine-tuning-llms-and-rag-in-real-time-82eb07795b87
        ... # Дополнительные ссылки Medium
        # Substack
        - https://decodingml.substack.com/p/real-time-feature-
        ➥pipelineswith?r=1ttoeh
        - https://decodingml.substack.com/p/building-ml-systems-the-
        ➥rightway?r=1ttoeh
        - https://decodingml.substack.com/p/reduce-your-pytorchs-
        ➥codelatency?r=1ttoeh
        ... # Дополнительные ссылки Substack
```

Для запуска используется команда poe:

```
poetry poe run-digital-data-etl-paul
```

Она вызывает следующую команду, ссылающуюся на конфигурационный файл Пола:

```
poetry run python -m tools.run --run-etl --no-cache --etl-config-filename
➥ digital_data_etl_paul_iusztin.yaml
```

Все конфигурационные файлы находятся в папке репозитория `configs/`. Кроме того, мы настроили команду, запускающую сбор данных сразу для всех поддерживаемых авторов:

```
poetry poe run-digital-data-etl
```

С помощью наших ODM-классов можно легко выполнять запросы к хранилищу MongoDB. Например, вот как извлечь все собранные статьи Пола Юстина:

```
from llm_engineering.domain.documents import ArticleDocument, UserDocument

user = UserDocument.get_or_create(first_name="Paul", last_name="Iusztin")
articles = ArticleDocument.bulk_find(author_id=str(user.id))

print(f"User ID: {user.id}")
print(f"User name: {user.first_name} {user.last_name}")
print(f"Number of articles: {len(articles)}")
print("First article link:", articles[0].link)
```

Результат выполнения этого запроса выглядит так:

```
User ID: 900fec95-d621-4315-84c6-52e5229e0b96
User name: Paul Iusztin
Number of articles: 50
First article link: https://medium.com/decodingml/an-end-to-end-frameworkfor
➥-production-ready-llm-systems-by-building-your-llm-twin-2cc6bb01141f
```

Благодаря всего двум строкам кода у нас появилась возможность фильтровать содержимое базы MongoDB, используя любые ODM-классы, определенные в нашем проекте.

Если вы хотите провести дополнительную проверку корректности работы пайплайна, воспользуйтесь *плагином MongoDB* для вашей IDE. Можно также задействовать сторонние инструменты визуализации данных NoSQL. Для подключения к локальной базе MongoDB используйте следующий URI: `mongodb://llm_engineering:llm_engineering@127.0.0.1:27017`. При работе с облачным кластером MongoDB идентификатор URI будет другим (см. главу 11).

Итак, теперь вы знаете, как запускать пайплайн сбора данных с помощью разных конфигураций ZenML, визуализировать выходные артефакты и извлекать из хранилища данные конкретного автора, относящиеся к нужной категории. На этом часть, посвященная инженерии данных, завершается, и мы переходим к заключению.

Устранение неполадок

Исходные данные, хранящиеся в базе MongoDB, будут играть ключевую роль на всех последующих этапах. Поэтому, если вы не смогли успешно выполнить код из этой главы из-за сбоев в работе краулеров, ниже вы найдете решения, которые позволят вам устранить ошибки и двигаться дальше.

Проблемы с Selenium

При работе с Selenium могут возникать проблемы, связанные с драйвером браузера, например с ChromeDriver. Если краулеры, использующие Selenium (например, `MediumCrawler`), не работают из-за ошибок ChromeDriver, вы можете обойти их, закомментировав ссылки на платформу Medium в YAML-файлах конфигурации пайплайна сбора данных. Для этого перейдите в папку `configs/` и найдите все YAML-файлы, имена которых начинаются с `digital_data_etl_*`, например: `digital_data_etl_maxime_labonne.yaml`. Откройте эти файлы и закомментируйте все URL-адреса, относящиеся к Medium, как показано на рис. 3.7. Ссылки на Substack и персональные блоги можно оставить, так как для их обработки используется `CustomArticleCrawler`, который не зависит от Selenium.

LLM-Engineers-Handbook / configs / **digital_data_etl_paul_iusztin.yaml**

iusztinpaul docs: Comment out Medium urls

Code | Blame | 62 lines (61 loc) · 4.62 KB

```
settings:
  docker:
    parent_image: 992382797823.dkr.ecr.eu-central-1.amazonaws.com/zenml-rlwlcs:latest
    skip_build: True
  orchestrator.sagemaker:
    synchronous: false

parameters:
  user_full_name: Paul Iusztin # [First Name(s)] [Last Name]
  links:
    # Medium (only articles that are not under the paid wall work)
    # - https://medium.com/decodingml/an-end-to-end-framework-for-production-ready-llm-systems-by-building-your-llm-twin-2cc6bb01141f
    # - https://medium.com/decodingml/a-real-time-retrieval-system-for-rag-on-social-media-data-9cc01d50a2a0
    # - https://medium.com/decodingml/sota-python-streaming-pipelines-for-fine-tuning-llms-and-rag-in-real-time-82eb07795b87
    # - https://medium.com/decodingml/the-4-advanced-rag-algorithms-you-must-know-to-implement-5d0c7f1199d2
    # - https://medium.com/decodingml/architect-scalable-and-cost-effective-llm-rag-inference-pipelines-73b94ef82a99
    # Substack
    - https://decodingml.substack.com/p/real-time-feature-pipelines-with?r=1ttoeh
    - https://decodingml.substack.com/p/building-ml-systems-the-right-way?r=1ttoeh
    - https://decodingml.substack.com/p/reduce-your-pytorchs-code-latency?r=1ttoeh
    - https://decodingml.substack.com/p/llm-agents-demystified?r=1ttoeh
```

Рис. 3.7. Устранение проблем, связанных с использованием Selenium при сборе исходных данных

Импорт резервной копии данных

Если ничего не помогает, вы можете наполнить хранилище MongoDB данными, содержащимися в каталоге `data/data_warehouse_raw_data`. Это позволит вам приступить к тонкой настройке модели и инференсу, не запуская пайплайн сбора данных. Чтобы импортировать все данные из указанного каталога, выполните следующую команду:

```
poetry poe run-import-data-warehouse-from-json
```

После ее выполнения ваша БД будет содержать точную копию набора данных, который использовался при разработке приведенного в книге кода. Если в результате в вашем хранилище MongoDB оказалось 88 статей и три пользователя, значит, операция импорта прошла успешно.

Резюме

В этой главе мы рассмотрели процесс проектирования и построения пайплайна сбора данных для реализации LLM-двойника. Вместо использования статического набора данных мы создали собственную выборку, чтобы смоделировать реальные условия и подготовиться к решению практических задач.

Сначала мы изучили архитектуру пайплайна сбора данных для LLM-двойника, который представляет собой классический ETL-процесс, а затем проанализировали его реализацию. Обсудили его оркестрацию с помощью ZenML, после чего подробно рассмотрели работу краулеров. Вы научились собирать данные тремя способами: путем клонирования репозитория с помощью консольных команд и веб-скрапинга с помощью библиотек LangChain, а также с использованием инструмента Selenium, позволяющего программно управлять браузером. Наконец, создали собственный ODM-класс, на основе которого построили иерархию классов документов для представления таких сущностей, как статьи, посты и репозитории.

В конце главы вы научились запускать ZenML-пайплайны с использованием разных YAML-файлов конфигурации и просматривать результаты их выполнения на дашборде, а также взаимодействовать с хранилищем MongoDB с помощью ODM-классов.

В следующей главе обсудим основные этапы работы RAG-пайплайна признаков, включая разбивку документов на фрагменты, создание эмбеддингов, их загрузку в векторную БД и оптимизацию для повышения эффективности извлечения. Мы также создадим всю необходимую инфраструктуру с использованием инструмента Pulumi и развернем RAG-пайплайн загрузки данных на платформе AWS.

Список используемых источников

- *Breuss M.* Beautiful Soup: Build a Web Scraper With Python. Real Python. https://realpython.com/beautiful-soup-web-scraper-python/.
- *David D.* Guide to Web Scraping with Selenium in 2025. Bright Data. https://brightdata.com/blog/how-tos/using-selenium-for-web-scraping.
- *Hjelle G. A.* Python 3.12 Preview: Static Typing Improvements. Real Python. https://realpython.com/python312-typing/.
- ORM Quick Start — SQLAlchemy 2.0 documentation. https://docs.sqlalchemy.org/en/20/orm/quickstart.html.
- *Ramos L. P.* Python and MongoDB: Connecting to NoSQL Databases. Real Python. https://realpython.com/introduction-to-mongodb-and-python/.
- Refactoring.Guru. Builder. https://refactoring.guru/design-patterns/builder.

4 RAG-пайплайн признаков

Генерация, дополненная поиском (RAG), является важнейшим элементом большинства приложений на базе генеративного ИИ. Основная задача RAG заключается во внедрении пользовательских данных в *большую языковую модель* для выполнения над ними, например, операций обобщения, переформулирования или извлечения информации. Обычно LLM применяется для работы с данными, на которых она не обучалась (то есть с личными или вновь появившимися). Поскольку тонкая настройка LLM — это крайне затратный процесс, RAG представляет собой эффективную стратегию, позволяющую обращаться к новой информации без тонкой настройки модели.

Мы начнем с теоретического раздела, посвященного основам RAG и принципам работы этого механизма. Затем подробно разберем все компоненты стандартной RAG-системы: разбиение информации на фрагменты, создание эмбеддингов и работу с векторными базами данных. После этого мы представим ряд оптимизаций, применяемых в продвинутых RAG-системах. Далее перейдем к изучению архитектуры RAG-пайплайна признаков нашего LLM-двойника, применив полученные теоретические знания. Наконец, рассмотрим практический пример, реализовав RAG-пайплайн признаков для LLM-двойника на основе описанных в книге архитектурных решений.

К концу главы вы получите полное и четкое представление о механизме RAG и о том, как он применяется в рамках нашего проекта.

Принципы работы механизма RAG

Механизм RAG повышает точность и надежность работы генеративных моделей ИИ за счет использования информации из внешних источников, дополняющей внутренние знания LLM. Прежде чем углубляться в детали, разберемся с тем, что обозначает аббревиатура RAG:

- *retrieval («поиск»)* — извлечение релевантных данных;
- *augmented («дополнение»)* — добавление найденных данных к промпту в качестве контекста;

- *generation («генерация»)* — использование дополненного промпта для генерации ответа LLM.

Любая языковая модель ограничена данными, на которых она была обучена. Эти данные иногда называют параметризованным знанием[1]. Следовательно, даже если модель способна точно ответить на вопросы о событиях прошлого, она не имеет доступа к свежей информации или к внешним источникам, с которыми не сталкивалась в процессе обучения.

В качестве примера рассмотрим модель GPT-4o. Для ее обучения использовались данные, собранные до октября 2023 года. Поэтому, если задать ей вопрос о событиях, произошедших во время пандемии 2020 года, она ответит корректно, опираясь на свое параметризованное знание. Однако, если спросить ее о результатах чемпионата Европы по футболу 2024 года, модель не сможет дать точный ответ, поскольку это событие для нее еще не наступило. В некоторых случаях она может даже галлюцинировать, то есть выдать убедительный на вид, но вымышленный или ошибочный ответ.

Механизм RAG устраняет обе эти фундаментальные проблемы. Он позволяет получить доступ к актуальным данным или внешним источникам информации, что снижает вероятность галлюцинаций и повышает точность и надежность работы генеративных моделей ИИ.

Причины для использования RAG

Итак, давайте подробно ответим на вопрос «зачем нужен RAG?», а затем перейдем к описанию базовой RAG-архитектуры.

Чтобы получить интуитивное представление о принципе работы механизма RAG, важно понимать, что при его использовании в промпт внедряется информация, необходимая для ответа на исходный вопрос пользователя. Затем этот дополненный промпт передается в языковую модель, которая формирует финальный ответ, задействуя предоставленный контекст.

Механизм RAG решает две фундаментальные проблемы, связанные:

- с галлюцинациями;
- устаревшими или закрытыми (приватными) данными.

Галлюцинации

Если задать языковой модели, не использующей механизм RAG, вопрос по теме, на которой она не обучалась, высока вероятность того, что она даст правдоподобный, но неверный ответ. Возьмем, к примеру, чемпионат Европы

[1] Это знания, хранящиеся в параметрах уравнения модели. — *Примеч. науч. ред.*

по футболу 2024 года. Если модель была обучена на данных, собранных до октября 2023 года, и мы спросим ее о результатах этого турнира, она, скорее всего, сгенерирует ответ, выглядящий достоверно, но не имеющий отношения к действительности. Даже если галлюцинации возникают не всегда, они подрывают доверие к ответам модели. Поэтому мы должны спросить себя: «Когда ответам LLM можно доверять?» и «Как оценить степень их соответствия действительности?».

С помощью RAG мы принуждаем модель формировать ответ, опираясь исключительно на предоставленный контекст. В этом случае LLM выступает в роли движка рассуждений, а информация, внедренная через механизм RAG, становится единым источником истины для формулирования ответа. Такой подход позволяет оперативно проверить, основан ли вывод модели на внешних данных или нет.

Устаревшие данные

Любая LLM обучается или тонко настраивается лишь на подмножестве всего корпуса знаний. Это обусловлено тремя ключевыми факторами:

- *приватные данные* — модель нельзя обучать на информации, которой вы не владеете или на использование которой не имеете прав;
- *новые данные* — поскольку данные создаются непрерывно, для обеспечения актуальности модель пришлось бы постоянно переобучать;
- *затраты* — обучение и тонкая настройка LLM крайне затратные операции, поэтому выполнять их ежечасно или даже ежедневно нецелесообразно.

Механизм RAG решает эти проблемы. Благодаря ему вам не нужно постоянно тонко настраивать модель на новых (или приватных) данных. Достаточно внедрить необходимые данные непосредственно в промпт, передаваемый LLM, чтобы получить от нее точные и полезные ответы.

Таким образом, механизм RAG является ключевым элементом для создания надежной и гибкой системы на базе генеративного ИИ. Технические детали выбора и внедрения нужных данных в промпт на основе пользовательского запроса мы рассмотрим в следующих разделах.

Базовая архитектура RAG

В своей основе все системы RAG устроены похожим образом (рис. 4.1). Сперва мы рассмотрим ее простейшую версию, а затем — более продвинутые техники RAG, направленные на повышение точности работы модели. Чтобы избежать повторов, в этой главе наряду с термином «базовая архитектура RAG» будут использоваться такие синонимы, как «стандартная» и «классическая».

Система RAG включает три основных, независимых друг от друга модуля:

- *пайплайн загрузки данных* (ingestion pipeline) — пайплайн пакетной или потоковой обработки, заполняющий векторную БД;
- *пайплайн поиска* (retrieval pipeline) — модуль, который обрабатывает пользовательский запрос и извлекает релевантные данные из векторной БД;
- *пайплайн генерации* (generation pipeline) — компонент, использующий извлеченные данные для дополнения промпта и его передачи в LLM для генерации ответа.

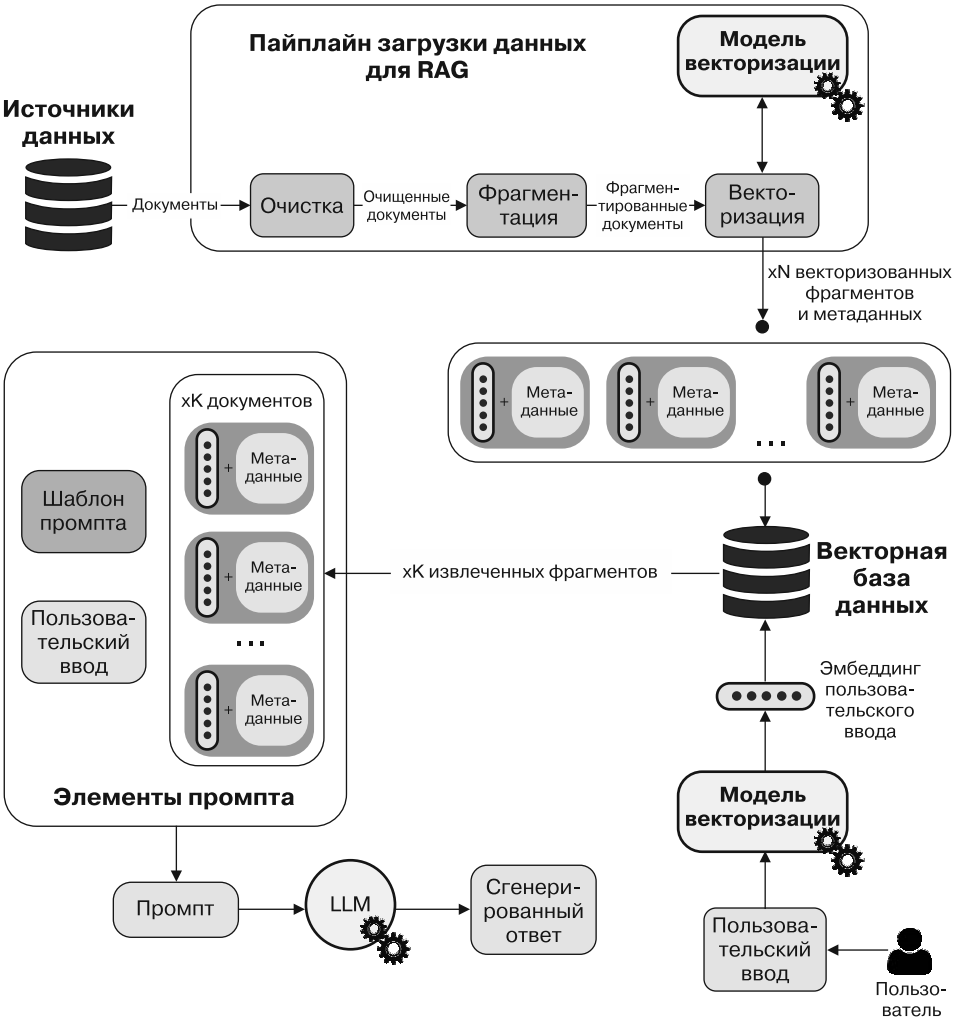


Рис. 4.1. Классическая архитектура RAG

Поскольку каждый из этих компонентов представляет собой отдельный класс или сервис, мы рассмотрим их по очереди. Но прежде кратко опишем, как они связаны между собой.

1. На стороне сервера пайплайн загрузки данных запускается по расписанию или работает в постоянном режиме, пополняя векторную базу данными из внешних источников.
2. На стороне клиента пользователь задает вопрос.
3. Этот вопрос передается в модуль поиска, который предварительно обрабатывает пользовательский ввод и формирует запрос к векторной БД.
4. Пайплайн генерации объединяет шаблон промпта, исходный запрос и найденный контекст, формируя окончательный промпт.
5. Получив промпт, LLM генерирует ответ.
6. Ответ предоставляется пользователю.

Механизм RAG следует реализовать в любом приложении на базе генеративного ИИ, если нужен доступ к внешним источникам данных. Например, цифровому финансовому помощнику, скорее всего, потребуется обращаться к последним новостям, отчетам и котировкам, чтобы предоставлять пользователю ценные ответы. А рекомендательная система для путешественников должна будет извлекать и обрабатывать списки достопримечательностей, ресторанов и мероприятий. Во время обучения LLM не имеет доступа к специфическим пользовательским данным, поэтому при реализации большинства проектов на базе генеративного ИИ приходится внедрять стратегию RAG. Теперь давайте подробно рассмотрим пайплайны загрузки данных, поиска и генерации.

Пайплайн загрузки данных

Пайплайн загрузки данных механизма RAG извлекает исходные документы из различных источников (например, хранилищ данных, озер данных, веб-страниц и т. д.). Затем он очищает эти данные, разбивает их на фрагменты и векторизует, то есть преобразует в эмбеддинги, которые загружаются в векторную БД (или иное векторное хранилище).

Таким образом, пайплайн загрузки данных включает следующие компоненты.

- *Модуль извлечения данных* собирает необходимые данные из различных источников, таких как базы данных, API или веб-страницы. Принцип работы этого модуля зависит от ваших данных. Он может сводиться как к простым SQL-запросам к хранилищу, так и к реализации более сложных процессов, вроде сбора данных со страниц Википедии.
- *Слой очистки* стандартизирует извлеченные данные и удаляет из них нежелательные символы, например не входящие в набор ASCII, а также выделенные жирным шрифтом или курсивом. Популярной практикой также

является замена URL-адресов заполнителями (placeholders). Однако выбор стратегии очистки зависит от конкретного источника данных и используемой модели векторизации.

- *Модуль разбиения на фрагменты* разбивает очищенные документы на более мелкие части. Поскольку содержимое документов передается модели векторизации, нам необходимо следить за тем, чтобы объем входного текста не превышал допустимый предел. Кроме того, разбиение помогает выделять семантически связанные фрагменты. Например, при разбиении главы книги близкие по смыслу абзацы логично группировать в один фрагмент. Это позволяет добавлять к промпту только релевантные данные на этапе поиска.
- *Компонент векторизации* использует модель векторизации для преобразования содержимого фрагмента (текста, изображений, аудио и т. д.) в вектор, насыщенный семантической информацией. Подробнее о векторизации мы поговорим в подразделе «Что такое эмбеддинги».
- *Модуль загрузки* сохраняет эмбеддинги вместе с метаданными, включая содержимое фрагмента, URL-адрес источника и дату публикации в Интернете. Эмбеддинг используется в качестве индекса для поиска схожих фрагментов, а метаданные — для получения информации, добавляемой к промпту.

Итак, теперь у нас есть RAG-пайплайн загрузки данных, который принимает на вход исходные документы, обрабатывает их и заполняет векторную БД. Далее мы поговорим о корректном извлечении релевантных данных из векторного хранилища.

Пайплайн поиска

Пайплайн поиска принимает входные данные пользователя (текст, изображение, аудио и т. д.), преобразует их в эмбеддинги (векторные представления) и извлекает наиболее близкие к ним векторы из векторной БД.

Основная задача этапа поиска — отобразить пользовательский ввод в то же векторное пространство, что и эмбеддинги, используемые в векторной БД в качестве индексов. Это позволяет найти K наиболее похожих фрагментов путем сравнения эмбеддинга пользовательского ввода с векторами, сохраненными в базе. Затем найденные фрагменты применяются в качестве контекста для дополнения промпта, который передается в LLM для генерации ответа.

Для сравнения двух векторов необходимо использовать метрику расстояния, например евклидово или манхэттенское расстояние. Однако наиболее популярной метрикой является косинусное расстояние, равное разности единицы и косинуса угла между двумя векторами:

$$\textit{Косинусное расстояние} = 1 - \cos(\theta) = 1 - \frac{A \cdot B}{\|A\| \cdot \|B\|}.$$

Значение этой метрики находится в диапазоне от –1 до 1. Она равна –1, если векторы A и B направлены строго в противоположные стороны, 0, если они ортогональны, и 1, если направлены в одну сторону.

Как правило, косинусное расстояние хорошо работает в сложных нелинейных векторных пространствах. Однако важно понимать, что выбор подходящей метрики зависит от типа данных и используемой модели векторизации.

Очень важно, чтобы пользовательский ввод и эмбеддинги в базе находились в одном и том же векторном пространстве. В противном случае вычислить расстояние между ними будет невозможно. В связи с этим пользовательский ввод должен обрабатываться тем же способом, что и исходные документы, подаваемые в RAG-пайплайн загрузки данных. Иначе говоря, пользовательский ввод нужно очистить, разбить на фрагменты (при необходимости) и векторизовать с использованием тех же функций, моделей и гиперпараметров. Аналогичная ситуация возникает при работе с признаками: если на этапе инференса данные обрабатываются иначе, чем при обучении модели, результат будет некорректным. Такое несоответствие называется рассогласованием между обучением и инференсом.

Пайплайн генерации

Последний этап работы RAG-системы предполагает формирование промпта из преобразованного пользовательского ввода, дополненного релевантными данными, а также его передачу в LLM для генерации содержательного ответа.

Итоговый промпт формируется на основе системного шаблона и шаблона промпта, которые заполняются пользовательским запросом и извлеченным контекстом. В приложении может применяться как один шаблон промпта, так и несколько. Обычно вся инженерия промптов сосредоточена именно на уровне шаблонов.

Ниже показано, как могут выглядеть универсальный системный шаблон и шаблон промпта и каким образом они используются совместно с логикой поиска и LLM для генерации ответа:

```
system_template = """
Ты — ассистент, который вежливо отвечает на все вопросы пользователя.
"""

prompt_template = """
Отвечай на вопросы пользователя, используя только предоставленный контекст.
Если не можешь ответить, используя контекст, ответь: "Я не знаю."

Context: {context}
User question: {user_question}
"""
```

```
user_question = "<your_question>"
retrieved_context = retrieve(user_question)

prompt = f"{system_template}\n"
prompt += prompt_template.format(context=retrieved_context,
user_question=user_question)

answer = llm(prompt)
```

По мере развития шаблонов промптов все изменения должны отслеживаться и версионироваться в соответствии с лучшими практиками *MLOps*. Это позволяет в любой момент установить, что конкретный ответ был сгенерирован определенной версией LLM и шаблона (шаблонов) промпта. Для этого можно использовать систему контроля версий (например, Git), базу данных или специализированные инструменты управления промптами, такие как LangFuse.

Как было показано в разделе о пайплайне поиска, на точность работы RAG-системы влияют: эмбеддинги внешних данных (обычно хранящиеся в векторных БД), эмбеддинг пользовательского запроса и способ нахождения сходства между ними (например, косинусное расстояние).

Чтобы лучше понять этот аспект алгоритма RAG, далее мы подробно рассмотрим эмбеддинги и способы их вычисления.

Что такое эмбеддинги

Представьте, что вы пытаетесь научить компьютер понимать окружающий мир. В этом случае эмбеддинг играет роль своеобразного «переводчика», который преобразует реальные объекты в числовой код. Однако такое кодирование выполняется не случайным образом, поскольку похожим словам или объектам присваиваются близкие по значению коды. В итоге получается своеобразная карта, на которой семантически близкие понятия группируются в одной и той же области пространства.

Согласно более формальному определению, эмбеддинги — это числовые представления объектов (вроде слов, изображений или элементов рекомендательной системы), закодированные в виде векторов в непрерывном векторном пространстве. Такое преобразование позволяет уловить семантическое значение и взаимосвязи между объектами. Например, в сфере *обработки естественного языка* (natural language processing, NLP) в результате этого преобразования семантически близкие слова оказываются рядом друг с другом в векторном пространстве.

Распространенной практикой является визуализация эмбеддингов с целью анализа их взаимного расположения. Поскольку эмбеддинги часто имеют не два или три измерения, а от 64 до 2048, их необходимо дополнительно проецировать в двумерное или трехмерное пространство (рис. 4.2).

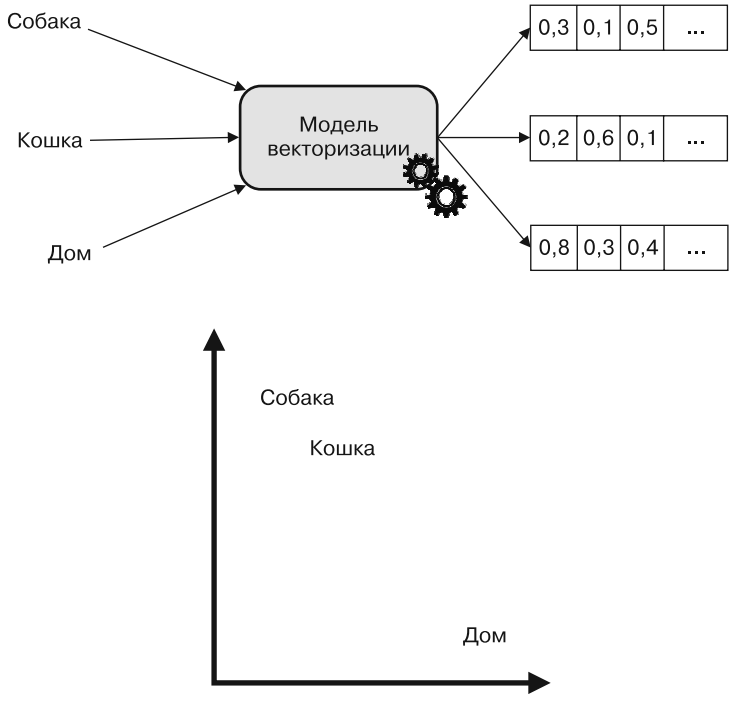


Рис. 4.2. Эмбеддинги

Для этого можно использовать UMAP (https://umap-learn.readthedocs.io/en/latest/index.html), метод понижения размерности, хорошо сохраняющий геометрические отношения между точками при их проецировании в двумерное или трехмерное пространство (рис. 4.3). Другим популярным алгоритмом визуализации векторных представлений является t-SNE (https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html). Однако по сравнению с UMAP он менее предсказуемый и хуже сохраняет близость точек, которые в многомерных векторах были близки.



В целом алгоритмы понижения размерности, такие как PCA, UMAP и t-SNE, представляют собой математические методы, позволяющие уменьшить число входных переменных или признаков в наборе данных при сохранении наиболее значимых структур, закономерностей и взаимосвязей. Их цель — преобразовать данные высокой размерности в более низкую размерность, что облегчает визуализацию, интерпретацию и последующую обработку, минимизируя потерю информации. Эти методы помогают бороться с проблемой, называемой проклятием размерности, повышают эффективность вычислительных процессов и нередко улучшают работу алгоритмов машинного обучения.

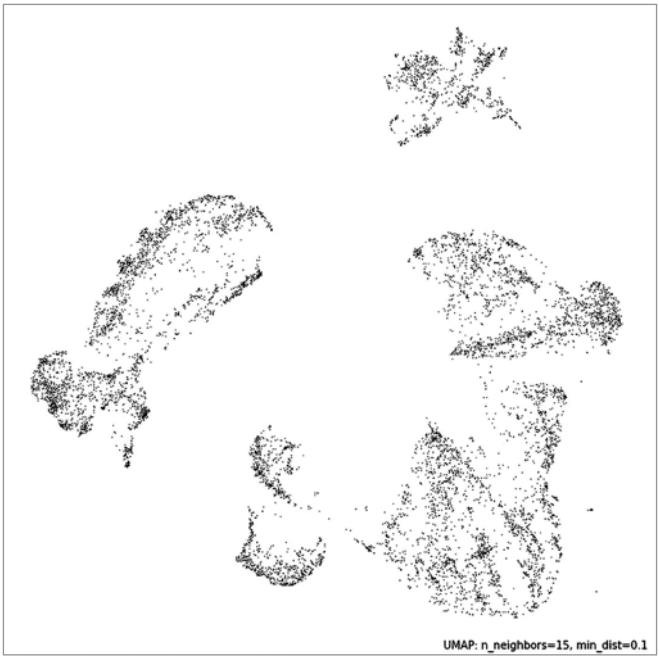


Рис. 4.3. Визуализация эмбеддингов с помощью UMAP (источник: документация UMAP)

Почему эмбеддинги настолько эффективны

Во-первых, ML-модели работают только с числовыми значениями. Это не вызывает затруднений при использовании табличных данных, так как они уже представлены в числовой форме или легко преобразуются в нее. Однако для подачи в модель слов, изображений или аудиоданных требуются эмбеддинги.

Например, при работе с трансформерами входной текст сначала токенизируется и с каждым токеном сопоставляется соответствующий эмбеддинг. Прелесть этого процесса заключается в его простоте: на вход трансформеру подается последовательность эмбеддингов, которую легко интерпретируют плотные слои нейросети.

На основе этого примера можно сделать вывод, что эмбеддинги подходят для кодирования любых категориальных переменных, которые затем подаются на вход модели. Но почему бы не использовать более простые методы, такие как *быстрое кодирование (one-hot encoding)*? Дело в том, что при применении классических методов кодирования к категориальным переменным высокой кардинальности (например, к словарям естественного языка) вы неминуемо столкнетесь с так называемым проклятием размерности. Например, если сло-

варь содержит 10 000 токенов, то после применения унитарного кодирования каждый токен будет представлен вектором длиной 10 000, содержащим одну единицу и 9999 нулей. Если входная последовательность содержит N токенов, то общее число входных параметров составит $N \times 10\,000$. При $N \geq 100$ (что типично для текстов) объем входных данных становится слишком большим для эффективной обработки. Другие классические методы, не страдающие от проклятия размерности, в частности *хеширование*, имеют другой недостаток — они разрушают семантические отношения между представлениями.



Унитарное кодирование — это метод преобразования категориальных переменных в бинарное матричное представление. Каждая категория представляется уникальным двоичным вектором. Для каждой переменной создается вектор длиной, равной числу уникальных категорий, в котором нулю равны все значения, за исключением позиции, соответствующей конкретной категории, — там устанавливается единица. Этот метод сохраняет всю информацию о категориях, отличается простотой и интерпретируемостью. Однако его существенным недостатком является то, что при большом числе уникальных значений размерность признакового пространства становится очень высокой, что делает применение данного метода непрактичным.

Хеширование признаков (также известное как трюк с хешированием) — это техника, используемая для преобразования категориальных переменных в числовые признаки путем применения хеш-функции к значениям категорий. В отличие от унитарного кодирования этот метод не зависит от количества уникальных категорий: он отображает их в фиксированное число ячеек (bins или buckets), тем самым уменьшая размерность признакового пространства. Это особенно полезно при работе с категориальными переменными высокой кардинальности, поскольку позволяет существенно уменьшить объем потребляемой памяти и вычислительных ресурсов. Однако данный метод допускает возможность коллизий, когда разные категории попадают в одну ячейку, что приводит к потере информации. Результат такого преобразования трудно интерпретировать из-за утраты связи между исходной категорией и хешированными признаками.

Эмбеддинги предоставляют нам возможность кодировать категориальные переменные, контролируя при этом размерность выходного вектора. В отличие от стандартных хеш-методов они используют более изощренные способы сжатия информации, позволяющие получить компактные и информативные векторные представления.

Во-вторых, преобразование входных данных в эмбеддинг позволяет уменьшить их размерность и сжать все семантическое содержание в плотный вектор. Этот подход широко используется при работе с изображениями: кодировщик на базе сверточной нейросети (CNN-кодер, от англ. convolutional neural networks) отображает представление высокой размерности в эмбеддинг, который затем передается в CNN-декодер, выполняющий задачу классификации или регрессии.

На рис. 4.4 показана типичная архитектура сверточной нейронной сети. Маленькие квадратики внутри каждого слоя — это так называемые рецептивные поля. Каждый такой квадрат передает информацию одному нейрону следующего слоя. По мере продвижения через сеть происходят два ключевых процесса:

- *сжатие изображения* — операции подвыборки уменьшают размер слоев, выделяя только наиболее важные детали;
- *извлечение признаков* — операции свертки, напротив, увеличивают размер слоев, по мере того как сеть обучается извлекать все более сложные признаки из изображения.

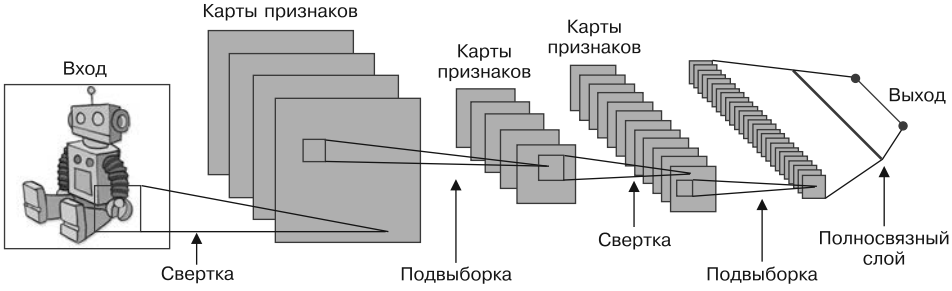


Рис. 4.4. Создание эмбеддингов из изображения с помощью сверточной нейросети

В конце этого процесса полносвязный слой нейросети принимает на вход всю предварительно обработанную информацию и преобразует ее в итоговый вектор — эмбеддинг, представляющий исходное изображение в числовой форме (например, если на изображении кошка в комнате на диване, то все обьекты и их характеристики кодируются в один вектор).



Приведенное выше изображение взято из хранилища Wikimedia Commons (https://commons.wikimedia.org/wiki/File:Typical_cnn.png) и распространяется на условиях лицензии Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0: https://creativecommons.org/licenses/by-sa/4.0/deed.ru).

Создание эмбеддингов

Эмбеддинги создаются с помощью моделей глубокого обучения, которые распознают контекст и семантику входных данных и проецируют их в непрерывное векторное пространство.

В зависимости от типа входных данных для создания эмбеддингов могут использоваться различные модели глубокого обучения, поэтому, прежде чем выбирать способ векторизации, крайне важно понять структуру данных и цели, которые вы перед собой ставите.

Что касается текстовых данных, то одними из первых методов создания эмбеддингов словарей стали Word2Vec и GloVe, которые до сих пор широко применяются для решения простых задач.

Другой распространенный подход заключается в использовании таких трансформеров, как BERT, или других моделей из того же семейства, например RoBERTa. Эти модели задействуют кодировщик трансформеров для эффективного отображения входных данных в плотное векторное пространство, элементы которого затем и служат эмбеддингами.

Для создания эмбеддингов в Python предусмотрен пакет `SentenceTransformer` (входит в состав библиотеки Hugging Face Transformers). Этот инструмент предоставляет удобный интерфейс, упрощающий и ускоряющий процесс векторизации.

В приведенном ниже фрагменте кода мы загружаем модель из `SentenceTransformer`, вычисляем эмбеддинги для трех предложений и, наконец, рассчитываем косинусное сходство между ними. Сходство предложения с самим собой всегда равно единице, между первым и вторым предложением — близко к нулю, поскольку они не имеют ничего общего, а между первым и третьим чуть выше, так как в них есть частичное совпадение по смыслу:

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("all-MiniLM-L6-v2")

sentences = [
"Собака сидит на улице и ждет угощения.",
"Я иду плавать.",
"Собака плавает."
]

embeddings = model.encode(sentences)
print(embeddings.shape)
# Ответ: [3, 384]

similarities = model.similarity(embeddings, embeddings)
print(similarities)
# Ответ:
# tensor([[ 1.0000, -0.0389, 0.2692],
# [-0.0389, 1.0000, 0.3837],
# [ 0.2692, 0.3837, 1.0000]])
#
# similarities[0, 0] = Сходство первого предложения с самим собой
# similarities[0, 1] = Сходство между первым и вторым предложениями
# similarities[2, 1] = Сходство между третьим и вторым предложениями
```

Исходный код этого примера доступен по адресу https://github.com/PacktPublishing/LLM-Engineering/blob/main/code_snippets/08_text_embeddings.py.



Примеры из раздела об эмбеддингах можно запускать в виртуальном окружении, используемом на протяжении всей книги, так как оно содержит все необходимые зависимости.

Следует учитывать, что модель векторизации, обеспечивающая наилучшие результаты, может меняться со временем и в зависимости от решаемой вами задачи. Найти подходящие модели с наилучшими результатами в целом, с максимальной точностью или с минимальным объемом занимаемой памяти можно с помощью бенчмарка *Massive Text Embedding Benchmark (MTEB)* от Hugging Face. Все зависит от ваших требований. Вместе с тем библиотека Hugging Face `SentenceTransformer` упрощает переключение между моделями, что позволяет вам экспериментировать с различными вариантами.

При работе с изображениями эмбеддинги создаются с помощью *сверточных нейросетей* (convolutional neural network, CNN). Наиболее популярные из них основаны на архитектуре ResNet. Однако использовать те же методы в отношении аудиозаписей невозможно. Вместо этого аудиосигналы преобразуются в визуальные представления (например, в спектрограммы), к которым затем применяются модели векторизации изображений. Это позволяет извлекать суть изображений и звуков в понятной компьютеру форме.

Благодаря моделям вроде CLIP (Contrastive Language-Image Pre-training — «контрастное предобучение на языке и изображениях») можно создавать эмбеддинги для текста и изображений в едином векторном пространстве. Это помогает, например, находить релевантные изображения по текстовому описанию или наоборот, что делает использование CLIP особенно практичным.

В приведенном ниже фрагменте кода мы применяем модель CLIP для векторизации изображения сумасшедшей кошки и трех предложений. В конце мы оцениваем степень сходства между изображением и текстами с помощью косинусного расстояния:

```
from io import BytesIO

import requests
from PIL import Image
from sentence_transformers import SentenceTransformer

response = requests.get(
    "https://github.com/PacktPublishing/LLM-Engineering/blob/main/images/
     ➥crazy_cat.jpg?raw=true"
)

image = Image.open(BytesIO(response.content))

model = SentenceTransformer("clip-ViT-B-32")
```

```
img_emb = model.encode(image)
text_emb = model.encode(
["Сумасшедшая кошка улыбается.",
"Бело-коричневая кошка с желтым бананом.",
"Человек, который ест в саду."]
)
print(text_emb.shape) # noqa
# Ответ: (3, 512)

similarity_scores = model.similarity(img_emb, text_emb)
print(similarity_scores) # noqa
# Ответ: tensor([[0.3068, 0.3300, 0.1719]])
```

Исходный код приведенного примера доступен по адресу https://github.com/PacktPublishing/LLM-Engineering/blob/main/code_snippets/08_text_image_embeddings.py.

Итак, мы кратко рассмотрели процесс вычисления эмбеддингов. На практике существует огромное количество специализированных реализаций, однако главное — понять, что эмбеддинги можно вычислять практически для любых цифровых объектов — слов, предложений, документов, изображений, видео и графов.

Важно знать, что при сравнении объектов разных типов (например, текста и изображения) необходимо использовать специализированные модели, такие как CLIP. Они проецируют оба типа данных в общее векторное пространство, что позволяет корректно вычислять расстояние между ними.

Способы применения эмбеддингов

С началом революции в области генеративного ИИ, использующего RAG-механизмы, эмбеддинги стали крайне популярны при решении таких задач информационного поиска, как семантический поиск по тексту, коду, изображениям и аудио, а также при реализации долговременной памяти интеллектуальных агентов. Однако и до появления генеративного ИИ эмбеддинги активно применялись для:

- представления категориальных переменных (токенов словаря), подаваемых на вход ML-модели (например, определение категории письма: спам/не спам);
- кодирования пользователей и объектов в рекомендательных системах для дальнейшей оценки их сходства (например, в YouTube для поиска похожих роликов);
- кластеризации и обнаружения выбросов[1] (например, новости с похожими эмбеддингами соответствуют одной тематике, новости с почти одинаковыми эмбеддингами — дубликаты);

---

[1] Выбросы, или аномалии, — это данные, сильно отличающиеся от остальных примеров в наборе. — *Примеч. науч. ред.*

- визуализации данных с помощью таких алгоритмов, как UMAP (например, тепловая карта с темами на форуме);
- классификации с использованием эмбеддингов в качестве признаков (например, получение набора данных о тональности отзывов о товаре; модель предварительно учится на размеченных отзывах);
- классификации по принципу zero-shot (без обучающих примеров) путем сравнения эмбеддинга каждого класса с эмбеддингом входного объекта и выбора наиболее близкого к нему (поиск похожих эмбеддингов, например, в отзыве, чтобы определить, есть ли в нем негатив; используется без предварительного обучения путем получения эмбеддингов из заранее выбранных характеристик, которые мы будем искать в тексте).

Для полного понимания работы механизма RAG нам осталось обсудить векторные базы данных и то, как они задействуют эмбеддинги для извлечения релевантной информации.

Векторные базы данных

Векторные базы данных — это специализированные БД, предназначенные для эффективного хранения, индексации и извлечения векторных эмбеддингов. Традиционные БД, работающие со скалярными значениями, плохо справляются со сложностью векторных данных, что делает векторные БД ключевым инструментом при решении таких задач, как семантический поиск в реальном времени.

Хотя автономные инструменты индексации, вроде FAISS, хорошо подходят для поиска на основе сходства, им недостает функций управления данными, присущих полноценным векторным БД. Последние поддерживают операции CRUD, фильтрацию по метаданным, масштабирование, обновление в реальном времени, резервное копирование, интеграцию в существующую инфраструктуру и обеспечивают высокий уровень безопасности, что делает их более подходящими для использования в продакшене.

Принцип работы векторной базы данных

Традиционные БД возвращают данные, точно совпадающие с введенным вами запросом. Векторные же работают иначе. Их задача состоит не в поиске точных совпадений, а в нахождении векторов, наиболее близких к вектору запроса. Для этого используются алгоритмы *приближенного поиска ближайших соседей* (Approximate Nearest Neighbors, ANN).

Хотя ANN-алгоритмы не гарантируют абсолютной точности (то есть не всегда находят наилучшие совпадения), они работают значительно быстрее, чем клас-

сические алгоритмы поиска ближайших соседей, и дают достаточно хорошие результаты. Поэтому на практике выбор часто делается в пользу ANN, как оптимального компромисса между точностью и скоростью.

Типичный рабочий процесс векторной БД включает три этапа.

1. *Индексация векторов*. Векторы индексируются с помощью структур данных, оптимизированных для пространств высокой размерности. К наиболее популярным методам относятся: *HNSW-граф*, *случайные проекции*, *квантование по продукту* и *локально-чувствительное хеширование*.
2. *Поиск похожих векторов*. При обработке запроса система ищет среди проиндексированных векторов те, которые максимально близки к входному вектору. Для сравнения векторов может использоваться косинусное сходство, евклидово расстояние или скалярное произведение. Каждая из метрик обладает уникальными преимуществами и подходит для решения разных задач.
3. *Постобработка результатов*. После выявления потенциальных совпадений результаты проходят постобработку для повышения точности, что позволяет вернуть пользователю наиболее релевантные векторы.

Векторные БД могут выполнять фильтрацию результатов по метаданным как до, так и после осуществления векторного поиска. Оба подхода предусматривают некий компромисс между производительностью и точностью. Поскольку запрос зависит не только от векторного индекса, но и от метаданных, он также содержит индекс метаданных, необходимый для выполнения соответствующей фильтрации.

Алгоритмы построения векторного индекса

Для построения индекса и эффективной организации поиска векторные БД используют различные алгоритмы. Вот некоторые из них.

- *Случайные проекции* (Random Projection). Этот метод снижает размерность векторов путем их проецирования в пространство меньшей размерности с помощью случайной матрицы, сохраняя относительные расстояния между векторами, что ускоряет процесс поиска.
- *Квантование по продукту* (Product Quantization, PQ). При использовании этого метода векторы разбиваются на более мелкие вложенные векторы, которые затем квантуются[1] в компактные кодовые представления. Такой

[1] Метод сжатия данных, который разбивает многомерные векторы (например, эмбеддинги) на небольшие сегменты и сжимает каждый из них независимо с помощью поиска ближайших «прототипов». Это ускоряет поиск сходств и сокращает расход памяти. — *Примеч. науч. ред.*

прием уменьшает объем потребляемой памяти и ускоряет поиск схожих векторов.

- *Локально-чувствительное хеширование* (Locality-Sensitive Hashing, LSH). Этот метод отображает похожие векторы в одну и ту же ячейку (bucket), что позволяет выполнять приближенный поиск ближайших соседей, фокусируясь на подмножестве данных и снижая тем самым вычислительную сложность.
- *HNSW-граф* (Hierarchical Navigable Small World — «иерархический граф малого мира»). Данный алгоритм строит многослойный граф, каждая вершина которого представляет набор векторов. Схожие вершины соединяются между собой, что предоставляет возможность перемещаться по графу и эффективно находить ближайших соседей.

Эти алгоритмы позволяют векторным БД эффективно обрабатывать сложные и объемные данные, что делает их применение особенно подходящим в приложениях на основе ИИ и машинного обучения.

Операции в векторных базах данных

Векторные базы данных унаследовали от традиционных СУБД множество операций, обеспечивающих высокую производительность, отказоустойчивость и удобство управления в продакшене. К ним относятся:

- *фрагментация* — данные разбиваются на части и распределяются по разным узлам, что обеспечивает масштабируемость и высокую доступность;
- *репликация* — создание копий данных на разных узлах помогает сохранить целостность и доступность информации в случае сбоя;
- *мониторинг* — постоянный мониторинг производительности БД, включая отслеживание задержки при обработке запросов и уровня использования ресурсов (оперативной памяти, процессора, дисков), позволяет поддерживать стабильную работу и заранее выявлять потенциальные проблемы;
- *контроль доступа* — реализация надежных механизмов контроля доступа гарантирует, что только авторизованные пользователи смогут обращаться к данным и изменять их (к таким механизмам, например, относятся ролевая модель разграничения прав и другие протоколы защиты конфиденциальной информации);
- *резервное копирование* — регулярное создание резервных копий данных критически важно для восстановления после сбоев, а автоматизированные процессы резервного копирования позволяют вернуть систему к предыдущему рабочему состоянию в случае потери или повреждения данных.

Обзор продвинутой RAG-архитектуры

Представленная выше классическая архитектура RAG не учитывает ряд фундаментальных факторов, влияющих на качество поиска и генерации ответов.

- Насколько извлеченные документы релевантны запросу пользователя?
- Достаточно ли извлеченного контекста для формирования корректного ответа?
- Содержат ли данные избыточную информацию, которая лишь загромождает дополненный промпт?
- Соответствует ли задержка на этапе поиска нашим требованиям?
- Что делать, если на основе извлеченных данных невозможно сгенерировать валидный ответ?

Из этих вопросов можно сделать два ключевых вывода. Во-первых, нам необходим надежный модуль оценки качества для системы RAG, который будет количественно измерять релевантность извлеченных данных и соотносить сгенерированные ответы с исходным запросом пользователя. Об этом мы подробно поговорим в главе 9. Во-вторых, нам нужно усовершенствовать саму архитектуру RAG, чтобы устранить ограничения поиска на уровне алгоритма. RAG-архитектура, включающая эти усовершенствования, называется продвинутой (advanced RAG).

Оптимизация классической архитектуры RAG может быть реализована на трех этапах (рис. 4.5).

- *Подготовка запроса к поиску (pre-retrieval)*. На этом этапе основное внимание уделяется структурированию и предварительной обработке данных для оптимизации как индексации, так и самих запросов.
- *Поиск (retrieval)*. Здесь акцент делается на улучшении моделей векторизации и фильтрации по метаданным, чтобы сделать векторный поиск более точным.
- *Работа с найденными документами перед генерацией (post-retrieval)*. Этот этап предусматривает применение методов устранения шума из выборки и сжатия промпта перед его подачей в LLM для генерации ответа.

Данный раздел не претендует на исчерпывающее описание всех методов оптимизации RAG-архитектуры. Наша цель — дать вам общее представление о возможных вариантах. В качестве примеров будут использоваться исключительно текстовые данные, однако сами принципы универсальны и не зависят от типа данных. Теперь давайте рассмотрим каждый из трех вышеупомянутых этапов более подробно.

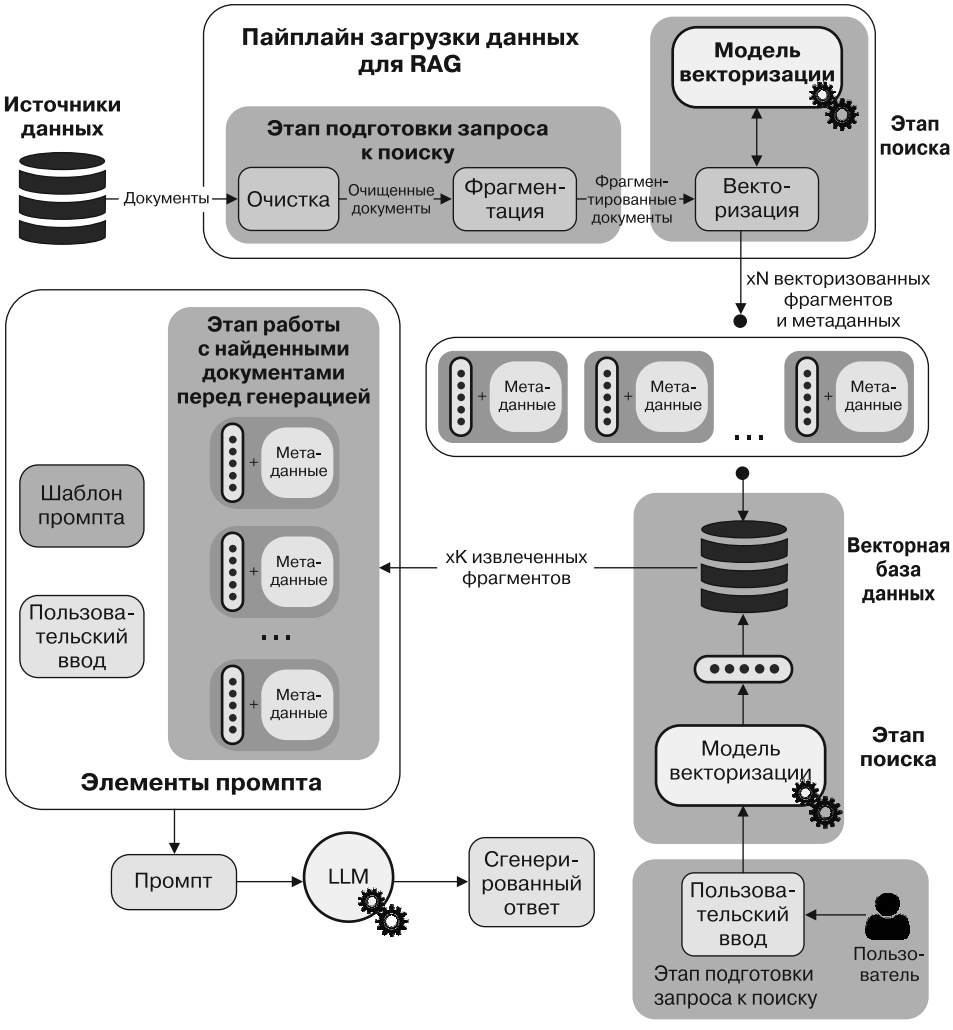


Рис. 4.5. Три этапа оптимизации RAG-архитектуры

Этап подготовки запроса к поиску

На предварительном этапе оптимизация реализуется двумя основными способами:

- *оптимизация индексации данных* выполняется в рамках работы RAG-пайплайна загрузки данных (как правило, за нее отвечают модули очистки или разбиения на фрагменты);

- *оптимизация запроса* пользователя выполняется непосредственно перед его векторизацией и извлечением соответствующих фрагментов из векторной БД.

Поскольку данные индексируются на основе эмбеддингов, отражающих семантическое содержание фрагментов, большинство методов *оптимизации индексации* ориентировано на предварительную обработку и улучшение структуры данных для повышения эффективности поиска.

- *Метод скользящего окна*. Обеспечивает пересечение между соседними текстовыми фрагментами, что позволяет сохранить важный контекст на границах и повысить точность поиска. Это особенно важно при работе с такими материалами, как юридические документы, научные статьи, логи общения технической поддержки с пользователями и медицинские записи, поскольку ключевая информация часто распределена в них между несколькими разделами. При этом эмбеддинг вычисляется с учетом как основного фрагмента, так и перекрывающейся части. Таким образом, метод скользящего окна помогает системе извлекать более релевантную и согласованную информацию за счет сохранения контекста на границах фрагментов.
- *Увеличение степени детализации и точности данных*. Включает такие приемы, как удаление нерелевантных сведений, проверка достоверности информации и обновление устаревших данных. Чистый набор данных способствует получению более точных результатов поиска.
- *Добавление метаданных*. Включение таких метаданных, как даты, URL-адреса, внешние идентификаторы или указатели глав, позволяет эффективно фильтровать результаты на этапе поиска.
- *Оптимизация структуры индекса*. Осуществляется за счет применения различных методов индексирования данных, включая варьирование размера фрагментов и использование стратегий множественной индексации.
- *Алгоритм «от меньшего к большему» (small-to-big)*. В этом случае фрагменты, которые в последующем будут использоваться для поиска, отделяются от контекста, подаваемого в LLM для генерации ответа. Эмбеддинг вычисляется на основе короткого фрагмента текста, в то время как сам текст вместе с более обширной областью контекста сохраняется в метаданных. Таким образом, если задействовать небольшие фрагменты, это повысит точность поиска, а более широкий контекст обеспечит полноту ответа, генерируемого LLM.

  Суть в том, что использование в процессе векторизации всего фрагмента текста может привнести слишком много шума или объединить несколько разных тем, что приведет к ухудшению семантического представления эмбеддинга.

Для *оптимизации запроса* можно применить такие техники, как *маршрутизация*, *переформулирование* и *расширение*, позволяющие повысить релевантность извлекаемой информации.

- *Маршрутизация запроса*. В зависимости от пользовательского ввода мы можем быть вынуждены взаимодействовать с разными категориями данных, каждая из которых обрабатывается по-своему. Маршрутизатор запроса выбирает нужное действие, аналогично конструкции `if-else`, но на основе естественного языка, а не логических условий (рис. 4.6).

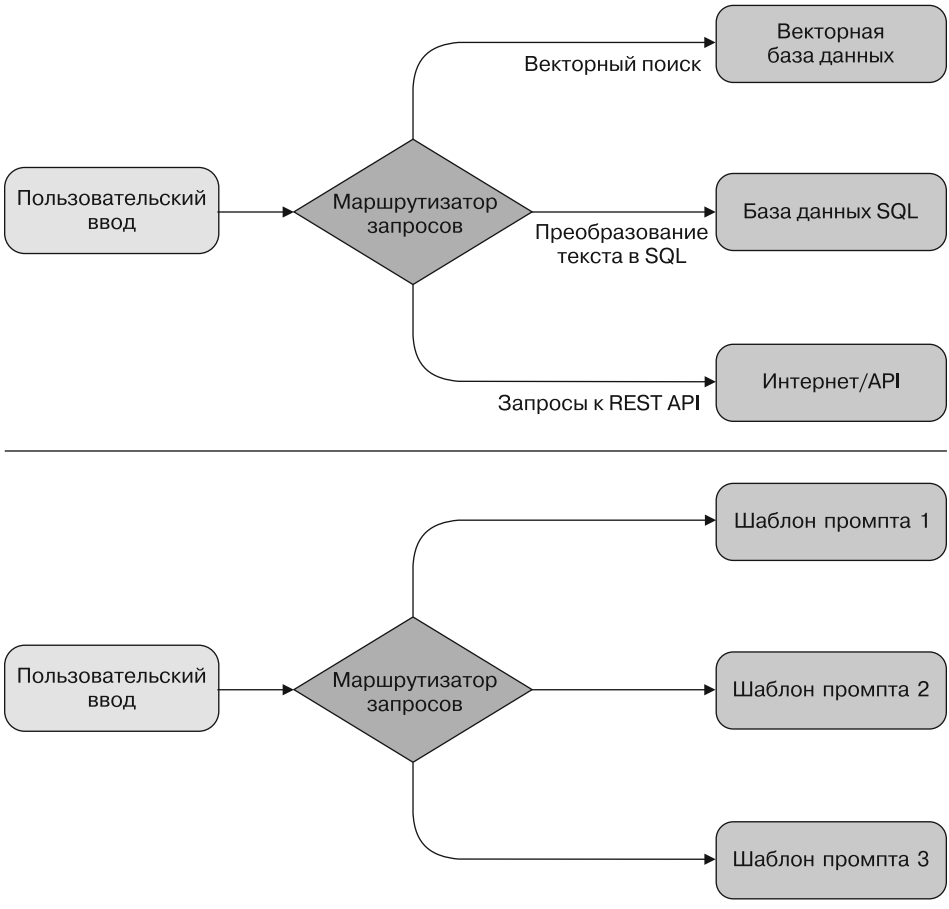


Рис. 4.6. Маршрутизация запроса

Допустим, в зависимости от пользовательского ввода для выполнения запроса в рамках RAG дополнительный контекст можно получить из разных источников: из векторной БД с помощью векторного поиска, из обычной

SQL-базы, преобразовав пользовательский запрос в SQL-команду, или из Интернета — через обращения к REST API (см. рис. 4.6). Маршрутизатор способен определить, насколько нам вообще нужен дополнительный контекст, что позволяет избежать лишних обращений к внешним источникам. Кроме того, он может выбрать наиболее подходящий шаблон промпта в зависимости от запроса. Например, для оптимизации генерации в системе LLM-двойника мы можем использовать разные шаблоны промптов в зависимости от того, что именно запрашивает пользователь — абзац статьи, пост или фрагмент кода. В большинстве случаев маршрутизация осуществляется с помощью LLM или эмбеддингов (за счет выбора пути с наибольшим сходством в векторном пространстве), что функционально аналогично конструкции `if-else`.

- *Переформулирование запроса*. Иногда исходный запрос пользователя не соответствует структуре индексированных данных. В таких случаях имеет смысл переформулировать запрос, используя следующие приемы:
  - *перефразирование* — изменение формулировки при сохранении смысла (например: «Каковы причины изменения климата?» → «Факторы, способствующие глобальному потеплению»);
  - *подстановка синонимов* — использование более распространенных слов вместо относительно редких (например: «жизнерадостный» → «счастливый»);
  - *разбиение на подзапросы* — чтобы упростить поиск релевантных документов, длинные запросы можно разбить на несколько более коротких и узконаправленных.
- *Генерация эмбеддингов вместе с ответом от LLM (Hypothetical Document Embeddings, HyDE)*. При использовании этого метода LLM сначала генерирует гипотетический ответ, который вместе с исходным запросом подается на вход пайплайна поиска.
- *Расширение запроса*. Метод предполагает обогащение исходного запроса дополнительными терминами и понятиями, раскрывающими его с разных сторон. Например, при поиске по слову «болезнь» можно добавить слова «заболевание» и «недуг», чтобы расширить охват.
- *Структурирование запроса через LLM (self-query)*. Суть этого метода заключается в том, чтобы преобразовать неструктурированные запросы в структурированные. LLM анализирует входной текст и выделяет в нем ключевые сущности, события и взаимосвязи, которые затем применяются в качестве фильтрующих параметров для сокращения области векторного поиска. Например, если в запросе упоминается город Париж, то он может использоваться при фильтрации для сужения области поиска.

Методы индексации данных и оптимизации запроса сильно зависят от структуры, источника и типа данных. Как и в случае с любым другим пайплайном

обработки данных, универсального решения не существует: каждая ситуация требует подбора подходящих техник и их адаптации. Оптимизация этапа подготовки запроса к поиску — это эмпирический процесс, требующий экспериментов с разными методами и итеративного улучшения системы.

Этап поиска

Двумя основными способами оптимизации данного этапа являются:

- *улучшение моделей векторизации*, применяемых в пайплайне RAG на этапе кодирования фрагментированных документов и при преобразовании пользовательского ввода на этапе инференса;
- *использование функций фильтрации и поиска, предусмотренных в базе данных*. Данный способ применяется исключительно на этапе инференса, когда необходимо извлечь наиболее релевантные фрагменты на основе пользовательского запроса.

Обе стратегии направлены на достижение общей цели — повышение эффективности векторного поиска за счет семантического сходства между запросом и индексированными данными.

Первая стратегия предполагает адаптацию моделей векторизации к специфике предметной области, особенно предусматривающей узкопрофильную терминологию или редко встречающиеся понятия. Это часто требует тонкой настройки предварительно обученных моделей.

Вместо этого можно задействовать специализированные модели (например, Instructor, https://huggingface.co/hkunlp/instructor-xl), которые управляют процессом векторизации с помощью инструкций или промптов, задающих нужный контекст. Использование такой модели для адаптации сети векторных представлений к вашим данным станет разумной альтернативой, поскольку полноценная тонкая настройка модели требует больших вычислительных ресурсов и участия специалистов.

Ниже приведен пример кода, в котором модель Instructor применяется для векторизации заголовков статей, посвященных ИИ:

```
from InstructorEmbedding import INSTRUCTOR

model = INSTRUCTOR("hkunlp/instructor-base")

sentence = "Основы RAG"

instruction = "Сгенерируй варианты заголовка для статьи на тему ИИ:"

embeddings = model.encode([[instruction, sentence]])
print(embeddings.shape) # noqa
# Ответ: (1, 768)
```

Исходный код доступен по адресу github.com/PacktPublishing/LLM-Engineering.



Для запуска кода модели Instructor необходимо создать отдельное виртуальное окружение и активизировать его:

```
python3 -m venv instructor_venv && source instructor_venv/bin/activate
```

а также установить все необходимые зависимости Python:

```
pip install sentence-transformers==2.2.2 InstructorEmbedding==1.0.1
```

Вторая стратегия использует классические методы фильтрации и поиска, доступные во многих базах данных.

- *Гибридный поиск*. Представляет собой комбинацию векторного поиска и поиска по ключевым словам. Последний позволяет находить документы, содержащие конкретные слова, что особенно важно, если вам нужны высокая точность и буквальное совпадение. Векторный поиск, напротив, может «пропустить» точное совпадение, но отлично справляется с выявлением семантической близости. Комбинируя оба метода, вы получаете возможность контролировать баланс между ними с помощью параметра, который задает вес каждого компонента. При этом два типа поиска выполняются независимо, после чего их результаты нормализуются и объединяются.
- *Векторный поиск с фильтрацией*. Позволяет использовать индекс метаданных для выполнения фильтрации по определенным ключевым словам. В отличие от гибридного поиска в этом случае извлечение данных происходит один раз — только на основе векторного индекса, а фильтрация осуществляется до или после выполнения поиска для сужения его области.

На практике оптимизацию этапа поиска обычно начинают с применения одного из вышеописанных классических методов, поскольку они относительно просто реализуются и позволяют гибко настраивать стратегию, исходя из уровня производительности. Если результаты окажутся недостаточно точными, можно прибегнуть к тонкой настройке модели векторизации.

Этап работы с найденными документами перед генерацией

На этом этапе методы оптимизации применяются исключительно к уже извлеченным данным во избежание снижения качества работы LLM из-за таких факторов, как ограниченное окно контекста или лишние данные. Дело в том, что извлеченный контекст может оказаться слишком объемным или содержать нерелевантную информацию — и то и другое отвлекает модель и снижает точность ее ответов.

На текущем этапе чаще всего применяются два подхода.

- *Сжатие промпта* — устранение лишних деталей при сохранении смыслового ядра данных.
- *Повторное ранжирование* — использование кросс-энкодера, то есть ML-модели, которая вычисляет оценку соответствия между пользовательским запросом и каждым извлеченным фрагментом. Извлеченные элементы сортируются по убыванию этой оценки, после чего отбираются только N наиболее релевантных результатов. Как показано на рис. 4.7, данный подход позволяет учитывать более сложные взаимосвязи между запросом и содержимым по сравнению с обычным поиском на основе сходства векторных представлений. Однако использовать такую модель на этапе первичного извлечения слишком затратно. На практике обычно применяют комбинированную стратегию: на первом этапе данные извлекаются на основе сходства векторных представлений, а затем уточняются с помощью модели повторного ранжирования (рис. 4.8).

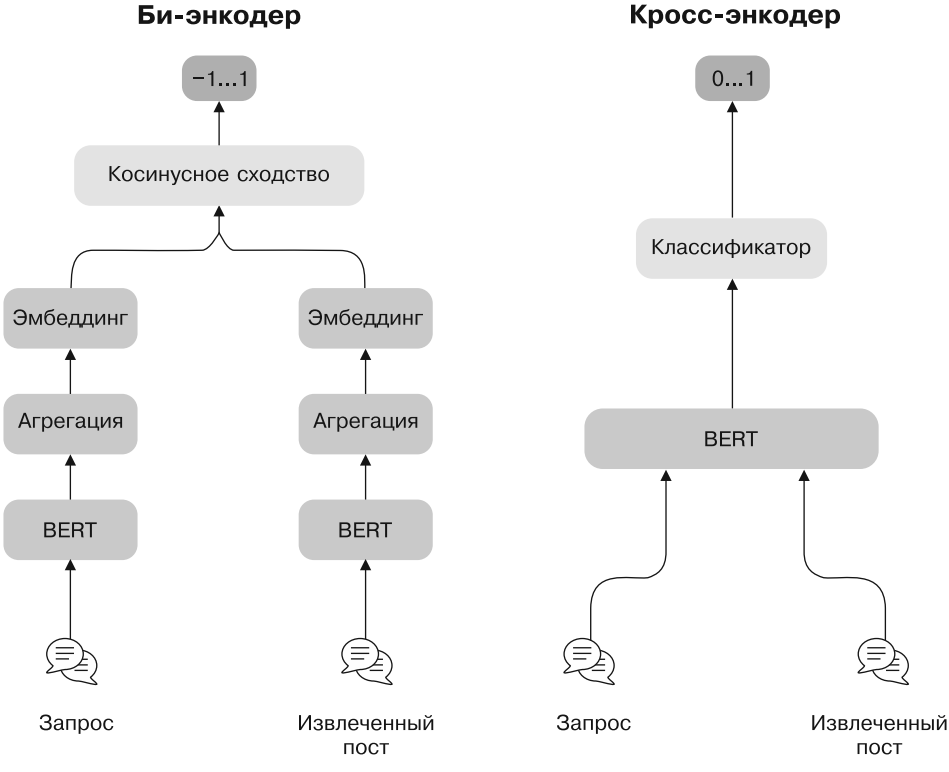


Рис. 4.7. Сравнение би-энкодера (стандартной модели векторизации) и кросс-энкодера

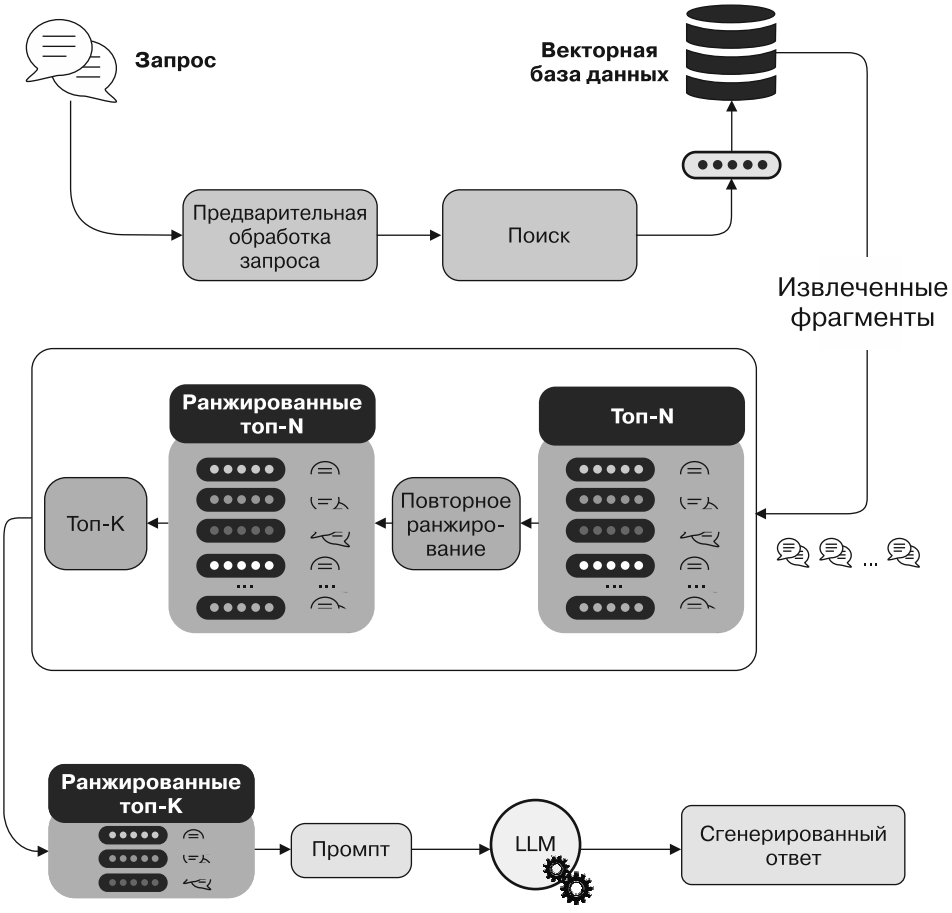


Рис. 4.8. Алгоритм повторного ранжирования

Следует понимать, что представленный выше список техник не является исчерпывающим. Мы привели лишь несколько примеров, чтобы вы получили представление о том, какие этапы работы пайплайна RAG можно (и нужно) оптимизировать. На практике конкретные методы могут существенно различаться в зависимости от типа обрабатываемых данных.

Например, при работе с такими мультимодальными данными, как текст и изображения, большинство из описанных ранее техник окажутся неприменимы, поскольку они ориентированы исключительно на текст.

Основная цель всех этих методов оптимизации — улучшить работу алгоритма RAG на трех ключевых этапах: до, во время и после поиска. К ним относятся предварительная обработка данных для повышения эффективности векторной

индексации, корректировка пользовательских запросов с целью повышения точности поиска, улучшение модели векторизации, использование стандартных операций фильтрации в БД и удаление зашумленных данных. Применение этих подходов позволяет вам эффективно оптимизировать весь рабочий процесс RAG от обработки до извлечения данных.

Архитектура RAG-пайплайна признаков для LLM-двойника

Теперь, когда у вас сформировалось четкое понимание принципа работы механизма RAG, вернемся к построению LLM-двойника и закрепим изученную теорию.

Любая RAG-система состоит из двух независимых компонентов.

- *Пайплайн загрузки данных (ingestion pipeline)* принимает исходные данные, очищает их, разбивает на фрагменты, векторизует и загружает в векторную БД.
- *Пайплайн инференса (inference pipeline)* выполняет запрос к векторной БД для извлечения релевантного контекста, после чего использует LLM для генерации ответа.

В текущей главе мы сосредоточимся на реализации пайплайна загрузки данных, а к разработке пайплайна инференса приступим в главе 9.

Прежде чем продолжить, вспомним, какую задачу мы решаем и откуда берутся исходные данные. Мы строим полноценную ML-систему, в которой все компоненты взаимодействуют через интерфейсы (или контракты), а каждый пайплайн имеет единственную зону ответственности. В нашем случае задача заключается в том, чтобы получить исходные документы, предварительно обработать их и сохранить в векторную базу данных.

Решаемая задача

Как говорилось в предыдущей главе, цель книги — демонстрация процесса построения LLM-двойника, основанного на ML-системе и готового к развертыванию в продакшене. В этой главе мы стремимся спроектировать RAG-пайплайн признаков, способный принимать исходные данные социальных сетей (например, статьи, репозитории кода и посты) из хранилища MongoDB. Текст исходных документов будет очищаться, разбиваться на фрагменты, из него будет построен вектор, а затем он будет загружен в хранилище признаков. Как

было сказано в главе 1, мы реализуем логическое хранилище признаков на базе артефактов ZenML и векторной базы данных Qdrant.

Поскольку мы стремимся создать полностью автоматизированный пайплайн признаков, нам нужно обеспечить синхронизацию между хранилищем данных и логическим хранилищем признаков. Напомним, что на этапе инференса контекст, необходимый для генерации ответа, извлекается из векторной БД. Следовательно, скорость синхронизации между хранилищем данных и хранилищем признаков будет напрямую влиять на точность работы алгоритма RAG.

Еще одной важной задачей является автоматизация работы пайплайна признаков и его интеграция с остальной ML-системой. Наша цель — минимизировать любое рассогласование между двумя хранилищами, поскольку это может поставить под угрозу целостность всей системы.

Таким образом, нам необходимо спроектировать пайплайн признаков, который будет обеспечивать непрерывную синхронизацию между хранилищем данных и логическим хранилищем признаков, обрабатывая содержимое соответствующим образом. Наличие актуальных данных в хранилище признаков критически важно для подготовки ML-системы к развертыванию в продакшене. Пайплайн инференса LLM-двойника будет обращаться к нему для выполнения генерации, дополненной поиском, а пайплайн обучения — получать из него отслеживаемые и версионируемые наборы данных, предназначенные для тонкой настройки.

Хранилище признаков

Хранилище признаков служит центральной *точкой доступа* ко всем признакам, используемым в пайплайнах обучения и инференса. Пайплайн обучения задействует очищенные данные из хранилища признаков (хранящиеся в виде артефактов) для тонкой настройки LLM. Пайплайн инференса обращается к векторной БД за семантически близкими фрагментами документов для выполнения генерации, дополненной поиском. Именно поэтому, помимо RAG-пайплайна загрузки данных, мы проектируем еще пайплайн признаков, который обычно включает в себя несколько подсистем, одной из которых является RAG-логика.

Напомним, что пайплайн признаков используется в первую очередь как схема, помогающая ориентироваться в сложности ML-систем. Он четко определяет, что на вход подаются необработанные данные, а выходом являются признаки (а иногда и метки), которые помещаются в хранилище признаков. Таким образом, вся логика, связывающая хранилище данных с хранилищем признаков

(MongoDB и QDrant), принадлежит пространству имен пайплайна признаков и состоит из одного или нескольких вложенных пайплайнов (sub-pipelines). Например, нам предстоит реализовать еще один пайплайн, который будет принимать очищенные данные, преобразовывать их в наборы данных с инструкциями и сохранять в виде артефактов. Этот пайплайн также входит в зону ответственности пайплайна признаков, поскольку артефакты являются частью их хранилища признаков. Другим примером служит реализация пайплайна валидации данных, работающего с необработанными данными или вычисленными признаками.

Важно также учитывать, что в соответствии со стандартными соглашениями текстовые данные, хранящиеся в виде строк, не считаются признаками. Признак — это то, что подается непосредственно на вход модели. Например, чтобы наборы данных с инструкциями или фрагментированные документы можно было считать признаками, их необходимо токенизировать. Почему? Потому что модель получает на вход не строки, а токены, что делает систему менее гибкой и более сложной. В связи с этим мы будем осуществлять процесс токенизации во время выполнения кода. Это важное наблюдение демонстрирует, что архитектуру FTI не следует воспринимать излишне жестко — ее нужно адаптировать под конкретный случай использования.

Откуда поступают исходные данные

Напомним, что все исходные документы располагаются в хранилище данных MongoDB, которое заполняется с помощью ETL-пайплайна сбора данных, описанного в главе 3. Этот пайплайн сканирует различные платформы, такие как Medium и Substack, стандартизирует данные и загружает их в MongoDB (см. главу 3).

Проектирование архитектуры RAG-пайплайна признаков

На заключительном этапе нам предстоит спроектировать архитектуру RAG-пайплайна признаков для нашего LLM-двойника. Мы воспользуемся пакетной архитектурой, предполагающей регулярное обращение к хранилищу MongoDB, обработку извлеченных данных и их последующую загрузку в векторную БД Qdrant. Первый вопрос, который мы должны себе задать: почему именно пакетный пайплайн?

Прежде чем на него ответить, разберемся в том, как работает пакетная архитектура и чем она отличается от архитектуры, обеспечивающей потоковую обработку данных (рис. 4.9).

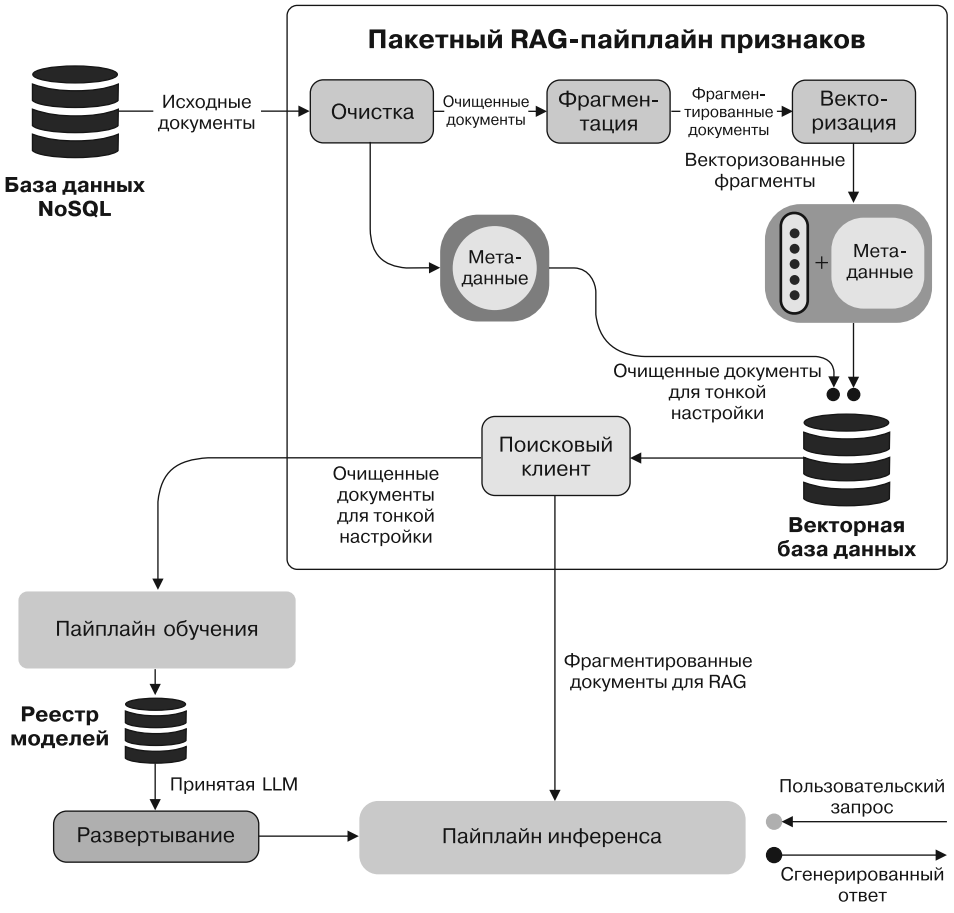


Рис. 4.9. Архитектура RAG-пайплайна признаков для LLM-двойника

Пакетные пайплайны

Пакетный пайплайн в системах обработки данных представляет собой метод, при котором данные собираются, обрабатываются и сохраняются через определенные интервалы времени и большими объемами, называемыми пакетами. Такой подход отличается от потоковой обработки, где данные обрабатываются сразу по мере поступления. Типичный пакетный пайплайн предусматривает следующие этапы.

1. *Сбор данных*. Данные поступают из различных источников и накапливаются до достижения нужного объема. Источниками могут быть базы данных, журналы событий, файлы и не только.

2. *Плановая обработка*. Данные обрабатываются пакетами на регулярной основе, например каждый час или раз в сутки. Обработка может включать очистку, преобразование, агрегацию и другие операции.
3. *Загрузка данных*. После обработки данные загружаются в целевую систему — базу данных, хранилище данных, озеро данных или хранилище признаков и становятся доступны для анализа, запросов или дальнейшей обработки.

Пакетные пайплайны особенно полезны, когда речь идет о больших объемах данных, которые не требуют немедленной обработки. Они обладают рядом преимуществ.

- *Эффективность*. Пакетная обработка позволяет работать с большими объемами данных значительно эффективнее за счет оптимального распределения ресурсов и параллельной обработки.
- *Поддержка сложной логики*. Пакетные пайплайны способны выполнять сложные преобразования и операции агрегации, которые слишком ресурсоемки для потоковой обработки.
- *Простота*. Архитектура пакетного пайплайна, как правило, проще по сравнению с архитектурой потоковой системы, что облегчает ее реализацию и сопровождение.

Пакетные и потоковые пайплайны

При проектировании пайплайнов признаков мы должны сделать выбор между пакетной (batch) и потоковой (streaming) обработкой данных. Рассмотрим различия между этими подходами и обсудим причины, по которым при разработке LLM-двойника была выбрана именно пакетная архитектура.

Потоковые пайплайны заслуживают отдельной главы, что само по себе указывает на их бóльшую сложность по сравнению с пакетными. Однако по мере роста популярности потоковых архитектур важно иметь хотя бы интуитивное представление об их устройстве, чтобы выбрать наиболее подходящий вариант для конкретной задачи.

Основу потоковых приложений составляют два ключевых элемента: распределенная платформа потоковой передачи событий (например, Apache Kafka или Redpanda), которая собирает события от множества клиентов, и движок потоковой обработки, такой как Apache Flink или Bytewax, обрабатывающий эти события в реальном времени. Для упрощения архитектуры вы можете заменить полноценную платформу потоковой передачи событий очередями сообщений (например, RabbitMQ), которые будут временно хранить события до момента их обработки. В табл. 4.1 приведено сравнение пакетных и потоковых пайплайнов по нескольким критериям, включая график обработки и общую сложность архитектуры.

Таблица 4.1. Сравнение пакетных и потоковых пайплайнов

Критерий	Пакетный пайплайн	Потоковый пайплайн
График обработки	Обрабатывает данные с определенной регулярностью (например, каждую минуту, час, день)	Обрабатывает данные непрерывно, с минимальной задержкой
Эффективность	Эффективно работает с большими объемами данных, оптимизируя использование ресурсов и параллельную обработку	Обрабатывает отдельные сообщения, обеспечивая мгновенную реакцию и обновления
Сложность обработки	Поддерживает выполнение сложных преобразований и операций агрегирования данных	Предназначен для обработки быстрых потоков данных с минимальной задержкой
Сценарии использования	Подходит для решения задач, не требующих немедленной обработки. Часто используется в хранилищах данных, системах отчетности, ETL-процессах и пайплайнах признаков	Идеален для приложений, осуществляющих аналитику в реальном времени, требующих актуальных признаков, выполняющих мониторинг и имеющих событийно-ориентированную архитектуру
Сложность системы	По сравнению с потоковыми системами более прост в реализации и сопровождении	По сравнению с пакетными системами более сложен в реализации и сопровождении из-за требований к времени задержки, устойчивости к сбоям и масштабируемости, а также из-за сложности используемого инструментария

Потоковые пайплайны особенно эффективны в рекомендательных системах таких социальных сетей, как TikTok. Поведение пользователя подобных сервисов меняется очень часто. В какой-то момент он хочет просто отдохнуть и посмотреть видео с котиками, но спустя 15 минут ему становится скучно и он переключается на более серьезный контент — образовательные ролики или новости. Это означает, что для поддержания вовлеченности пользователя рекомендательная система должна мгновенно подстраиваться под изменения его интересов. Поскольку такие переключения внимания являются цикличными и непредсказуемыми, мы не можем полагаться на пакетный пайплайн, обновляющий рекомендации каждые 30 минут или час. Теоретически можно было бы запускать пайплайн раз в минуту, но это потребовало бы избыточных вычислений и расходов, поскольку большинство сгенерированных предсказаний не было бы использовано. В то же время реализация потокового пайплайна позволяет обновлять признаки конкретного пользователя в режиме реального

времени и передавать их цепочке моделей, которые формируют новые рекомендации на лету.

Потоковые архитектуры также лежат в основе алгоритмов обнаружения мошенничества, применяемых, например, в системах Stripe или PayPal. Потенциально мошеннические транзакции крайне важно выявлять в момент их совершения, а не спустя несколько минут или часов, как это происходило бы при пакетной обработке. Та же логика действует и для платформ высокочастотной торговли, где изменения цены акций прогнозируются на основе непрерывного потока рыночных данных, что позволяет трейдерам принимать решения за доли секунды.

С другой стороны, пакетная архитектура хорошо подходит для рекомендательных офлайн-систем. Например, при построении рекомендательной системы для маркетплейсов или видеостримингового сервиса высокая реактивность не требуется, поскольку поведение пользователя меняется относительно редко. В таких случаях обновлять рекомендации раз в сутки на основе исторических данных проще и экономичнее.

Еще одним типичным примером использования пакетной архитектуры являются ETL-пайплайны, предназначенные для извлечения, преобразования и загрузки данных. Такой подход широко применяется для переноса данных между различными БД, а также для их агрегации с целью дальнейшего анализа, когда необходимо извлечь информацию из нескольких источников, объединить ее и загрузить в хранилище, из которого читает система анализа данных. Это актуально в самых разных областях: от электронной коммерции и маркетинга до финансов и научных исследований.

Пайплайн сбора данных, используемый в нашей системе LLM-двойника, представляет собой еще один пример ETL-пайплайна, который извлекает данные из Интернета, структурирует их и загружает в хранилище для последующей обработки.

Помимо «свежести» предсказаний и признаков, недостатком пакетных пайплайнов, по сравнению с потоковыми, является избыточность предсказаний. Возьмем, к примеру, рекомендательную систему для стримингового платформы вроде Netflix. Каждую ночь система генерирует предсказания для всех пользователей. Однако значительная часть из них может вовсе не войти в систему на следующий день. Более того, пользователи, как правило, не просматривают все рекомендации, а выбирают из нескольких первых. В итоге большая часть предсказаний остается невостребованной, что приводит к напрасной трате вычислительных ресурсов.

Именно поэтому имеет смысл начать с пакетной архитектуры, которую проще и быстрее реализовать, а после вывода продукта на рынок — постепенно переходить к потоковой архитектуре, чтобы снизить затраты и улучшить пользовательский опыт.

Итак, для реализации пайплайна признаков LLM-двойника мы выбрали пакетную (а не потоковую) архитектуру по следующим причинам.

- *Отсутствие необходимости в немедленной обработке данных*. Хотя синхронизация хранилищ данных и признаков критически важна для обеспечения точности работы RAG-системы, небольшая задержка (в несколько минут) вполне допустима. Поэтому мы можем запускать пакетный пайплайн каждую минуту, обеспечивая постоянную синхронизацию между двумя хранилищами. Эта стратегия оказывается эффективной благодаря сравнительно небольшому объему данных: поскольку в хранилище содержатся тысячи записей, а не миллионы или миллиарды, их можно быстро обработать и синхронизировать обе БД.
- *Простота*. Как уже отмечалось, реализовать потоковый пайплайн значительно сложнее. В реальных условиях важно сохранять систему максимально простой, так как это облегчает ее понимание, отладку и сопровождение. Кроме того, простота почти всегда означает снижение затрат на создание инфраструктуры и разработку.

На рис. 4.10 показано, какие инструменты целесообразно использовать в зависимости от архитектуры (пакетная или потоковая) и объема обрабатываемых данных (маленький или большой). Поскольку в нашем случае речь идет о пакетной обработке небольшого объема данных, мы выбрали комбинацию, состоящую из языка Python и таких инструментов генеративного ИИ, как LangChain, `SentenceTransformer` и Unstructured.

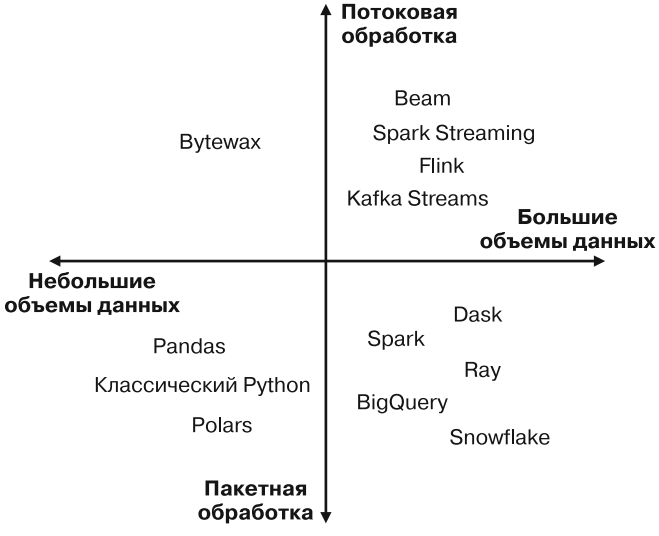


Рис. 4.10. Выбор инструментов в зависимости от архитектуры и объема обрабатываемых данных

В пункте «Отслеживание изменений в данных: синхронизация между хранилищами данных и признаков» мы поговорим о том, в каких случаях имеет смысл переключаться с пакетной архитектуры на потоковую.

Основные этапы

Большинство RAG-пайплайнов признаков включают пять основных этапов. Пайплайн LLM-двойника не является исключением. Этапы его работы перечислены ниже (вы можете легко адаптировать эту схему для других RAG-приложений).

1. *Извлечение данных*. В нашем случае необходимо извлечь последние статьи, код из репозиториев и посты из хранилища данных MongoDB. На этом этапе, как правило, агрегируются все данные, которые понадобятся для дальнейшей обработки.
2. *Очистка*. Хотя данные в хранилище уже частично стандартизированы и очищены, нам необходимо убедиться в том, что текст содержит только полезную информацию, не дублируется и может быть корректно интерпретирован моделью векторизации. Например, перед передачей текста в модель нужно очистить и нормализовать все символы, не входящие в набор ASCII. Кроме того, чтобы сохранить высокую семантическую плотность текста, мы решили заменить все URL-адреса на заполнители и удалить все эмодзи. Очистка текста — это больше искусство, чем *наука*. Поэтому после первой итерации и внедрения механизма оценки вы, скорее всего, будете неоднократно возвращаться к этому этапу для улучшения его результатов.
3. *Фрагментация*. Стратегию фрагментации необходимо выбирать с учетом типа данных и используемой модели векторизации. Например, при работе с репозиториями кода предпочтительны более крупные фрагменты, а со статьями — более мелкие, например абзацы. В зависимости от данных документ можно разбивать по главам, разделам, абзацам, предложениям или блокам, умещающимся в окно фиксированного размера. При этом размер фрагмента не должен превышать максимально допустимый размер входных данных модели векторизации. Именно поэтому выбор метода фрагментации обычно зависит от структуры данных и ограничений модели.
4. *Векторизация*. Каждый фрагмент передается в выбранную модель векторизации. С точки зрения реализации этот этап обычно является самым простым, поскольку инструменты вроде `SentenceTransformer` и Hugging Face предоставляют высокоуровневые интерфейсы для взаимодействия со многими моделями. Как объяснялось в подразделе «Что такое эмбеддинги», на этом этапе необходимо выбрать модель и оценить целесообразность ее тонкой настройки. Например, мы использовали модель `all-mpnet-base-v2` из `SentenceTransformer`, которая является довольно компактной и запускается

на большинстве машин. Вместе с тем мы предоставили конфигурационный файл, в котором вы при желании можете легко заменить модель на более производительную. Альтернативные модели представлены в рейтинге MTEB на сайте Hugging Face: https://huggingface.co/spaces/mteb/leaderboard.

5. *Загрузка данных*. На последнем этапе векторное представление каждого фрагмента объединяется с такими метаданными, как имя автора, идентификатор документа, содержимое, URL-адрес, платформа и дата создания. Затем вектор и метаданные упаковываются в структуру, совместимую с системой Qdrant, и загружаются в векторную БД. Поскольку мы хотим использовать Qdrant в качестве единственного источника истины для признаков, туда также помещаются и очищенные (нефрагментированные) документы. При этом данные можно загрузить и без векторов, так как индекс метаданных Qdrant ведет себя как база данных NoSQL. Таким образом, сохранение метаданных без соответствующих векторов эквивалентно использованию стандартной NoSQL-системы.

Отслеживание изменений в данных: синхронизация между хранилищами данных и признаков

Как уже неоднократно подчеркивалось, данные постоянно изменяются, что может привести к рассогласованию баз данных, озер данных, хранилищ данных и хранилищ признаков. Стратегия *CDC* (Change Data Capture — «захват изменений данных») позволяет обеспечивать синхронизацию между двумя и более типами хранилищ данных без лишних издержек, связанных с вычислениями и операциями ввода/вывода. Она предполагает отслеживание операций CRUD, выполняемых в исходной БД, и их репликацию в целевом хранилище. При необходимости между этими действиями можно вставить этапы предварительной обработки данных.

Проблемы синхронизации возникают и при построении пайплайна признаков. Поэтому одним из ключевых проектных решений является выбор стратегии обеспечения синхронизации между хранилищем данных и хранилищем признаков, чтобы данные оставались актуальными для решения конкретной задачи.

В случае с LLM-двойником мы выбрали самый простой подход и реализовали пакетный пайплайн, который запускается периодически или вручную. Он считывает все сырые данные из хранилища, обрабатывает их пакетами и вставляет новые записи или обновляет существующие в векторной БД Qdrant. Такой подход работает вполне приемлемо при небольших объемах данных (тысячи или десятки тысяч записей), однако он поднимает следующие вопросы.

- Что произойдет, если объем данных внезапно возрастет до миллионов записей (или больше)?

- Что, если какая-то из записей будет удалена из хранилища данных? Как это отразится на хранилище признаков?
- А если мы хотим обрабатывать только новые или обновленные элементы, а не все сразу?

К счастью, паттерн CDC позволяет решить все эти проблемы. При его реализации можно выбрать один из двух подходов: push или pull.

- *Push* (активная отправка). Здесь основную роль играет исходная БД. Она самостоятельно отслеживает изменения и немедленно передает их в целевые системы. Такой метод обеспечивает почти мгновенное обновление на стороне получателя, однако, если целевая система в этот момент окажется недоступна, данные могут быть потеряны. Чтобы избежать подобных инцидентов, в архитектуру обычно добавляют систему передачи сообщений, которая играет роль буфера.
- *Pull* (периодическое извлечение). В рамках этого подхода исходная БД играет более пассивную роль, которая ограничивается регистрацией изменений. Целевая система периодически запрашивает эти изменения и применяет их у себя. Такой подход снижает нагрузку на источник данных, но привносит задержку в процесс распространения обновлений. Здесь система сообщений тоже необходима для предотвращения потерь данных в случае временной недоступности целевой системы.

Метод push идеально подходит для приложений, требующих немедленного доступа к обновленным данным, тогда как метод pull лучше использовать для крупномасштабной передачи данных в тех случаях, когда обновление в реальном времени не является критически важным. Существует несколько способов отслеживания изменений в данных. Ниже приведены основные CDC-паттерны, используемые в индустрии.

- *На основе временных меток (timestamp-based)*. Этот подход предполагает добавление в таблицы базы данных столбца с временем последнего изменения, например `LAST_MODIFIED` или `LAST_UPDATED`. Подсистемы могут обращаться к этому столбцу, чтобы выявить записи, которые были обновлены с момента последней проверки. Такой метод легко реализуется, но не позволяет отслеживать удаленные записи и создает дополнительную нагрузку из-за необходимости сканировать всю таблицу.
- *На основе триггеров (trigger-based)*. Этот подход предполагает использование триггеров базы данных для автоматической фиксации результатов изменения данных после выполнения операций `INSERT`, `UPDATE` и `DELETE` в отдельной таблице событий. Такой метод позволяет отслеживать все изменения, но может негативно повлиять на производительность базы,

поскольку требует выполнения дополнительных операций записи при наступлении каждого события.

- *На основе журналов транзакций (log-based)*. Современные СУБД ведут журналы транзакций, фиксирующие все изменения, включая время их внесения. Хотя эти журналы предназначены для восстановления данных, их можно использовать и для передачи изменений в целевые системы в режиме реального времени. Такой подход минимизирует нагрузку на исходную БД, фиксирует все преобразования и не требует модификации схемы. Однако у него есть один существенный недостаток в виде отсутствия единого стандарта формата журналов, что приводит к зависимости от конкретной реализации СУБД.



Если вы хотите узнать больше о технологии CDC, рекомендуем прочитать статью What is Change Data Capture?, опубликованную в блоге Confluent: https://www.confluent.io/en-gb/learn/change-data-capture/.

Учитывая описанные подходы CDC, мы можем довольно быстро реализовать pull-стратегию на основе временных меток в рамках нашего RAG-пайплайна признаков. Это позволит более эффективно синхронизировать хранилища данных и признаков по мере роста объема данных. Однако вместо обращения к полю последнего обновления в исходной базе наша реализация будет извлекать из хранилища все содержимое.

На практике наиболее популярным и оптимальным методом считается подход на основе журналов транзакций. Он не создает дополнительной нагрузки на исходную СУБД, минимизирует задержку и поддерживает все операции CRUD. Его главный недостаток — сложность, так как он требует использования очереди сообщений для фиксации всех событий CRUD и потокового пайплайна для их обработки.

Поскольку книга посвящена LLM, а не инженерии данных, мы стремились использовать как можно более простые решения. Тем не менее знание о существовании этих техник поможет вам улучшить текущую реализацию, если она перестанет удовлетворять требованиям вашей системы.

Почему мы сохраняем два снимка данных

В логическом хранилище признаков мы сохраняем два снимка (snapshots) данных:

- *после очистки* — для тонкой настройки LLM;
- *после фрагментации и векторизации документов* — для RAG.

Почему мы спроектировали систему именно таким образом? Как вы помните, доступ к признакам как во время обучения, так и во время инференса должен осуществляться исключительно через хранилище признаков. Выбранный нами подход обеспечивает согласованность архитектуры и делает ее более упорядоченной.

Кроме того, сохранение данных, специально очищенных для тонкой настройки модели и векторизации, в хранилище MongoDB было бы примером антипаттерна. Дело в том, что данные из этого хранилища используются по всей компании, следовательно, их обработка под конкретную задачу может привести к проблемам. Представьте, что у нас появляется другой сценарий использования, например резюмирование, для которого требуется иная стратегия очистки и предварительной обработки. В этом случае нам пришлось бы создавать отдельную таблицу с очищенными данными, в названии которой отражался бы соответствующий сценарий. И так каждый раз. Чтобы структура централизованного хранилища не стала хаотичной, данные в нем должны оставаться максимально близкими к исходным, а их адаптация под конкретное применение должна происходить в низлежащих компонентах — в нашем случае в хранилище признаков.

Как уже отмечалось в пункте «Основные этапы», мы можем использовать индекс метаданных векторной БД в качестве базы данных NoSQL. С учетом этого было принято решение хранить в Qdrant как очищенные документы, так и их фрагментированные и векторизованные версии.

При переходе к эксплуатации системы LLM-двойника пайплайн создания набора данных с инструкциями (будет описан в главе 5) будет считывать очищенные документы из Qdrant, обрабатывать их и сохранять в виде версионируемого артефакта ZenML. Пайплайну обучения нужен именно набор данных, а не просто текстовые документы. Это еще раз подчеркивает тот факт, что наше хранилище признаков состоит из векторной базы данных Qdrant (применяемой для онлайн-инференса) и артефактов ZenML (используемых для офлайн-обучения).

Оркестрация

Инструмент ZenML будет осуществлять оркестрацию пакетного RAG-пайплайна признаков. С его помощью можно запускать пайплайн по расписанию (например, каждый час) или вручную, а также сразу после завершения работы ETL-пайплайна сбора данных.

Интеграция пайплайна признаков с системой оркестрации (будь то ZenML или другой инструмент) позволяет перевести его в эксплуатационный режим для осуществления *непрерывного обучения* (CT).

Все детали оркестрации, составления расписаний и непрерывного обучения будут подробно рассмотрены в главе 11.

Реализация RAG-пайплайна признаков для LLM-двойника

Реализуем RAG-пайплайн признаков для LLM-двойника, чтобы применить на практике изученную теорию. В этом разделе мы подробно обсудим:

- код ZenML;
- доменные сущности Pydantic;
- пользовательскую реализацию паттерна *объектно-векторного отображения* (Object-Vector Mapping, OVM);
- логику очистки, фрагментации и векторизации для всех категорий данных.

Мы будем придерживаться подхода «сверху вниз», поэтому начнем с класса `Settings` и ZenML-пайплайна.

Класс Settings

Мы применим механизм Pydantic Settings (https://docs.pydantic.dev/latest/concepts/pydantic_settings/) для определения глобального класса `Settings`, который загружает чувствительные и нечувствительные переменные из файла `.env`. Этот подход позволяет воспользоваться всеми преимуществами Pydantic, включая проверку типов. Например, если вместо целого числа в качестве значения переменной `QDRANT_DATABASE_PORT` будет передана строка, программа завершится с ошибкой. Такое поведение делает работу приложения более предсказуемой и надежной.

Ниже приведен пример класса `Settings` с перечнем переменных, необходимых для построения RAG-пайплайна признаков:

```
from pydantic import BaseSettings

class Settings(BaseSettings):
    class Config:
        env_file = ".env"
        env_file_encoding = "utf-8"

    ... # Другие настройки...

    # RAG
    TEXT_EMBEDDING_MODEL_ID: str = "sentence-transformers/all-MiniLML6-v2"
    RERANKING_CROSS_ENCODER_MODEL_ID: str = "cross-encoder/ms-marco-MiniLM-
    ➥L-4-v2"
    RAG_MODEL_DEVICE: str = "cpu"


    # Векторная БД QdrantDB
    USE_QDRANT_CLOUD: bool = False
```

```
    QDRANT_DATABASE_HOST: str = "localhost"
    QDRANT_DATABASE_PORT: int = 6333
    QDRANT_CLOUD_URL: str = "str"
    QDRANT_APIKEY: str | None = None

    ... # Дополнительные настройки...

settings = Settings()
```

Как указано во вложенном классе `Config`, все переменные имеют значения по умолчанию или могут быть переопределены с помощью файла `.env`.

Пайплайн и шаги ZenML

Пайплайн ZenML служит точкой входа для RAG-пайплайна генерации признаков и отражает пять основных этапов работы RAG-пайплайна загрузки данных: извлечение исходных документов, очистку, фрагментацию, векторизацию и загрузку эмбеддингов в логическое хранилище признаков. Вызовы внутри функции `feature_engineering()` представляют собой ZenML-шаги, каждый из которых реализует один из пяти этапов RAG-пайплайна. Исходный код этого пайплайна доступен в файле репозитория GitHub — https://github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/pipelines/feature_engineering.py:

```
from zenml import pipeline

from llm_engineering.interfaces.orchestrator.steps import
➥feature_engineering as fe_steps

@pipeline
def feature_engineering(author_full_names: list[str]) -> None:
    raw_documents = fe_steps.query_data_warehouse(author_full_names)

    cleaned_documents = fe_steps.clean_documents(raw_documents)
    last_step_1 = fe_steps.load_to_vector_db(cleaned_documents)

    embedded_documents = fe_steps.chunk_and_embed(cleaned_documents)
    last_step_2 = fe_steps.load_to_vector_db(embedded_documents)

    return [last_step_1.invocation_id, last_step_2.invocation_id]
```

На рис. 4.11 показано, как выглядят запуски пайплайна генерации признаков на дашборде ZenML.

На рис. 4.12 шаги RAG-пайплайна признаков и соответствующие выходные артефакты представлены в виде ориентированного ациклического графа (DAG). Как уже было сказано, результат выполнения каждого ZenML-шага автоматически сохраняется в виде артефакта, регистрируется в реестре артефактов ZenML, версионируется и становится доступным для повторного использования в других частях приложения.



Рис. 4.11. Запуски пайплайна признаков на дашборде ZenML

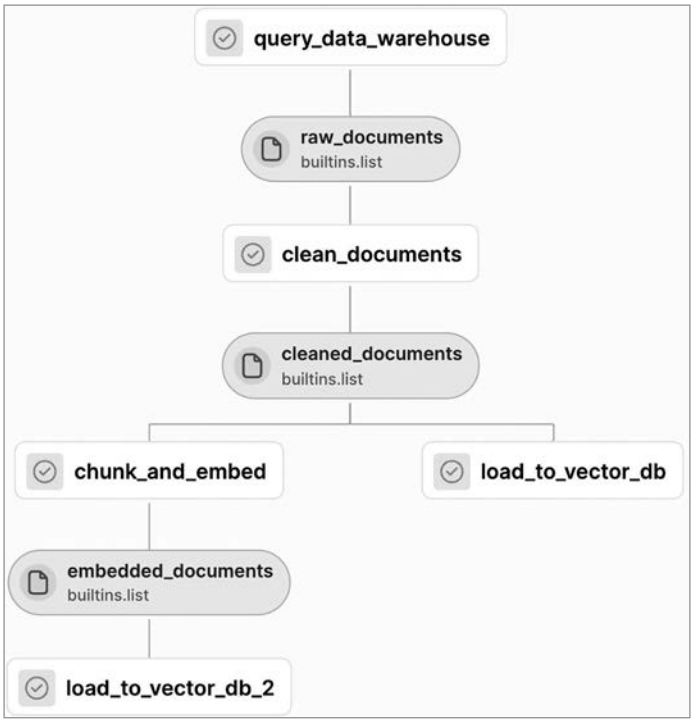


Рис. 4.12. DAG-представление пайплайна признаков на дашборде ZenML

Теперь разберемся с динамической настройкой RAG-пайплайна признаков. Все его настройки доступны через параметры функции. В данном случае нам требуется только список имен авторов, что видно из сигнатуры функции: `feature_engineering(author_full_names: list[str])`. Во время выполнения мы подгружаем YAML-файл конфигурации, содержащий все необходимые значения в зависимости от конкретного сценария использования. Показанный ниже

пример конфигурации, включающий список всех авторов книги, можно найти в файле репозитория GitHub: `configs/feature_engineering.yaml`:

```
parameters:
    author_full_names:
        - Alex Vesa
        - Maxime Labonne
        - Paul Iusztin
```

Главное преимущество этого подхода состоит в том, что вам не нужно менять код, чтобы запускать пайплайн с разными входными значениями. Достаточно указать при запуске другой конфигурационный файл:

```
feature_engineering.with_options(config_path=".../feature_engineering.yaml")
()
```

Вы можете либо жестко прописать путь к файлу конфигурации в коде, либо передать переменную `config_path` через интерфейс командной строки (CLI), позволяющий легко изменять конфигурацию пайплайна от запуска к запуску. Чтобы не усложнять, мы прописали путь к файлу конфигурации в коде для запуска пайплайна с помощью скрипта `run.py`:

```
python -m tools.run --no-cache --run-feature-engineering
```

Кроме того, можно добавить еще один аргумент командной строки для передачи переменной `config_path` либо использовать для запуска пайплайна признаков следующую команду `poe`:

```
poetry poe run-feature-engineering-pipeline
```

Теперь подробно рассмотрим каждый из ZenML-шагов. Исходный код всех шагов пайплайна генерации признаков находится в папке репозитория GitHub: `steps/feature_engineering`. Начнем с обращения к хранилищу данных для извлечения нового контента, подлежащего преобразованию в признаки.

Выполнение запроса к хранилищу данных

Первым делом следует обратить внимание на то, что ZenML-шаг представляет собой обычную функцию Python, обернутую декоратором `@step`, как и сам пайплайн ZenML. Ниже показана функция, которая принимает на вход список полных имен авторов и выполняет следующие действия.

- Пытается получить или создать экземпляр `UserDocument`, используя имя и фамилию, после чего добавляет его в список авторов (`authors`).
- Извлекает все исходные документы этого пользователя из хранилища данных и помещает их в список документов (`documents`).

- Формирует словарь с описательными метаданными, которые записываются в журнал и отслеживаются системой ZenML.

```
... # другие операторы импорта
from zenml import get_step_context, step

@step
def query_data_warehouse(
    author_full_names: list[str],
    ) -> Annotated[list, "raw_documents"]:
    documents = []
    authors = []
    for author_full_name in author_full_names:
        logger.info(f"Querying data warehouse for user: {author_full_name}")

        first_name, last_name = utils.split_user_full_name(author_full_name)
        logger.info(f"First name: {first_name}, Last name: {last_name}")
        user = UserDocument.get_or_create(first_name=first_name,
        ➥last_name=last_name)
        authors.append(user)

        results = fetch_all_data(user)
        user_documents = [doc for query_result in results.values() for doc
        ➥in query_result]

        documents.extend(user_documents)

    step_context = get_step_context()
    step_context.add_output_metadata(output_name="raw_documents",
    ➥metadata=_get_metadata(documents))

    return documents
```

Функция извлечения данных использует пул потоков, где каждый запрос выполняется в отдельном потоке. Поскольку мы имеем дело с несколькими категориями данных, нам необходимо выполнять отдельные запросы к коллекциям статей, постов и репозиториев. Скорость выполнения запросов к хранилищу ограничена сетевыми задержками и временем отклика БД, а не уровнем загрузки CPU. Поэтому, распределив запросы по разным потокам, мы сможем выполнять их параллельно. В итоге время выполнения функции извлечения данных будет определяться не суммой всех задержек, а максимальным значением среди них.

Использование потоков для параллельного выполнения операций, ограниченных скоростью ввода/вывода, считается хорошей практикой в Python, поскольку такие операции не блокируются *механизмом GIL* (Global Interpreter Lock — «глобальная блокировка интерпретатора»). В свою очередь, запуск каждого вызова в отдельном процессе приводит к значительным издержкам, так как запуск процесса занимает больше времени, чем запуск потока.

В Python процессы целесообразно использовать для распараллеливания только тех операций, скорость которых ограничена ресурсами процессора или объемом доступной памяти, поскольку на них механизм GIL действительно оказывает влияние. Каждый процесс имеет собственный GIL, поэтому параллельное выполнение таких вычислений, как обработка пакета документов или изображений, уже загруженных в память, не подвержено ограничениям, связанным с глобальной блокировкой интерпретатора Python.

```
def fetch_all_data(user: UserDocument) -> dict[str, list[NoSQLBaseDocument]]:
    user_id = str(user.id)
    with ThreadPoolExecutor() as executor:
        future_to_query = {
            executor.submit(__fetch_articles, user_id): "articles",
            executor.submit(__fetch_posts, user_id): "posts",
            executor.submit(__fetch_repositories, user_id): "repositories",
        }

        results = {}
        for future in as_completed(future_to_query):
            query_name = future_to_query[future]
            try:
                results[query_name] = future.result()
            except Exception:
                logger.exception(f"'{query_name}' request failed.")

                results[query_name] = []

    return results
```

Функция `_get_metadata()` получает на вход список извлеченных документов и список авторов, после чего подсчитывает их количество для каждого типа данных:

```
def _get_metadata(documents: list[Document]) -> dict:
    metadata = {
        "num_documents": len(documents),
    }
    for document in documents:
        collection = document.get_collection_name()
        if collection not in metadata:
            metadata[collection] = {}
        if "authors" not in metadata[collection]:
            metadata[collection]["authors"] = list()

        metadata[collection]["num_documents"] = metadata[collection].
        ➥get("num_documents", 0) + 1
        metadata[collection]["authors"].append(document.author_full_name)

    for value in metadata.values():
        if isinstance(value, dict) and "authors" in value:
            value["authors"] = list(set(value["authors"]))

    return metadata
```

Мы публикуем эти метаданные на дашборде ZenML, чтобы оперативно отслеживать статистику по загруженным данным. Например, на рис. 4.13 показана вкладка Metadata (Метаданные) шага `query_data_warehouse()`, где видно, что в рамках конкретного запуска пайплайна признаков были загружены 76 документов, созданных тремя авторами. Такая визуализация является чрезвычайно полезной для мониторинга и отладки пакетных пайплайнов.

Вы всегда можете расширить состав метаданных, добавив в них любую информацию, актуальную для вашего сценария использования.

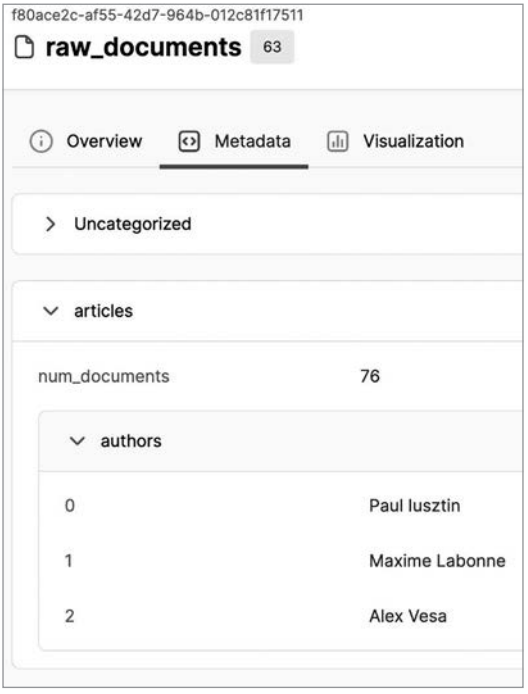


Рис. 4.13. Метаданные ZenML-шага, связанного с обращением к хранилищу данных

Очистка документов

На этапе очистки мы перебираем документы и делегируем всю логику диспетчеру `CleaningDispatcher`, который знает, какую стратегию очистки применить в зависимости от типа данных. Как вы помните, мы собираемся применить разные техники очистки к статьям, постам и репозиториям кода.

```
@step
def clean_documents(
    documents: Annotated[list, "raw_documents"],
```

```
    ) -> Annotated[list, "cleaned_documents"]:
    cleaned_documents = []
    for document in documents:
        cleaned_document = CleaningDispatcher.dispatch(document)
        cleaned_documents.append(cleaned_document)

    step_context = get_step_context()
    step_context.add_output_metadata(output_name="cleaned_documents",
    ➥metadata=_get_metadata(cleaned_documents))

    return cleaned_documents
```

Поскольку метаданные, вычисляемые на этом этапе, аналогичны тем, которые мы записывали в журнал на шаге `query_data_warehouse()`, то сразу перейдем к следующему этапу — фрагментации и векторизации.

Фрагментация и векторизация очищенных документов

Как и на этапе очистки, логика фрагментации и векторизации делегируется диспетчеру, который знает, как обрабатывать каждый тип данных. Обратите внимание: диспетчер фрагментации возвращает не один объект, а список, что логично, поскольку документ разбивается на несколько фрагментов. Мы подробно рассмотрим принцип работы этого диспетчера в подразделе «Слой диспетчеризации».

```
@step
def chunk_and_embed(
    cleaned_documents: Annotated[list, "cleaned_documents"],
) -> Annotated[list, "embedded_documents"]:
    metadata = {"chunking": {}, "embedding": {}, "num_documents":
    ➥len(cleaned_documents)}

    embedded_chunks = []
    for document in cleaned_documents:
        chunks = ChunkingDispatcher.dispatch(document)
        metadata["chunking"] = _add_chunks_metadata(chunks,
        ➥metadata["chunking"])

    for batched_chunks in utils.misc.batch(chunks, 10):
        batched_embedded_chunks = EmbeddingDispatcher.
        ➥dispatch(batched_chunks)
        embedded_chunks.extend(batched_embedded_chunks)

    metadata["embedding"] = _add_embeddings_metadata(embedded_chunks,
    ➥metadata["embedding"])
    metadata["num_chunks"] = len(embedded_chunks)
    metadata["num_embedded_chunks"] = len(embedded_chunks)

    step_context = get_step_context()
    step_context.add_output_metadata(output_name="embedded_documents",
    ➥metadata=metadata)

    return embedded_chunks
```

На рис. 4.14 представлены метаданные ZenML-шага, связанного с фрагментацией и векторизацией. В них можно увидеть, что 76 документов были преобразованы в 2373 фрагмента, а также ознакомиться со свойствами, использованными при фрагментации статей, такими как `chunk_size` = `500` и `chunk_overlap` = `50`.

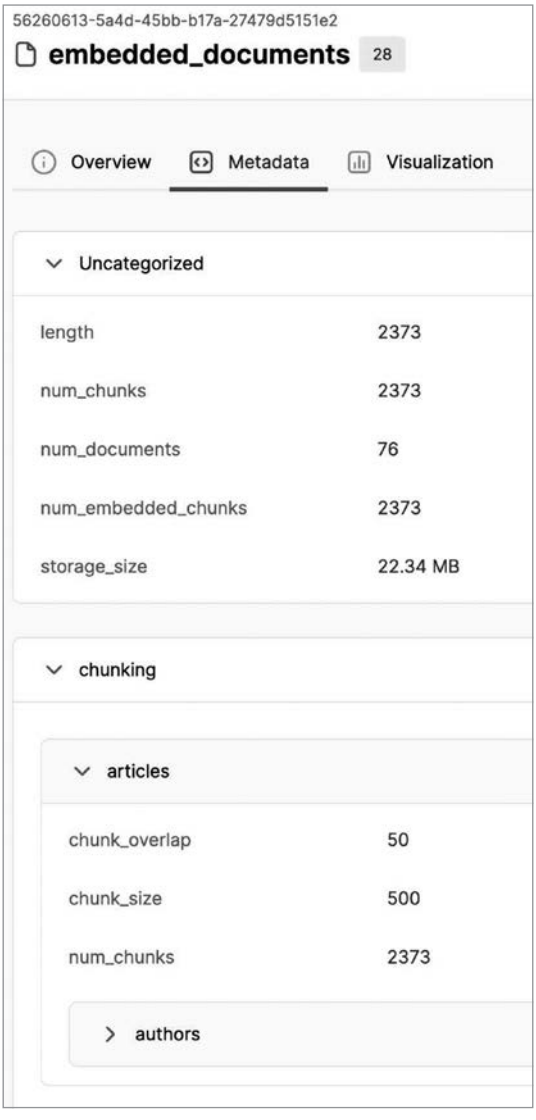


Рис. 4.14. Метаданные ZenML-шага, связанного с фрагментацией и векторизацией, содержащиеся в разделах uncategorized и chunking

На рис. 4.15 представлены остальные метаданные этого шага, в частности модель векторизации и ее параметры для вычисления векторов.

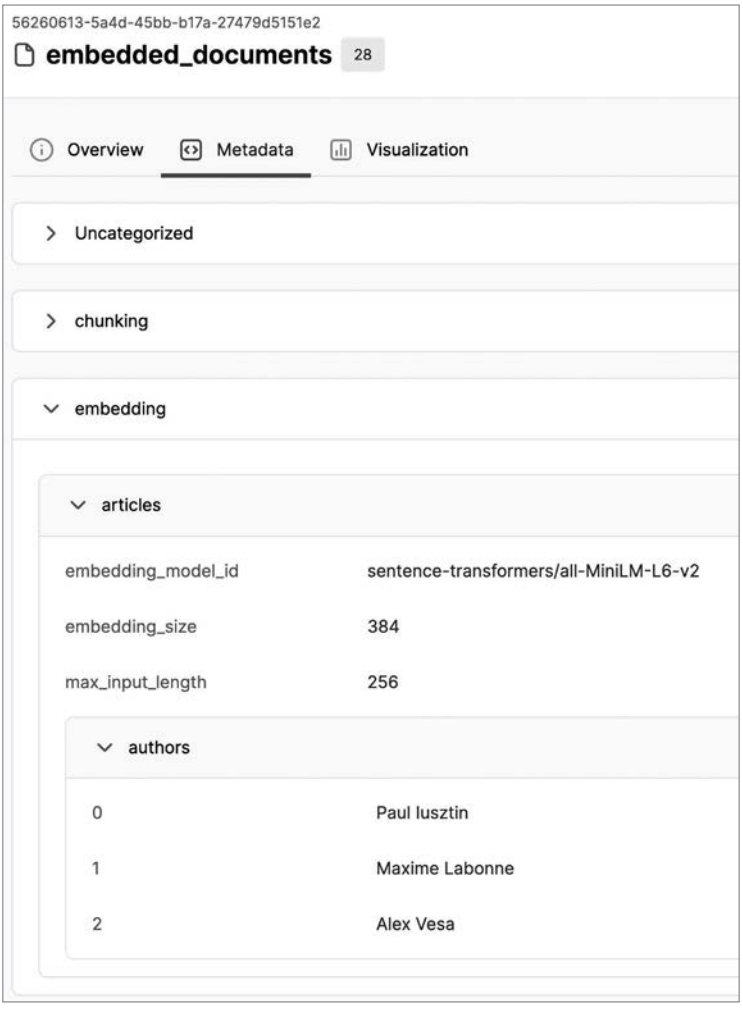


Рис. 4.15. Метаданные ZenML-шага, связанного с фрагментацией и векторизацией, содержащиеся в разделе embedding

Поскольку ML-системы, находящиеся в эксплуатации, могут давать сбои из-за дрейфа данных или непредвиденных обстоятельств, использование метаданных для мониторинга обрабатываемой информации является мощным инструментом, позволяющим существенно сократить время отладки и сэкономить десятки тысяч долларов (если не больше).

Загрузка документов в векторную базу данных

Поскольку статьи, посты и репозитории кода сохраняются в разные коллекции внутри векторной БД, нам необходимо сначала сгруппировать документы по категориям, а затем загрузить каждую группу в векторную БД Qdrant:

```
@step
def load_to_vector_db(
    documents: Annotated[list, "documents"],
) -> None:
    logger.info(f"Loading {len(documents)} documents into the vector database.")

    grouped_documents = VectorBaseDocument.group_by_class(documents)
    for document_class, documents in grouped_documents.items():
        logger.info(f"Loading documents into
        ➥{document_class.get_collection_name()}")
        for documents_batch in utils.misc.batch(documents, size=4):
            try:
                document_class.bulk_insert(documents_batch)
            except Exception:
                return False

    return True
```

Доменные сущности Pydantic

Важно понять, с какими доменными сущностями (моделями предметной области) мы работаем. Реализация LLM-двойника частично основана на принципах *предметно-ориентированного проектирования* (Domain-Driven Design, DDD), в рамках которого ядром приложения являются доменные сущности. Поэтому перед тем, как двигаться дальше, необходимо разобраться в иерархии используемых нами классов.



Исходный код доменных сущностей доступен в репозитории GitHub: https://github.com/PacktPublishing/LLM-Engineering/tree/main/llm_engineering/domain.

Для моделирования доменных сущностей мы использовали Pydantic — пакет Python для написания структур данных со встроенной проверкой типов. Поскольку Python является языком с динамической типизацией, использование Pydantic для валидации типов во время выполнения значительно повышает надежность системы, гарантируя работу с данными нужного типа на всех этапах.

Предметная область нашего LLM-двойника определяется двумя параметрами:

- *тип исходных данных* (пост, статья, репозиторий);
- *состояние данных* (очищенные, фрагментированные, векторизованные).

Мы решили создать базовый класс для каждого состояния документа, в результате чего получили следующие абстрактные базовые классы:

- `class CleanedDocument(VectorBaseDocument, ABC);`
- `class Chunk(VectorBaseDocument, ABC);`
- `class EmbeddedChunk(VectorBaseDocument, ABC).`

Обратите внимание: все эти классы наследуют от `VectorBaseDocument` — нашей пользовательской реализации паттерна *объектно-векторного отображения* (Object-Vector Mapping, OVM), которую мы рассмотрим в следующем пункте. Кроме того, эти классы наследуются от `ABC`, что делает их абстрактными. Таким образом, создать объект на основе этих классов напрямую невозможно — их можно только наследовать. Именно поэтому базовые классы всегда объявляются как абстрактные.

Все эти базовые абстрактные классы, описывающие состояние, имеют подклассы, соответствующие категории данных. Например, класс `CleanedDocument` включает следующие подклассы:

- `class CleanedPostDocument(CleanedDocument);`
- `class CleanedArticleDocument(CleanedDocument);`
- `class CleanedRepositoryDocument(CleanedDocument).`

Как показано на рис. 4.16, аналогичная логика применяется к базовым абстрактным классам `Chunk` и `EmbeddedChunk`. Таким образом, для каждой комбинации категории данных и состояния мы создаем отдельный класс документа, получая в итоге девять типов доменных сущностей. Например, при загрузке исходного документа шаг очистки создает экземпляр `CleanedArticleDocument`, шаг фрагментации возвращает список объектов `ArticleChunk`, а шаг векторизации — экземпляры `EmbeddedArticleChunk`, которые включают в себя эмбеддинг и все необходимые метаданные для загрузки в векторную БД.

Та же логика реализуется в отношении постов и репозиториев.

Мы выбрали такую архитектуру, поскольку список состояний будет изменяться крайне редко, в то время как список категорий данных планируется расширять. Таким образом, структурирование классов на основе состояний позволяет добавлять новые категории данных путем наследования от этих базовых абстрактных классов.

Давайте разберем код иерархии классов на примере очищенного документа. Все атрибуты документа сохраняются в метаданных векторной БД. Например, метаданные очищенной статьи всегда включают ее текст, название платформы, идентификатор автора, его полное имя и ссылку на статью.

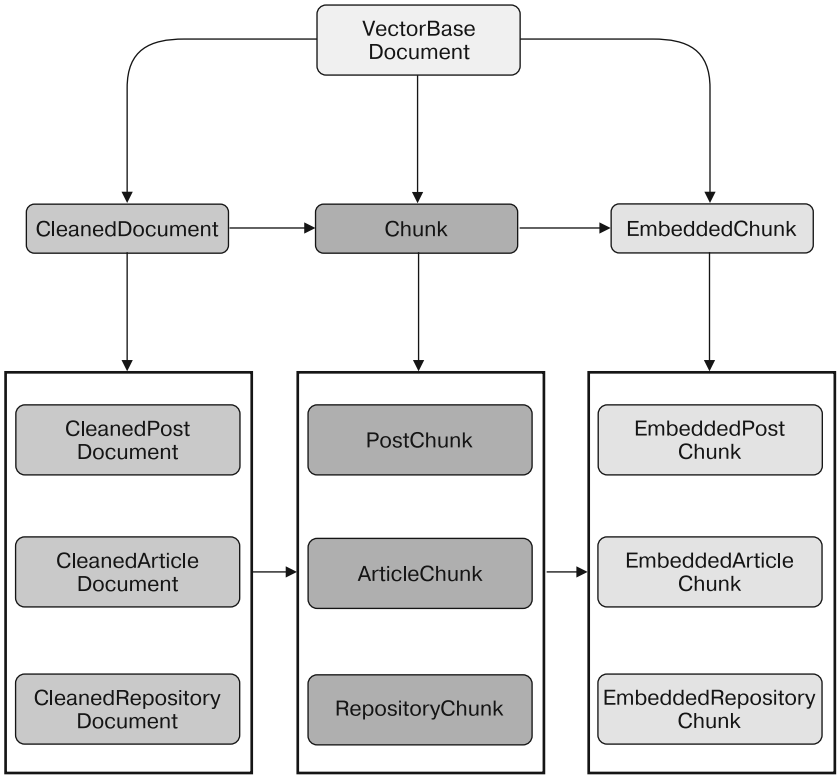


Рис. 4.16. Иерархия классов доменных сущностей и их взаимодействие

Еще одним важным элементом является внутренний класс `Config`, в котором задаются имя коллекции в векторной БД, категория данных соответствующей сущности и информация о том, следует ли использовать векторный индекс при создании коллекции:

```
class CleanedDocument(VectorBaseDocument, ABC):
    content: str
    platform: str
    author_id: UUID4
    author_full_name: str

class CleanedPostDocument(CleanedDocument):
    image: Optional[str] = None

    class Config:
        name = "cleaned_posts"
        category = DataCategory.POSTS
        use_vector_index = False
```

```
class CleanedArticleDocument(CleanedDocument):
    link: str

    class Config:
        name = "cleaned_articles"
        category = DataCategory.ARTICLES
        use_vector_index = False

class CleanedRepositoryDocument(CleanedDocument):
    name: str
    link: str

    class Config:
        name = "cleaned_repositories"
        category = DataCategory.REPOSITORIES
        use_vector_index = False
```

В завершение рассмотрим базовый абстрактный класс для фрагментов и их векторизованных версий:

```
class Chunk(VectorBaseDocument, ABC):
    content: str
    platform: str
    document_id: UUID4
    author_id: UUID4
    author_full_name: str
    metadata: dict = Field(default_factory=dict)

... # PostChunk, ArticleChunk, RepositoryChunk

class EmbeddedChunk(VectorBaseDocument, ABC):
    content: str
    embedding: list[float] | None
    platform: str
    document_id: UUID4
    author_id: UUID4
    author_full_name: str
    metadata: dict = Field(default_factory=dict)

... # EmbeddedPostChunk, EmbeddedArticleChunk, EmbeddedRepositoryChunk
```

Мы также создали класс с константами (`enum`) для объединения всех типов данных:

```
class DataCategory(StrEnum):
    POSTS = "posts"
    ARTICLES = "articles"
    REPOSITORIES = "repositories"
```

Чтобы лучше понять принцип работы доменных объектов, давайте подробно обсудим класс `VectorBaseDocument`.

Объектно-векторное отображение

Паттерн OVM (Object-Vector Mapping — «объектно-векторное отображение») вдохновлен паттерном *объектно-реляционного отображения* (ORM), который мы обсуждали в главе 3. Выбор названия объясняется тем, что мы работаем не с табличными структурами и базами данных SQL, а с эмбеддингами и векторными хранилищами. В остальном паттерны ORM и OVM следуют одним и тем же принципам.

Как и в главе 3, мы реализуем собственную версию OVM. Несмотря на свою простоту, приведенное исполнение демонстрирует способ создания модульных и расширяемых классов с помощью принципов объектно-ориентированного программирования.



Полная реализация класса VectorBaseDocument доступна на GitHub: https://github.com/PacktPublishing/LLM-Engineering/blob/main/llm_engineering/domain/base/vector.py.

Наш базовый OVM-класс называется `VectorBaseDocument` и предназначен для поддержки операций CRUD с хранилищем Qdrant. В нашей системе он будет использоваться только для операций создания и чтения, однако его легко можно расширить для поддержки обновления и удаления данных.

Рассмотрим определение класса `VectorBaseDocument`:

```
from pydantic import UUID4, BaseModel
from typing import Generic

from llm_engineering.infrastructure.db.qdrant import connection

T = TypeVar("T", bound="VectorBaseDocument")

class VectorBaseDocument(BaseModel, Generic[T], ABC):
    id: UUID4 = Field(default_factory=uuid.uuid4)

    @classmethod
    def from_record(cls: Type[T], point: Record) -> T:
        _id = UUID(point.id, version=4)
        payload = point.payload or {}

        attributes = {
            "id": _id,
            **payload,
        }
        if cls._has_class_attribute("embedding"):
            payload["embedding"] = point.vector or None

        return cls(**attributes)
```

```
    def to_point(self: T, **kwargs) -> PointStruct:
        exclude_unset = kwargs.pop("exclude_unset", False)
        by_alias = kwargs.pop("by_alias", True)

        payload = self.dict(exclude_unset=exclude_unset, by_alias=by_alias,
        ➥**kwargs)

        _id = str(payload.pop("id"))
        vector = payload.pop("embedding", {})
        if vector and isinstance(vector, np.ndarray):
            vector = vector.tolist()

        return PointStruct(id=_id, vector=vector, payload=payload)
```

Этот класс наследует от класса `BaseModel` из библиотеки Pydantic и помогает структурировать атрибуты одной записи из векторной ДБ. Каждый объект OVM по умолчанию инициализируется с использованием UUID4 в качестве уникального идентификатора. Благодаря применению обобщений (путем наследования от `Generic[T]`) сигнатуры всех подклассов `VectorBaseDocument` адаптируются под соответствующий тип. Например, метод `from_record()` класса `Chunk`, наследующего от `VectorBaseDocument`, будет возвращать именно тип `Chunk`, что существенно облегчает работу статического анализатора и таких средств проверки типов, как `mypy` (https://mypy.readthedocs.io/en/stable/).

Метод `from_record()` преобразует точку данных[1] из формата Qdrant в нашу внутреннюю структуру на базе Pydantic. А метод `to_point()`, наоборот, преобразует атрибуты текущего экземпляра в формат `PointStruct()`, поддерживаемый Qdrant. Эти два метода будут использоваться в нашем приложении при выполнении операций записи и чтения.

Все взаимодействия с Qdrant происходят через экземпляр подключения (`connection`), который создается на инфраструктурном уровне приложения.

Метод `bulk_insert()` сопоставляет каждый документ с точкой и задействует экземпляр подключения к Qdrant для загрузки всех точек в указанную коллекцию. Если операция вставки оказывается неудачной, метод попытается создать коллекцию и повторить попытку. Хорошей практикой считается разделение логики на две функции, одна из которых — приватная, в нашем случае `_bulk_insert()`, — реализует основную механику, а вторая — публичная — обрабатывает ошибки и сценарии сбоев.

```
class VectorBaseDocument(BaseModel, Generic[T], ABC):
    ... # Остальная часть класса
```

---

[1] Так называется запись в Qdrant, поскольку эмбеддинг в многомерном пространстве является точкой. — *Примеч. науч. ред.*

```
    @classmethod
    def bulk_insert(cls: Type[T], documents: list["VectorBaseDocument"]) -> bool:
        try:
            cls._bulk_insert(documents)
        except exceptions.UnexpectedResponse:
            logger.info(
                f"Collection '{cls.get_collection_name()}' does not exist.
                ➥Trying to create the collection and reinsert the documents."
            )

            cls.create_collection()

            try:
                cls._bulk_insert(documents)
            except exceptions.UnexpectedResponse:
                logger.error(f"Failed to insert documents in '{cls.get_
                ➥collection_name()}'.")

                return False

        return True

    @classmethod
    def _bulk_insert(cls: Type[T], documents: list["VectorBaseDocument"])
    -> None:
        points = [doc.to_point() for doc in documents]

        connection.upsert(collection_name=cls.get_collection_name(),
        ➥points=points)
```

Имя коллекции выводится из класса `Config`, определяемого в каждом подклассе, наследующем от OVM-класса:

```
class VectorBaseDocument(BaseModel, Generic[T], ABC):
    ... # Остальная часть класса

    @classmethod
    def get_collection_name(cls: Type[T]) -> str:
        if not hasattr(cls, "Config") or not hasattr(cls.Config, "name"):
            raise ImproperlyConfigured(
                "Класс должен определять класс Config со свойством 'name',
                ➥которое отражает название коллекции."
            )

        return cls.Config.name
```

Теперь нам необходимо определить метод, который позволит извлекать все записи из векторной БД, не используя логику поиска по векторному сходству. Метод `bulk_find()` предоставляет возможность поочередно извлекать записи из коллекции, возвращая список точек данных, которые затем преобразуются во внутренние структуры с помощью метода `from_record()`.

Параметр `limit` определяет количество записей, возвращаемых в рамках одного вызова, а `offset` задает идентификатор точки, с которой начинается выдача записей.

```
class VectorBaseDocument(BaseModel, Generic[T], ABC):
    ... # Остальная часть класса

    @classmethod
    def bulk_find(cls: Type[T], limit: int = 10, **kwargs) -> tuple[list[T],
    ➥UUID | None]:
        try:
            documents, next_offset = cls._bulk_find(limit=limit, **kwargs)
        except exceptions.UnexpectedResponse:
            logger.error(f"Failed to search documents in
            ➥'{cls.get_collection_name()}'.")

            documents, next_offset = [], None

        return documents, next_offset

    @classmethod
    def _bulk_find(cls: Type[T], limit: int = 10, **kwargs) ->
    ➥tuple[list[T], UUID | None]:
        collection_name = cls.get_collection_name()

        offset = kwargs.pop("offset", None)
        offset = str(offset) if offset else None

        records, next_offset = connection.scroll(
            collection_name=collection_name,
            limit=limit,
            with_payload=kwargs.pop("with_payload", True),
            with_vectors=kwargs.pop("with_vectors", False),
            offset=offset,
            **kwargs,
        )
        documents = [cls.from_record(record) for record in records]
        if next_offset is not None:
            next_offset = UUID(next_offset, version=4)

    return documents, next_offset
```

Наконец, нам нужно определить метод, реализующий поиск по векторному сходству. Как и ранее, мы определяем публичный метод `search()` и приватный метод `_search()`. Сам поиск выполняется системой Qdrant при вызове функции `connection.search()`.

```
class VectorBaseDocument(BaseModel, Generic[T], ABC):
    ... # Остальная часть класса

    @classmethod
    def search(cls: Type[T], query_vector: list, limit: int = 10, **kwargs)
    ➥ -> list[T]:
```

```
        try:
            documents = cls._search(query_vector=query_vector, limit=limit,
            ➥**kwargs)
        except exceptions.UnexpectedResponse:
            logger.error(f"Failed to search documents in
            ➥'{cls.get_collection_name()}'.")
            documents = []

        return documents

    @classmethod
    def _search(cls: Type[T], query_vector: list, limit: int = 10, **kwargs)
    ➥ -> list[T]:
        collection_name = cls.get_collection_name()
        records = connection.search(
            collection_name=collection_name,
            query_vector=query_vector,
            limit=limit,
            with_payload=kwargs.pop("with_payload", True),
            with_vectors=kwargs.pop("with_vectors", False),
            **kwargs,
        )
        documents = [cls.from_record(record) for record in records]

        return documents
```

Теперь, разобравшись в устройстве доменных сущностей и принципах работы OVM, перейдем к обсуждению диспетчеров, которые отвечают за очистку, фрагментацию и векторизацию документов.

Слой диспетчеризации

Диспетчер принимает на вход документ и применяет соответствующий обработчик в зависимости от его категории (статья, пост или репозиторий). Обработчик, в свою очередь, выполняет очистку, фрагментацию или векторизацию этого документа.

Начнем с `CleaningDispatcher`. В его основе лежит метод `dispatch()`, который получает на вход исходный документ. В зависимости от категории данных метод `dispatch()` создает и вызывает соответствующий обработчик, реализующий логику очистки данных этого конкретного типа:

```
class CleaningDispatcher:
    cleaning_factory = CleaningHandlerFactory()

    @classmethod
    def dispatch(cls, data_model: NoSQLBaseDocument) -> VectorBaseDocument:
        data_category = DataCategory(data_model.get_collection_name())
        handler = cls.cleaning_factory.create_handler(data_category)
        clean_model = handler.clean(data_model)
```

```
        logger.info(
            "Data cleaned successfully.",
            data_category=data_category,
            cleaned_content_len=len(clean_model.content),
        )

        return clean_model
```

Ключевым элементом логики диспетчеризации является функция `CleaningHandlerFactory()`, которая создает нужный обработчик на основе категории документа.

```
class CleaningHandlerFactory:
    @staticmethod
    def create_handler(data_category: DataCategory) -> CleaningDataHandler:
        if data_category == DataCategory.POSTS:
            return PostCleaningHandler()
        elif data_category == DataCategory.ARTICLES:
            return ArticleCleaningHandler()
        elif data_category == DataCategory.REPOSITORIES:
            return RepositoryCleaningHandler()
        else:
            raise ValueError("Unsupported data type")
```

Фабрика и диспетчер предоставляют простой и интуитивно понятный интерфейс для выполнения различных операций над документами. Благодаря им мы можем не задумываться о типе документа, когда работаем с бизнес-логикой. Таким образом, у нас есть только один класс для бизнес-логики по очистке любых документов, что соответствует принципу программирования DRY (Don't repeat yourself — «Не повторяйся»). Следование этому принципу дает такие преимущества, как наличие одного места, где сосредоточена наша бизнес-логика, и простота расширения системы. Например, при появлении нового типа данных нам достаточно будет изменить только фабричный класс, а не множество участков кода.

Классы `ChunkingDispatcher` и `EmbeddingDispatcher` построены по одному и тому же шаблону. Они используют фабрики `ChunkingHandlerFactory` и `EmbeddingHandlerFactory`, которые инициализируют подходящий обработчик на основе категории входного документа, а затем вызывают его и возвращают результат.



Исходный код всех диспетчеров и фабрик доступен в репозитории GitHub: https://github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/llm_engineering/application/preprocessing/dispatchers.py.

Фабричный класс реализует порождающий паттерн проектирования Абстрактная фабрика (Abstract Factory, https://refactoring.guru/design-patterns/abstract-factory), который создает целое семейство классов, реализующих общий интерфейс. В нашем случае все обработчики реализуют метод `clean()` независимо от их конкретного типа.

Кроме того, семейство классов `Handler` реализует поведенческий паттерн проектирования Стратегия (Strategy, https://refactoring.guru/design-patterns/strategy), который позволяет инкапсулировать разные варианты алгоритма внутри объекта и переключаться между ними во время выполнения кода.

Работа вышеупомянутых паттернов в слое диспетчеризации может быть описана следующим образом.

1. Мы изначально понимаем, что данные необходимо будет очищать, но, поскольку категория данных становится известной только во время выполнения, у нас нет возможности заранее выбрать подходящую стратегию.
2. Мы можем инкапсулировать логику очистки в универсальный интерфейс `Handler()`, реализующий паттерн Стратегия.
3. Получив очередную точку данных, мы применяем паттерн Абстрактная фабрика, чтобы создать подходящий обработчик для очистки этого конкретного типа данных.
4. В итоге слой диспетчеризации использует этот обработчик и запускает соответствующую стратегию.

Благодаря такому подходу мы:

- изолируем логику для каждой категории данных;
- используем полиморфизм, избавляя код от лишних конструкций `if-else`;
- обеспечиваем модульность и расширяемость кода, поскольку при появлении новой категории данных нам достаточно реализовать новый обработчик и внести изменения только в фабричный класс, не затрагивая остальные части кода.



До сих пор мы лишь моделировали сущности и потоки данных в нашем приложении, но не написали ни одной строки кода, отвечающего за очистку, фрагментацию или векторизацию. В этом и состоит главное отличие между демонстрацией и полноценным приложением. В демонстрационных версиях, как правило, не уделяют внимания лучшим практикам разработки и не заботятся об обеспечении устойчивости кода к будущим изменениям. Однако при создании реального продукта критически важно писать чистый, модульный и масштабируемый код, поскольку именно эти характеристики определяют его надежность и долговечность.

В следующем разделе мы обсудим реализацию обработчиков, отвечающих за очистку, фрагментацию и векторизацию.

Обработчики

Структура обработчиков напрямую соответствует нашей предметной области, то есть для каждой доменной сущности предусмотрен свой обработчик, как показано на рис. 4.17. Всего у нас будет девять классов `Handler`, каждый из которых реализует один из следующих базовых интерфейсов:

- `class CleaningDataHandler()`;
- `class ChunkingDataHandler()`;
- `class EmbeddingDataHandler()`.

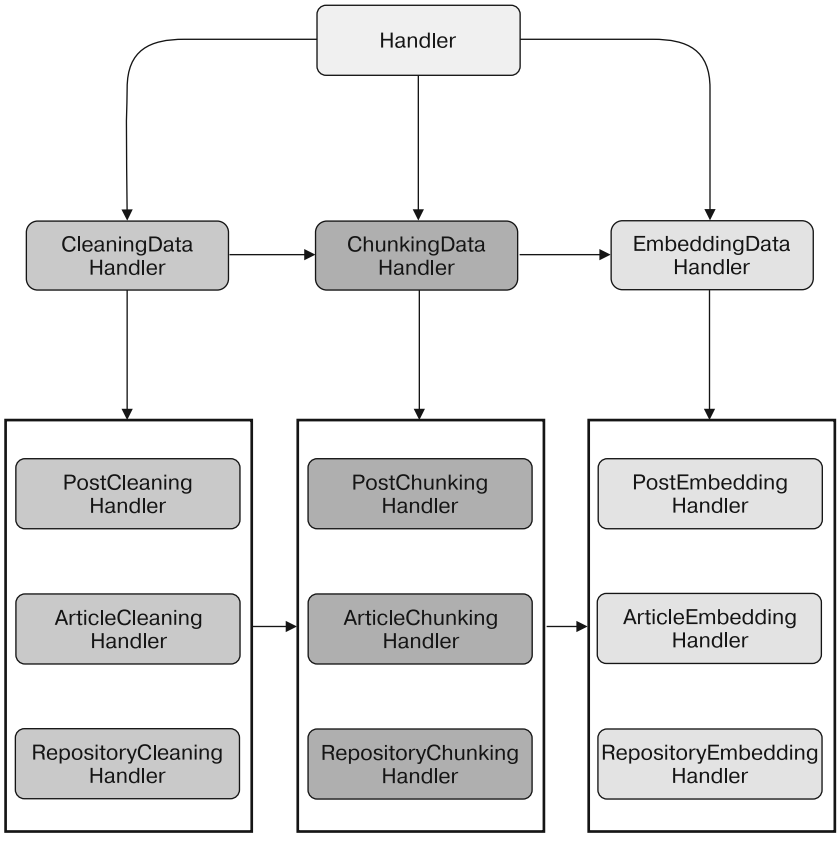


Рис. 4.17. Иерархия классов обработчиков и их взаимодействие



Исходный код всех обработчиков доступен в репозитории GitHub: https://github.com/PacktPublishing/LLM-Engineering/tree/main/llm_engineering/application/preprocessing.

Рассмотрим каждое семейство обработчиков и разберем их устройство.

Отвечающие за очистку

Интерфейс обработчика `CleaningDataHandler()` имеет следующий вид:

```
... # Другие операторы импорта
from typing import Generic, TypeVar

DocumentT = TypeVar("DocumentT", bound=Document)
CleanedDocumentT = TypeVar("CleanedDocumentT", bound=CleanedDocument)

class CleaningDataHandler(ABC, Generic[DocumentT, CleanedDocumentT]):
    @abstractmethod
    def clean(self, data_model: DocumentT) -> CleanedDocumentT:
        pass
```

Для каждой категории данных, будь то пост, статья или репозиторий, мы реализуем отдельный обработчик:

```
class PostCleaningHandler(CleaningDataHandler):
    def clean(self, data_model: PostDocument) -> CleanedPostDocument:
        return CleanedPostDocument(
            id=data_model.id,
            content=clean_text(" #### ".join(data_model.content.values())),
            ... # Скопируйте остальные параметры из объекта data_model
        )

class ArticleCleaningHandler(CleaningDataHandler):
    def clean(self, data_model: ArticleDocument) -> CleanedArticleDocument:
        valid_content = [content for content in data_model.content.values()
        ➥if content]

        return CleanedArticleDocument(
            id=data_model.id,
            content=clean_text(" #### ".join(valid_content)),
            platform=data_model.platform,
            link=data_model.link,
            author_id=data_model.author_id,
            author_full_name=data_model.author_full_name,
        )

class RepositoryCleaningHandler(CleaningDataHandler):
    def clean(self, data_model: RepositoryDocument) -> CleanedRepositoryDocument:
        return CleanedRepositoryDocument(
            id=data_model.id,
            content=clean_text(" #### ".join(data_model.content.values())),
            ... # Скопируйте остальные параметры из объекта data_model
        )
```

Каждый обработчик принимает на вход исходную доменную сущность, очищает ее содержимое и возвращает обработанный документ. Чтобы не усложнять пример, мы применили одинаковую технику очистки текста для всех категорий данных — функцию `clean_text()`. Однако в реальных системах для каждой категории нужно подбирать свой метод. Это легко сделать с помощью паттерна Стратегия, просто заменив функцию очистки в соответствующем обработчике.

Стадии очистки, реализованные в функции `clean_text()`, подробно рассматриваются в разделе «Создание набора данных с инструкциями» главы 5. На данном же этапе нас интересует прежде всего автоматизация и интеграция всей логики в состав RAG-пайплайна признаков. После ввода ML-системы в эксплуатацию все очищенные данные, используемые для тонкой настройки модели, будут извлекаться из логического хранилища признаков, которое станет единым источником истины для получения данных.

Отвечающие за фрагментацию

Сперва рассмотрим интерфейс базового обработчика `ChunkingDataHandler()`. Мы реализовали словарь `metadata` в виде свойства, позволяющего агрегировать все необходимые для фрагментации параметры в одной структуре. Такая организация дает нам возможность записывать в журнал всю информацию в системе ZenML для отслеживания и отладки логики фрагментации. Данный обработчик принимает на вход очищенные документы и возвращает сущности-фрагменты. Все обработчики доступны в репозитории GitHub: https://github.com/PacktPublishing/LLM-Engineering/tree/main/llm_engineering/application/preprocessing.

```
... # Другие операторы импорта
from typing import Generic, TypeVar

CleanedDocumentT = TypeVar("CleanedDocumentT", bound=CleanedDocument)
ChunkT = TypeVar("ChunkT", bound=Chunk)

class ChunkingDataHandler(ABC, Generic[CleanedDocumentT, ChunkT]):

    @property
    def metadata(self) -> dict:
        return {
            "chunk_size": 500,
            "chunk_overlap": 50,
        }

    @abstractmethod
    def chunk(self, data_model: CleanedDocumentT) -> list[ChunkT]:
        pass
```

Теперь посмотрим, как реализован класс `ArticleChunkingHandler()`. Первым шагом является переопределение свойства `metadata` и задание специфичных для логики фрагментации параметров. Например, при работе со статьями нас интересует минимальная и максимальная длина фрагмента.

Метод chunk() данного обработчика принимает очищенные статьи и возвращает список сущностей-фрагментов. Для разбиения текста на фрагменты используется функция chunk_text(), поведение которой зависит от параметров min_length и max_length, заданных в метаданных. Каждому фрагменту присваивается идентификатор chunk_id, вычисляемый на основе его содержимого с применением алгоритма хеширования MD5. Таким образом, фрагменты с идентичным содержанием получают одинаковый идентификатор, что позволяет легко устранить дубликаты. На последнем этапе создается список сущностей-фрагментов, который возвращается вызывающей функции.

```
class ArticleChunkingHandler(ChunkingDataHandler):
    @property
    def metadata(self) -> dict:
        return {
            "min_length": 1000,
            "max_length": 1000,
        }

    def chunk(self, data_model: CleanedArticleDocument) ->
    ➥list[ArticleChunk]:
        data_models_list = []

        cleaned_content = data_model.content
        chunks = chunk_article(
            cleaned_content, min_length=self.metadata["min_length"],
            ➥max_length=self.metadata["max_length"]
        )

        for chunk in chunks:
            chunk_id = hashlib.md5(chunk.encode()).hexdigest()
            model = ArticleChunk(
                id=UUID(chunk_id, version=4),
                content=chunk,
                platform=data_model.platform,
                link=data_model.link,
                document_id=data_model.id,
                author_id=data_model.author_id,
                author_full_name=data_model.author_full_name,
                metadata=self.metadata,
            )
            data_models_list.append(model)

        return data_models_list
```

Функция chunk_article() выполняет две основные задачи.

1. С помощью регулярного выражения она находит все предложения в тексте, ориентируясь на точки, вопросительные и восклицательные знаки, за которыми следует пробел. При этом она старается не разбивать текст в тех случаях, когда знак препинания входит в состав сокращений (например, «т. е.» или «т. к.»).

2. Предложения группируются в один фрагмент вплоть до достижения его максимальной длины (`max_length`). Если максимальный размер фрагмента достигнут и его длина превышает минимально допустимое значение, этот фрагмент добавляется в итоговый список, возвращаемый функцией.

```
def chunk_article(text: str, min_length: int, max_length: int) -> list[str]:
    sentences = re.split(r"(?<!\w\.\w.)(?<![A-Z][a-z]\.)(?<=\.|\?|\!)\s", text)

    extracts = []
    current_chunk = ""
    for sentence in sentences:
        sentence = sentence.strip()
        if not sentence:
            continue

        if len(current_chunk) + len(sentence) <= max_length:
            current_chunk += sentence + " "
        else:
            if len(current_chunk) >= min_length:
                extracts.append(current_chunk.strip())
            current_chunk = sentence + " "

    if len(current_chunk) >= min_length:
        extracts.append(current_chunk.strip())

    return extracts
```

Обработчики `PostChunkingHandler` и `RepositoryChunkingHandler` (доступные в файле репозитория GitHub: `llm_engineering/application/preprocessing/chunking_data_handlers.py`) имеют аналогичную структуру, но задействуют более универсальную функцию `chunk_text()`, процесс работы которой состоит из двух этапов.

1. Сначала используется функция `RecursiveCharacterTextSplitter()` из библиотеки LangChain для разбиения текста на основе заданного разделителя или фиксированного размера фрагмента. При этом первым делом функция пытается определить абзацы с помощью разделителя, но, если они отсутствуют или оказываются слишком длинными, разбиение текста осуществляется на основе размера фрагмента.
2. Далее необходимо убедиться, что размер фрагмента не превышает максимально допустимой длины входного значения для модели векторизации. Для этого мы передаем все созданные фрагменты функции `SentenceTransformersTokenTextSplitter()`, которая учитывает ограничения модели. На данном этапе также применяется логика пересечения фрагментов (`chunk_overlap`), что является актуальным только после проверки длины фрагмента.

```
... # Другие операторы импорта
from langchain.text_splitter import RecursiveCharacterTextSplitter,
➥SentenceTransformersTokenTextSplitter
```

```
from llm_engineering.application.networks import EmbeddingModelSingleton

def chunk_text(text: str, chunk_size: int = 500, chunk_overlap: int = 50) ->
list[str]:
    character_splitter = RecursiveCharacterTextSplitter(separators=["\n\n"],
    ➥chunk_size=chunk_size, chunk_overlap=0)
    text_split_by_characters = character_splitter.split_text(text)

    token_splitter = SentenceTransformersTokenTextSplitter(
        chunk_overlap=chunk_overlap,
        tokens_per_chunk=embedding_model.max_input_length,
        model_name=embedding_model.model_id,
    )
    chunks_by_tokens = []
    for section in text_split_by_characters:
        chunks_by_tokens.extend(token_splitter.split_text(section))

    return chunks_by_tokens
```

Таким образом, эта функция возвращает список фрагментов, соответствующих как заданным параметрам фрагментации, так и ограничениям модели векторизации.

Отвечающие за векторизацию

Обработчики, отвечающие за векторизацию, немного отличаются от остальных, поскольку большая часть логики реализована в интерфейсе `EmbeddingDataHandler()`. Мы выбрали такой подход потому, что при вызове модели векторизации желательно обрабатывать как можно больше фрагментов одновременно, группируя их в пакеты, так как это поможет существенно ускорить работу модели на этапе инференса. При запуске модели на графическом процессоре эти пакеты обрабатываются параллельно и независимо друг от друга, что позволяет увеличить уровень производительности в десять и более раз в зависимости от размера пакета и характеристик оборудования.

Мы реализовали два метода: `embed()` — для векторизации одной точки данных и `embed_batch()` — для их пакетной обработки. Метод `embed_batch()` принимает на вход фрагментированные документы, извлекает их содержимое в список, передает в модель векторизации и сопоставляет полученные результаты с сущностями предметной области, соответствующими векторизованным фрагментам. Это сопоставление осуществляется с помощью абстрактного метода `map_model()`, который необходимо настроить для каждой категории данных.

```
... # Другие операторы импорта
from typing import Generic, TypeVar, cast
from llm_engineering.application.networks import EmbeddingModelSingleton

ChunkT = TypeVar("ChunkT", bound=Chunk)
EmbeddedChunkT = TypeVar("EmbeddedChunkT", bound=EmbeddedChunk)
```

```
embedding_model = EmbeddingModelSingleton()

class EmbeddingDataHandler(ABC, Generic[ChunkT, EmbeddedChunkT]):
    """
    Абстрактный базовый класс для обработчиков данных эмбеддингов.
    Содержит всю логику преобразования данных на этом этапе.
    """

    def embed(self, data_model: ChunkT) -> EmbeddedChunkT:
        return self.embed_batch([data_model])[0]

    def embed_batch(self, data_model: list[ChunkT]) -> list[EmbeddedChunkT]:
        embedding_model_input = [data_model.content for data_model in
        ➥data_model]
        embeddings = embedding_model(embedding_model_input, to_list=True)

        embedded_chunk = [
            self.map_model(data_model, cast(list[float], embedding))
            for data_model, embedding in zip(data_model, embeddings,
            ➥strict=False)
        ]

        return embedded_chunk

    @abstractmethod
    def map_model(self, data_model: ChunkT, embedding: list[float]) ->
    ➥EmbeddedChunkT:
        pass
```

Далее рассмотрим только реализацию `ArticleEmbeddingHandler()`, поскольку остальные обработчики устроены похожим образом. Как видите, нам достаточно определить метод `map_model()`, который принимает на вход фрагмент, вычисляет эмбеддинги и сопоставляет результат с сущностью Pydantic `EmbeddedArticleChunk`.

```
class ArticleEmbeddingHandler(EmbeddingDataHandler):
    def map_model(self, data_model: ArticleChunk, embedding: list[float]) ->
    ➥EmbeddedArticleChunk:
        return EmbeddedArticleChunk(
            id=data_model.id,
            content=data_model.content,
            embedding=embedding,
            platform=data_model.platform,
            link=data_model.link,
            document_id=data_model.document_id,
            author_id=data_model.author_id,
            author_full_name=data_model.author_full_name,
            metadata={
                "embedding_model_id": embedding_model.model_id,
                "embedding_size": embedding_model.embedding_size,
                "max_input_length": embedding_model.max_input_length,
            },
        )
```

Теперь нам осталось разобраться с принципом работы класса `EmbeddingModelSingleton()`. Он представляет собой обертку над классом `SentenceTransformer()` из одноименной библиотеки, которая инициализирует модель векторизации. Создание обертки над внешними пакетами — это хорошая практика, которая позволяет заменить сторонний инструмент, модифицировав только внутреннюю логику обертки, не затрагивая остальной код.

Класс `SentenceTransformer()` инициализируется с помощью идентификатора модели (`model_id`), определенного в классе `Settings`. Благодаря этому мы легко можем тестировать разные модели векторизации, меняя только конфигурационный файл и не переписывая код. Таким образом, мы не заостряем внимание на выборе конкретной модели: она подбирается исходя из задачи, особенностей данных, доступного оборудования и допустимой продолжительности задержки. Универсальный класс с гибкой настройкой позволяет экспериментировать с разными моделями вплоть до нахождения оптимального варианта.

```
from sentence_transformers.SentenceTransformer import SentenceTransformer
from llm_engineering.settings import settings
from .base import SingletonMeta

class EmbeddingModelSingleton(metaclass=SingletonMeta):
    def __init__(
        self,
        model_id: str = settings.TEXT_EMBEDDING_MODEL_ID,
        device: str = settings.RAG_MODEL_DEVICE,
        cache_dir: Optional[Path] = None,
    ) -> None:
        self._model_id = model_id
        self._device = device

        self._model = SentenceTransformer(
        self._model_id,
        device=self._device,
        cache_folder=str(cache_dir) if cache_dir else None,
    )
    self._model.eval()

    @property
    def model_id(self) -> str:
        return self._model_id

    @cached_property
    def embedding_size(self) -> int:
        dummy_embedding = self._model.encode("")

        return dummy_embedding.shape[0]

    @property
    def max_input_length(self) -> int:
        return self._model.max_seq_length
```

```
    @property
    def tokenizer(self) -> AutoTokenizer:
        return self._model.tokenizer

    def __call__(
        self, input_text: str | list[str], to_list: bool = True
    ) -> NDArray[np.float32] | list[float] | list[list[float]]:
        try:
            embeddings = self._model.encode(input_text)
        except Exception:
            logger.error(f"Error generating embeddings for {self._model_id=}
            ➥and {input_text=}")

            return [] if to_list else np.array([])

        if to_list:
            embeddings = embeddings.tolist()

        return embeddings
```

Класс `EmbeddingModelSingleton()` реализует порождающий паттерн проектирования Одиночка (Singleton, https://refactoring.guru/design-patterns/singleton), который допускает создание только одного экземпляра класса и предоставляет глобальную точку доступа к нему. Данный класс наследует от класса `SingletonMeta`, благодаря чему при любой попытке создания нового экземпляра возвращается один и тот же объект. Это особенно удобно при работе с ML-моделями, которые достаточно загрузить в память единожды, а затем использовать в любом месте кода. Такой подход позволяет избежать повторной загрузки модели и проблем, связанных с избыточным потреблением памяти. Кроме того, это упрощает доступ к параметрам вроде размера выходного эмбеддинга (`embedding_size`), вычисление которого требует выполнения пробного прохода вперед через модель векторизации. Благодаря паттерну Одиночка такой проход осуществляется лишь раз, а полученное значение остается доступным на протяжении всего времени выполнения программы.

Резюме

Мы начали главу с обсуждения технологии RAG и ситуаций, в которых имеет смысл ее применять. Мы также выяснили, как работают эмбеддинги и векторные базы данных — ключевые компоненты любой RAG-системы. Затем поговорили о продвинутой RAG-архитектуре и причинах для ее использования. Мы обсудили, какие этапы работы RAG-пайплайна можно оптимизировать, и рассмотрели популярные техники для работы с текстовыми данными. Далее мы попрактиковались в проектировании архитектуры RAG-пайплайна признаков для LLM-двойника. Мы также обсудили различия между пакетной и потоковой архитектурой, а затем познакомились с паттерном CDC, позволяющим синхронизировать два хранилища данных.

В заключительной части главы мы пошагово реализовали RAG-пайплайн признаков для LLM-двойника: интегрировали систему ZenML в качестве оркестратора, описали доменные сущности приложения и спроектировали модуль объектно-векторного отображения (OVM). Кроме того, мы рассмотрели способы применения таких паттернов проектирования, как Абстрактная фабрика и Стратегия, для создания модульного и расширяемого слоя, способного выбирать подходящие стратегии очистки, фрагментации и векторизации данных в зависимости от категории обрабатываемого документа.

В этой главе мы создали только пайплайн загрузки данных, который является лишь частью полноценного RAG-приложения. В главе 9 мы завершим построение RAG-системы, реализовав компоненты поиска и генерации, а затем интегрируем их в пайплайн инференса. В следующей главе поговорим о том, как сгенерировать собственный набор данных на основе собранной информации и использовать его для тонкой настройки LLM.

Список используемых источников

- *Kenton L., Devlin J., Ming-Wei C., Toutanova L. K.* BERT: Pre-training of deep bidirectional transformers for language understanding // Proceedings of NAACL-HLT, June 2019.
- *Liu Y.* RoBERTa: A robustly optimized BERT pretraining approach. arXiv preprint arXiv:1907.11692. 2019.
- *Mikolov T.* Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781. 2013.
- *Pennington J., Socher R., Manning C.* GloVe: Global Vectors for Word Representation // Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014. P. 1532–1543. Association for Computational Linguistics.
- *He K., Zhang X., Ren S., Sun J.* Deep residual learning for image recognition // Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016. P. 770–778.
- *Radford A., Kim J. W., Hallacy C., Ramesh A., Goh G., Agarwal S., Sastry G., Askell A., Mishkin P., Clark J., Krueger G.* Learning transferable visual models from natural language supervision // International Conference on Machine Learning, July 2021. P. 8748–8763. PMLR.
- Confluent. What is Change Data Capture (CDC)? https://www.confluent.io/en-gb/learn/change-data-capture/.
- Refactoring.Guru. Singleton. https://refactoring.guru/design-patterns/singleton.
- Refactoring.Guru. Abstract Factory. https://refactoring.guru/design-patterns/abstract-factory.

- Refactoring.Guru. Strategy. https://refactoring.guru/design-patterns/strategy.
- *Schwaber-Cohen R.* What is a Vector Database & How Does it Work? Use Cases + Examples. Pinecone. https://www.pinecone.io/learn/vector-database/.
- *Monigatti L.* A guide on 12 tuning strategies for production-ready RAG applications. Medium. https://towardsdatascience.com/a-guide-on-12-tuning-strategies-for-production-ready-rag-applications-7ca646833439
- *Monigatti L.* Advanced Retrieval-Augmented Generation: From Theory to LlaMaIndex Implementation. Medium. https://towardsdatascience.com/advanced-retrieval-augmented-generation-from-theory-to-llamaindex-implementation-4de1464a9930
- *Maameri S.* Routing in RAG-Driven Applications. Medium. https://towards-datascience.com/routing-in-rag-driven-applications-a685460a7220.

5 Тонкая настройка с учителем

Тонкая настройка, или дообучение, с учителем (Supervised Fine-Tuning, SFT) — это ключевой этап подготовки больших языковых моделей к практическому использованию. После предварительного обучения, в ходе которого модель приобретает способность предсказывать следующий токен в последовательности, на этапе SFT ее поведение уточняется с помощью тщательно отобранных пар «инструкция — ответ». У этого процесса две основные задачи: обучить модель понимать и соблюдать определенный формат диалога, превращая ее в полноценного собеседника, и позволить адаптировать обширную базу знаний модели для эффективного решения задач в специализированных предметных областях.

Тонкая настройка помогает преодолеть разрыв между общим языковым пониманием модели и ее практической полезностью. Предоставляя модели примеры желаемых связей между запросом и ответом, механизм SFT формирует ее поведение в соответствии с конкретными целями — будь то выполнение задач (например, обобщение или перевод текста) или демонстрация экспертных знаний в определенной области (скажем, в медицине или юриспруденции). Такой целенаправленный подход не только повышает эффективность модели в заданных сценариях, но и улучшает ее способность следовать инструкциям и генерировать более релевантные и связные ответы.

К концу главы вы научитесь самостоятельно создавать наборы инструкций и эффективно настраивать LLM на их основе.



Все примеры кода из этой главы доступны в репозитории GitHub: https://github.com/PacktPublishing/LLM-Engineering.

Создание набора данных с инструкциями

Во многих случаях именно создание набора данных с инструкциями оказывается самой сложной частью процесса тонкой настройки. На это есть множество причин. Практически любую задачу можно связать с исходным текстом, однако естественные пары «инструкция — ответ» встречаются крайне редко.

Следовательно, такие тексты необходимо преобразовывать в формат, включающий и инструкции, и ответы. Кроме того, важным фактором является качество данных. По этой причине значительное время уходит на ручную проверку и верификацию отдельных образцов. Такой тщательный анализ позволяет обеспечить достоверность и полезность набора данных для обучения модели.

В этом разделе мы рассмотрим общий подход к созданию наборов данных с инструкциями, не зависящий от конкретного сценария использования. Затем мы построим такой набор на основе данных, собранных в главе 3. Этапы этого процесса показаны на рис. 5.1.

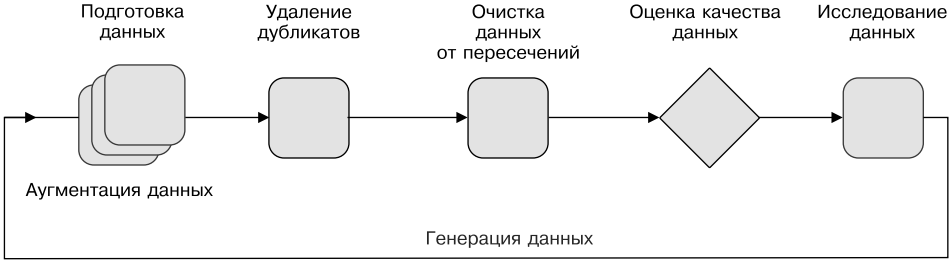


Рис. 5.1. Обзор пайплайна подготовки данных, рассматриваемого в этой главе

Процесс создания наборов данных с инструкциями

Наборы данных для обучения следованию инструкциям состоят из пар «инструкция — ответ». Инструкция служит входными данными для модели и используется в качестве контекста при тонкой настройке. Ответ — это ожидаемый результат генерации. Модель можно тонко настраивать и на парах «инструкция — ответ», и на одних ответах. Такие пары имеют определенный шаблон. Некоторые шаблоны инструкций, например Alpaca, включают дополнительные поля вроде `inputs` и `system`, которые можно рассматривать как вложенные элементы поля `instruction`. Поле `inputs` содержит конкретные данные, необходимые для выполнения инструкции, а `system` представляет собой метапромпт, определяющий общее поведение модели. Рассмотрим пример из набора данных SlimOrca, содержащий поля `system` и `instruction` (табл. 5.1).

Пример показывает, как с помощью поля `system` можно задать специфическое поведение модели, суть которой заключается в том, чтобы быть полезной, всегда предоставлять объяснения и формулировать ответы, понятные даже ребенку. Поле `instruction` содержит исходные данные (понятия) и задачу (составление предложения), а поле `output` — ожидаемый ответ, который, хоть и не является единственно возможным, представляет собой качественный вариант ответа.

Таблица 5.1. Пример из набора данных Open-Orca/SlimOrca

Системное задание: (поле `system`) Ты полезный помощник, способный объяснить все что угодно. Представь, что отвечаешь пятилетнему ребенку
Инструкция: (поле `instruction`) Понятия: здание, магазин, город. Составь предложение, в котором используются все эти слова
Результат: (поле `output`) В нашем городе есть магазин внутри большого здания, куда люди приходят для того, чтобы купить любимые игрушки и конфеты

Чтобы сформировать обучающий набор инструкций, необходимо сначала отобрать данные, отражающие реальные сценарии использования модели, а затем отфильтровать их, оставив лишь высококачественные. В этом контексте качество данных можно оценить по трем основным критериям.

- *Точность*. Подразумевает фактологическую достоверность и релевантность образцов. В контексте наборов данных с инструкциями это означает, что ответы должны соответствовать не только фактам, но и конкретным инструкциям. Высокая точность данных критически важна для обучения моделей, способных предоставлять надежную и достоверную информацию.
- *Разнообразие*. Наш набор данных должен охватывать широкий спектр сценариев, отражающих разнообразные запросы и задачи, с которыми может столкнуться развернутая LLM. Это разнообразие должно распространяться на тематику, контекст, длину текстов и стиль изложения. Использование репрезентативной выборки для обучения модели делает ее более способной следовать инструкциям.
- *Сложность*. Примитивные или слишком простые примеры практически не способствуют повышению качества работы модели. В набор стоит включать сложные задачи, приближенные к реальности и требующие многоэтапного рассуждения.

В следующих подразделах мы рассмотрим техники фильтрации и оценки образцов инструкций по этим критериям.

Объем данных. На платформе Hugging Face Hub представлено множество наборов данных с инструкциями — как общего назначения, так и специализированных, ориентированных на конкретные задачи или области (рис. 5.2). При разработке новой прикладной модели вы можете использовать открытые наборы данных для тонкой настройки. Это особенно актуально, если имеющийся в вашем распоряжении объем данных невелик (например, менее 1000 образцов) и его необходимо дополнить высококачественными данными.

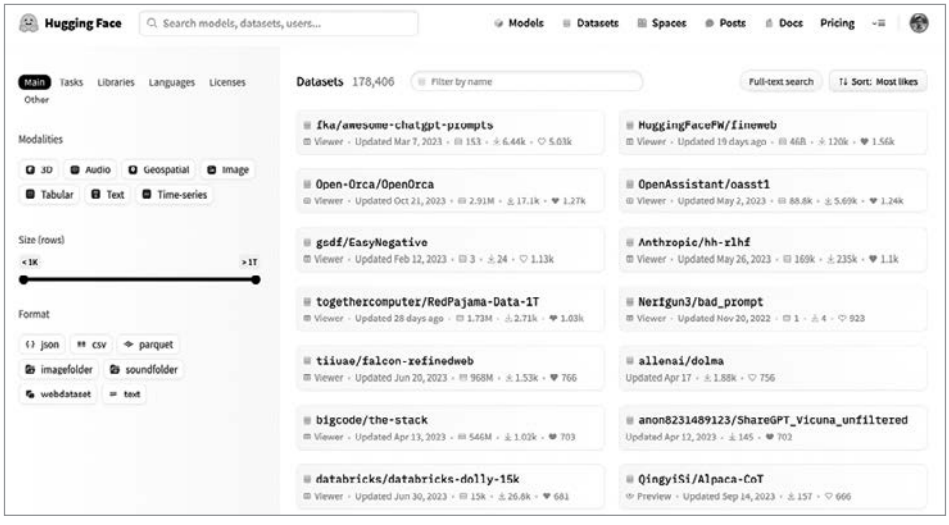


Рис. 5.2. Самые популярные наборы данных на платформе Hugging Face Hub

Подсчитать оптимальное количество обучающих примеров — задача непростая, поскольку на него существенно влияет и качество данных, и размер модели. Для крупных моделей (имеющих порядка 70 млрд параметров) может быть достаточно всего 1000 качественных образцов (см. статью *LIMA: Less Is More for Alignment*). Однако моделям меньшего размера (предусматривающим около 7 млрд параметров) требуется гораздо больше примеров даже для освоения корректного шаблона ведения диалога. В любом случае качество данных имеет решающее значение и почти всегда желательно использовать увеличенный объем выборки.

Чтобы получить дополнительное представление о масштабах тонкой настройки, можно обратиться к моделям, обученным компаниями и сообществом разработчиков открытого ПО. Здесь можно выделить два основных типа моделей: универсальные, стремящиеся воспроизвести возможности таких моделей, как GPT, и специализированные — ориентированные на достижение высокой точности в рамках конкретной предметной области.

Универсальные модели охватывают довольно широкий круг тем, что требует использования большего количества обучающих примеров. В корпоративном секторе наблюдается значительное разнообразие подходов. Так, модели Yi от компании 01.AI были тонко настроены на менее чем 10 000 образцов. На противоположном конце спектра находится компания, которая сообщила об использовании 10 млн образцов в процессе тонкой настройки модели Llama 3 (включая этап выравнивания модели по предпочтениям человека). Модели с открытым

исходным кодом, такие как OpenHermes и Dolphin, задействуют около 1 млн образцов. Учитывая качество полученных результатов, для создания конкурентоспособной модели общего назначения мы рекомендуем использовать набор данных с инструкциями, содержащий как минимум 1 млн примеров. В то же время для обучения специализированных моделей требуется меньший объем данных. В этом контексте важно различать модели, ориентированные на определенную задачу и на определенную предметную область.

Узкоспециализированные, или задача-ориентированные (task-specific), модели оптимизируются под выполнение одной функции, например перевода, резюмирования или анализа тональности текста. Такие модели выигрывают от узконаправленного обучения, что позволяет достигать высокой точности даже при относительно небольшом размере модели (менее 8 млрд параметров). Объем данных, необходимый для тонкой настройки подобной модели, обычно невелик и составляет от 100 до 100 000 примеров, что делает этот подход особенно привлекательным при ограниченных ресурсах.

Предметно-ориентированные модели (domain-specific) обучаются на специализированных знаниях и лексике, характерной для конкретной отрасли. Они особенно актуальны в медицине, юриспруденции, финансах, электронной коммерции, инженерии и гостиничном деле. Необходимый объем данных для их тонкой настройки зависит от сложности и масштаба предметной области. Например, в случае с такими обширными сферами, как медицина и право, для тонкой настройки моделей может потребоваться столько же данных, сколько для моделей общего назначения. Напротив, модели для узкоспециализированных областей вроде гостиничного бизнеса или электронной коммерции обходятся меньшим числом примеров, сопоставимым с объемом, характерным для тонкой настройки моделей, ориентированных на конкретную задачу.

Ключевыми факторами, определяющими объем данных, необходимый для тонкой настройки предметно-ориентированных моделей, являются размер соответствующей области (то есть объем специализированных знаний и количество терминов), а также степень ее представления среди общего массива данных, использованных для предварительного обучения модели. Если тематика уже хорошо отражена в исходных данных, то модель может демонстрировать приемлемые результаты без существенной тонкой настройки. Напротив, в случае с узкоспециализированными или слабо представленными областями модели понадобится более внушительный обучающий набор данных для тонкой настройки. Даже при использовании LLM с открытым исходным кодом следует учитывать, что многие наборы данных, использованные на этапе предварительного обучения, являются закрытыми. Это вынуждает исследователей оценивать состав таких наборов лишь приблизительно (например, что 30 % данных представляют собой программный код, а 20 % — материалы по математике).

Подготовка данных

При подготовке данных для тонкой настройки подходы к их сбору различаются в зависимости от типа модели. В случае узкоспециализированных моделей подготовка данных, как правило, предполагает сбор примеров выполнения целевой задачи из существующих наборов данных или создание новых. Например, в случае модели, предназначенной для резюмирования текста, это может означать подбор пар «исходный текст — краткое изложение», а для модели, предназначенной для перевода, — сбор предложений на разных языках.

Подготовка данных для тонкой настройки предметно-ориентированных моделей обычно представляет собой более сложный процесс, требующий участия экспертов для сбора и проверки релевантных текстов, научных публикаций, технической документации и других специфических материалов. В некоторых случаях может потребоваться участие организаций, обладающих доступом к крупным специализированным БД. Качество и тематическая релевантность этих данных имеют решающее значение, поскольку они напрямую влияют на способность модели понимать и генерировать информацию в контексте предметной области.

Следует отметить, что в последние годы *метод обучения на малом числе примеров (few-shot prompting)* зарекомендовал себя как альтернатива тонкой настройке в контексте узкоспециализированных приложений. Этот подход использует возможности крупных и мощных моделей, предоставляя им несколько примеров выполнения задачи непосредственно в составе входного промпта. Хотя такой метод не может полностью заменить тонкую настройку во всех сценариях (например, при необходимости освоить новую предметную область), он представляет собой эффективный способ адаптации моделей к новым задачам без необходимости проведения обширного дополнительного обучения.

На практике граница между двумя моделями, рассмотренными в данном разделе, часто размыта. Например, модель, тонко настроенная для постановки медицинских диагнозов, может одновременно считаться как узкоспециализированной (поскольку фокусируется на диагностике), так и предметно-ориентированной (поскольку опирается на знания в области медицины). Таким образом, важно понимать основную цель тонкой настройки и выбирать соответствующий подход.

К этому моменту мы уже должны обладать целой коллекцией наборов данных, подходящих для предполагаемого сценария использования. Следующим шагом станет повышение их качества путем фильтрации на основе правил, удаления дубликатов, очистки от пересечений и оценки.

Фильтрация на основе правил

Фильтрация на основе правил представляет собой систематический подход к контролю качества данных, основанный на использовании заранее определенных, явно сформулированных правил для оценки и отсева образцов. Такие правила обычно разрабатываются для устранения типичных проблем, связанных с качеством, и могут варьироваться от простых проверок до более сложных логических операций. Основной целью фильтрации на основе правил является поддержание высокого уровня качества данных путем удаления образцов, не соответствующих установленным требованиям.

Фильтрация по длине — это один из наиболее простых и в то же время эффективных приемов, суть которого заключается в задании пороговых значений для длины ответов в наборе данных. Слишком короткие ответы часто не несут достаточной смысловой нагрузки, а чрезмерно длинные могут содержать избыточную или нерелевантную информацию. Следует отметить, что допустимые пределы длины ответов могут значительно варьироваться в зависимости от конкретной задачи и предметной области. Например, в случае набора данных, предназначенного для генерации кратких изложений, максимальный допустимый объем текста будет меньше, чем для набора, ориентированного на генерацию подробных объяснений.

Удаление по ключевым словам — это еще один действенный метод фильтрации на основе правил, когда внимание уделяется не структуре примеров, а их наполнению. Суть этого метода — в составлении списка ключевых слов или выражений, связанных с низким качеством или нежелательным содержанием, и удалении всех примеров, в которых они встречаются. Такой список может включать ненормативную лексику, термины, характерные для спам-сообщений, а также специфический контент, не соответствующий предметной области. Например, в наборе данных для профессионального помощника по написанию текстов могут быть исключены образцы, в которых употребляются сленговые выражения или разговорные обороты, не соответствующие предполагаемому стилю и тону письма.

Проверка формата — эту технику рекомендуется применять к наборам данных, где присутствуют структурированные элементы или строгие требования к оформлению. Такая проверка гарантирует согласованность примеров и облегчает их дальнейшую обработку. Она особенно актуальна для наборов данных, содержащих фрагменты кода, JSON-структуры и другой отформатированный текст. Например, в наборе данных с программными заданиями можно установить правила, проверяющие синтаксическую корректность фрагментов кода и их соответствие заданным стилевым нормам.

Фильтрация на основе правил обладает рядом существенных преимуществ при подготовке наборов данных с инструкциями. Ее высокая скорость и эффективность позволяют быстро обрабатывать большие объемы данных, что делает этот подход легко масштабируемым. Последовательное применение правил обеспечивает единообразную обработку данных, снижая вероятность ошибок и предвзятости со стороны человека. Кроме того, явное задание критериев фильтрации способствует прозрачности и интерпретируемости процесса, облегчая его понимание, аудит и настройку. Возможность автоматизировать фильтрацию на основе правил сводит к минимуму необходимость в ручной проверке и позволяет организовать постоянный мониторинг качества данных.

Тем не менее у этого подхода есть и ограничения. Заранее заданные правила не всегда способны учитывать все нюансы языка и контекста, что может привести к удалению корректных, но редких примеров. Бинарный характер правил не всегда адекватно отражает сложность естественного языка и разнообразие инструкций. Кроме того, по мере изменения структуры данных и повышения стандартов качества сами правила должны пересматриваться и обновляться, чтобы оставаться эффективными. Существует также риск того, что плохо сформулированные правила приведут к появлению или усилению предвзятости в наборе данных.

Удаление дубликатов

Разнообразие данных в наборе играет ключевую роль в обучении моделей, способных обобщать информацию и работать с новыми, ранее не встречавшимися примерами. Наличие дубликатов или почти идентичных записей в наборе данных может привести к следующим проблемам.

- *Переобучение*. Модель начинает запоминать конкретные примеры вместо того, чтобы выявлять общие закономерности.
- *Искажение результатов*. Избыточное количество определенных примеров может исказить поведение модели, делая ее предвзятой к конкретным типам входных данных.
- *Неэффективное обучение*. Повторяющиеся записи увеличивают время обучения, не добавляя полезной информации.
- *Завышение показателей производительности*. Наличие дубликатов в тестовой выборке может создать ложное впечатление о высокой производительности при оценке модели.

Устранение повторов в данных может осуществляться на основе точных и нечетких совпадений. *Удаление дубликатов на основе точных совпадений* (exact deduplication) предполагает отсеивание полностью идентичных записей посредством стандартной процедуры, включающей нормализацию данных, вычисление хешей и устранение дубликатов. Нормализация данных позволяет

стандартизировать формат записей, например преобразовать текст в нижний регистр. Далее для каждой записи вычисляется уникальный хеш с помощью алгоритмов вроде MD5 или SHA-256. Совпадающие хеши указывают на дубликаты, которые затем удаляются. В результате в наборе данных остается только один экземпляр каждой записи. Хотя такой метод эффективен в отношении полностью идентичных образцов, он не позволяет выявить почти идентичные или семантически схожие записи. Для этого требуются более сложные подходы.

Наиболее распространенным методом *удаления дубликатов на основе нечетких совпадений* (fuzzy deduplication) является алгоритм MinHash. По сравнению с другими методами нечеткого сопоставления этот подход обеспечивает высокую точность при значительно меньших вычислительных затратах. Алгоритм MinHash формирует компактные представления, или сигнатуры, которые служат своеобразными «отпечатками» элементов данных, отражающими их суть при существенном снижении размерности. На практике MinHash преобразует элементы данных (например, текстовые документы) в наборы шинглов (shingle, фрагмент текста фиксированной длины), применяет к ним несколько хеш-функций и выбирает минимальные значения хешей для формирования вектора сигнатуры. Эти сигнатуры затем можно сравнивать между собой с помощью таких метрик, как коэффициент сходства Жаккара, для выявления близких по содержанию элементов.

Помимо удаления дубликатов на основе точных и нечетких совпадений, существует подход, основанный на *семантическом сходстве*, который ориентирован на смысловую близость анализируемых текстов. При использовании этого метода слова или целые тексты преобразуются в векторные представления с помощью различных техник обработки естественного языка. Модели векторизации слов, такие как Word2Vec, GloVe и FastText, преобразуют отдельные слова в плотные векторы, отражающие семантические взаимосвязи между ними.

Для получения более контекстно зависимых представлений можно использовать языковые модели, например BERT, или кросс-энкодеры, которые позволяют формировать эмбеддинги для целых предложений или документов. После получения таких векторных представлений удаление дубликатов осуществляется путем сравнения степени сходства между векторами. Наиболее распространенными метриками для этого являются косинусное сходство и евклидово расстояние. Образцы, чья степень сходства превышает заданный порог, считаются дубликатами. В случае больших наборов данных можно использовать методы кластеризации для группировки семантически близких векторов. Такие подходы, как метод k-средних, DBSCAN или иерархическая кластеризация, позволяют эффективно организовать векторное пространство и выделить кластеры, представляющие семантически схожие тексты. Внутри каждого такого кластера можно сохранить один репрезентативный образец, пометив остальные как дубликаты.

Очистка данных от пересечений

Очистка данных от пересечений (data decontamination) предполагает устранение из обучающей выборки примеров, идентичных или близких к тем, что содержатся в тестовом или проверочном наборе данных. Этот этап критически важен для обеспечения объективной оценки модели и предотвращения ее переобучения или запоминания тестовых данных.

Для устранения таких данных применяются те же методы, что и для удаления дубликатов. На первом этапе из обучающей выборки убираются все примеры, в точности совпадающие с тестовыми. Это достигается с помощью хеш-функций или прямого сравнения строк. Далее можно использовать такие методы выявления близких (почти совпадающих) примеров, как алгоритм MinHash и оценка степени сходства на основе *n*-грамм и векторных представлений.



Простой способ реализовать очистку данных — добавить проверочный набор к набору данных с инструкциями на этапе удаления дубликатов. При этом важно обеспечить устранение дубликатов только из набора с инструкциями. Это можно реализовать разными способами, например путем исключения только первого дубликата или предварительной фиксации индексов образцов из оценочного набора. Идеальным вариантом является автоматическое включение оценочных наборов на этапе удаления дубликатов. Такой подход особенно эффективен при итеративной работе с несколькими версиями пользовательских бенчмарков.

Еще одним результатом очистки данных от пересечений является фильтрация образцов, полученных из тех же источников, что и данные в оценочном наборе. Этот процесс может включать поиск совпадающих фраз, сходных синтаксических конструкций или общих метаданных. В некоторых случаях могут применяться методы отслеживания происхождения данных (provenance tracking), чтобы выявить и исключить материалы из источников, использованных при формировании наборов для оценки модели.

Оценка качества данных

Оценка качества данных — это критически важный аспект машинного обучения, особенно в контексте работы с большими языковыми моделями (LLM). Этот процесс предполагает анализ различных характеристик набора данных, таких как точность, разнообразие и сложность. Некоторые свойства, вроде математической точности (например, при сравнении числа из ответа с эталонным), можно проверить автоматически с помощью интерпретаторов Python и других инструментов, однако оценка субъективного или открытого контента (например, если правильных ответов на вопрос множество или это просьба написать стихотворение) остается весьма сложной задачей.

К традиционным методам оценки качества данных относится ручная оценка, которая, как правило, отличается высокой точностью, но требует значительных ресурсов. Чтобы решить проблему масштабируемости, были разработаны методы автоматизации оценки данных на основе машинного обучения. К ним относится использование языковых моделей в роли экспертов (LLM-as-a-judge), а также моделей вознаграждения и классификаторов, специально обученных для предсказания оценки качества.

Стратегия привлечения *LLM в качестве эксперта* предполагает применение языковой модели для оценки качества каждого отдельного примера путем подачи на ее вход соответствующего промпта. Этот подход получил широкое распространение благодаря своей гибкости и простоте реализации, однако он сопряжен с рядом трудностей. Разные LLM демонстрируют различную производительность в зависимости от типа задачи, и их оценки часто оказываются близки к суждениям неспециалистов. В случае предметно-ориентированных наборов данных использование специализированной LLM вместо более мощной, но универсальной может оказаться целесообразнее. На практике сравнительные методы оценки (например, «Превосходит ли ответ A ответ B?») обычно оказываются более эффективными, чем абсолютные (например, «Оцените ответ A по шкале от 1 до 4»), хотя оба подхода вполне могут применяться при условии грамотного составления промптов. Мы рекомендуем тестировать различные формулировки промптов на репрезентативной подвыборке и вручную проверять корректность получаемых оценок. Рассмотрим пример пользовательского промпта для LLM-эксперта.

Пример промпта для LLM-эксперта, применяемого для оценки качества данных

Инструкция

Ты выступаешь в роли эксперта по оценке качества данных. Твоя задача — проанализировать инструкцию и соответствующий ей ответ, чтобы определить, насколько эффективно ответ решает поставленную задачу.

В процессе оценки ты должен сформулировать отзыв, описывающий сильные и слабые стороны ответа, и присвоить ему оценку по шкале от 1 до 4.

Оценка 1 означает, что ответ крайне плох и не имеет отношения к инструкции.

Оценка 2 означает, что ответ бесполезен и упускает важные аспекты инструкции.

Оценка 3 означает, что ответ полезен, но нуждается в улучшении с точки зрения релевантности, точности или глубины.

Оценка 4 означает, что ответ отличный и полностью соответствует заданной инструкции.

Оформи свою оценку в следующем виде:

Отзыв: (укажи сильные и слабые стороны ответа)

Оценка: (число от 1 до 4)

Известно, что при использовании LLM-эксперта могут возникать определенные искажения. Во-первых, при сравнительной оценке наблюдается позиционная предвзятость, когда модель склонна отдавать предпочтение первому из двух представленных ответов. Это можно компенсировать, изменяя порядок ответов A и B. Кроме того, как и люди, LLM-эксперты чаще выбирают длинные ответы. Для снижения такого эффекта при использовании абсолютной оценки можно применять техники нормализации по длине. Наконец, для моделей-экспертов характерна внутрисемейная предвзятость, то есть они чаще отдают предпочтение ответам, сгенерированным моделями из того же семейства (например, GPT-4o предпочитает ответы, сгенерированные моделями GPT-4 и GPT-4o mini). Эту проблему можно смягчить, если задействовать не одну, а несколько моделей-экспертов.

Чтобы повысить надежность оценки, рекомендуется использовать совокупность моделей (LLM-жюри). Такой подход снижает влияние искажений и повышает согласованность результатов. Включение в состав «жюри» менее ресурсоемких моделей позволяет снизить затраты, повысить уровень точности и компенсировать внутрисемейную предвзятость. При разработке прикладных моделей, например чат-ботов, целесообразно добиваться высокой степени согласованности между оценками LLM-экспертов и людей (порядка 80 %). Рекомендуется также использовать простые шкалы оценки и бенчмарки, ориентированные на конкретную задачу, чтобы обеспечить интерпретируемость и релевантность результатов.

Модели вознаграждения (reward-модели) представляют собой еще один способ использования LLM для оценки качества данных. Данный термин заимствован из области обучения с подкреплением на основе отзывов людей (см. главу 6). В общем случае моделью вознаграждения называется модель, которая принимает на вход пару «инструкция — ответ» и возвращает числовую оценку. Обычно подобные модели создаются путем добавления линейного слоя поверх моделей, использующих только декодер, таких как Gemma или Llama. Затем они адаптируются под решение конкретной задачи с помощью методов обучения с подкреплением или традиционной тонкой настройки с учителем. На рис. 5.3 представлена архитектура модели ArmoRM-Llama3-8B-v0.1, в которой поверх модели Llama 3 с 8 млрд параметров добавлены регрессионный и управляющий слои. Данная модель возвращает несколько оценок, каждая из которых соответствует таким аспектам качества данных, как полезность, точность, согласованность, сложность и многословие, что позволяет осуществлять более детализированную оценку.

Таблица лидеров RewardBench, разработанная Институтом искусственного интеллекта Аллена и размещенная на платформе Hugging Face (allenai/reward-bench), представляет собой полезный ресурс для сравнения моделей возна-

граждения различных типов, включая генеративные модели, классификаторы, модели, обученные с помощью DPO (Direct Preference Optimization — «прямая оптимизация предпочтений»), и др. Эти модели оцениваются на тщательно отобранных примерах, содержащих как предпочтительные, так и отклоненные ответы для каждой инструкции. Хотя эта задача напрямую не связана с оценкой качества данных с инструкциями, RewardBench служит хорошим ресурсом для поиска моделей, способных отличать качественные ответы от некачественных.

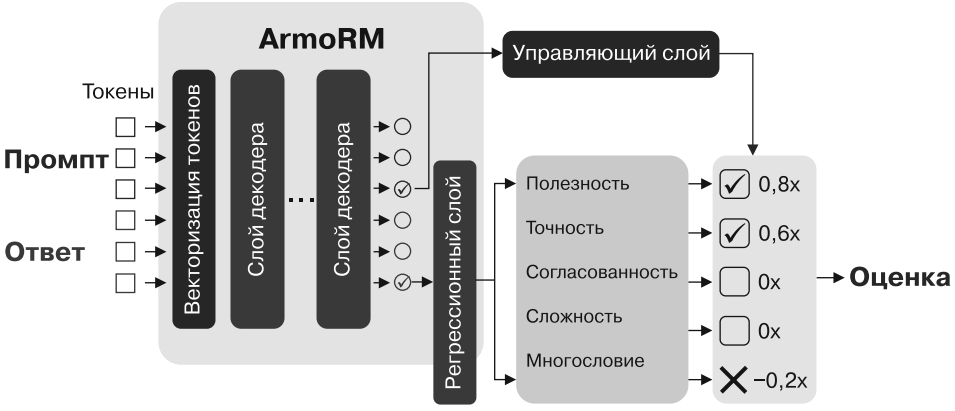


Рис. 5.3. Архитектура модели RLHFlow/ArmoRM-Llama3-8B-v0.1 на базе Llama 3 (источник: https://doi.org/10.48550/arXiv.2406.12845)

Классификаторы или модели, использующие только энкодер (encoder-only), тоже могут быть обучены для оценки качества данных. Хорошим примером является модель HuggingFaceFW/fineweb-edu-classifier — классификатор, предназначенный для оценки образовательной ценности веб-страниц. Изначально эта модель применялась для фильтрации данных, используемых на этапе предварительного обучения, однако аналогичный подход может применяться и для масштабной оценки наборов данных с инструкциями.

Такой подход основан на применении моделей, использующих только энкодер. Они обычно компактнее и лучше подходят для решения задач классификации. Благодаря малому числу параметров такие модели работают быстрее и легко масштабируются до миллионов образцов. Однако они менее точны по сравнению с крупными архитектурами (особенно при решении задач, требующих сложного рассуждения), поскольку не способны улавливать тонкие смысловые различия. Тем не менее на малых масштабах подобные модели хорошо справляются с фильтрацией выбросов и эффективны в составе автоматизированных пайплайнов, предназначенных для быстрой обработки данных.

Исследование данных

Исследование данных — это непрерывный процесс, в ходе которого исследователи знакомятся с обучающим набором. Он включает как ручную проверку, так и автоматизированный анализ — оба этих подхода играют ключевую роль в понимании свойств, сильных сторон и потенциальных недостатков набора данных.

Ручное исследование набора данных, несмотря на трудоемкость, является важным этапом, позволяющим выявить ошибки и несоответствия, которые могут быть упущены при автоматическом анализе, в том числе проблемы с форматированием, ошибки ввода, нелогичные рассуждения и фактические неточности. Этот процесс помогает качественно оценить содержание и стиль данных. Для повышения его эффективности можно применять такие методы, как стратифицированная выборка (отбор разнообразных образцов), систематический обзор (на основе перечня критериев) и совместное рецензирование (с участием нескольких экспертов).

На рис. 5.4 приведен пример такого анализа с использованием Argilla — платформы для совместной оценки и исследования качества данных.

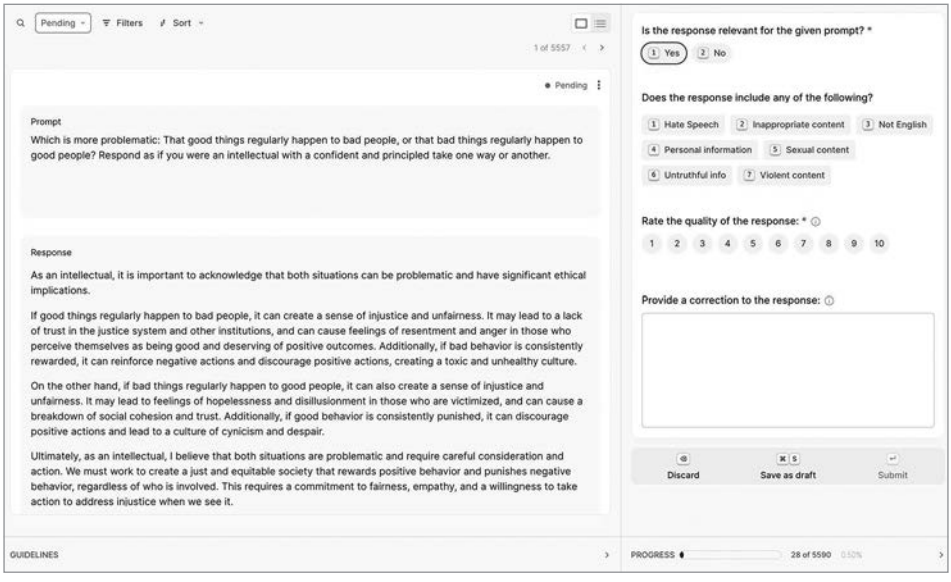


Рис. 5.4. Интерфейс платформы Argilla для совместной оценки и исследования качества данных

Статистический анализ дополняет ручное исследование данных и помогает выявить разнообразие словаря, возможные искажения и способы представления понятий. Этот подход опирается на библиотеки для обработки естественного языка, например NLTK или spaCy, которые отвечают за токенизацию и анализ больших объемов текста. Инструменты визуализации, вроде Matplotlib и Seaborn, позволяют строить гистограммы и облака слов, облегчая распознавание закономерностей. Эти методы дают представление о составе набора данных, разнообразии языковых средств и возможных культурных или контекстных[1] предпочтениях, способных повлиять на поведение модели.

Кластеризация тем позволяет автоматически группировать схожие документы или текстовые фрагменты, выявляя скрытые темы и закономерности в данных. Этот метод особенно важен для анализа больших текстовых массивов, выявления трендов и упорядочения информации. Часто кластеризация сопровождается визуализацией, например построением диаграмм, отображающих группы близких по содержанию образцов.

Представьте, что вы создаете набор данных с инструкциями, посвященных различным языкам программирования. Вы собрали обширный корпус текстов, связанных с программированием, из форумов, документации и обучающих материалов. Во-первых, кластеризация тем позволяет выделить отдельные языки в наборе (Python, JavaScript и т. д.). Во-вторых, внутри каждого кластера можно дополнительно идентифицировать подтемы — например, «обработка ошибок», «структуры данных», «веб-фреймворки», что обеспечит сбалансированное представление всех языков и подтем в корпусе.

Таким образом, это гарантирует полноценный охват каждой темы в контексте каждого языка программирования.

Для кластеризации тем существует множество инструментов, каждый из которых основан на собственных подходах и обладает своими преимуществами. Например, библиотека `text-clustering` от Hugging Face предлагает простой пайплайн, в рамках которого пакет `SentenceTransformer` используется для векторизации текста, UMAP — для понижения размерности и DBSCAN — для кластеризации. Данная библиотека также может автоматически присваивать метки кластерам с помощью LLM и создавать визуализации. Альтернативными вариантами, предоставляющими аналогичный функционал и ряд дополнительных возможностей, являются Nomic Atlas (рис. 5.5), BunkaTopics и Lilac.

---

[1] Означают типичные формулировки для данного контекста. Например, в медицинских текстах часто пишут: «Пациент жалуется на...» — *Примеч. науч. ред.*

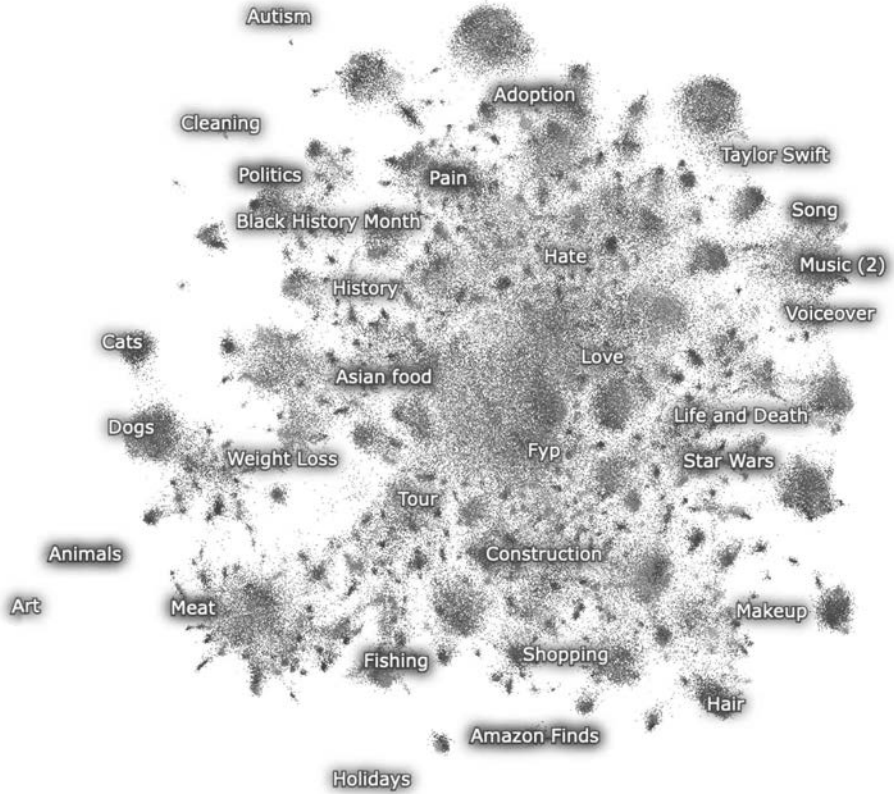


Рис. 5.5. Визуальное представление набора данных Historical TikTok, созданное с помощью инструмента Nomic Atlas

Генерация данных

Когда доступных наборов данных с инструкциями оказывается недостаточно, возникает необходимость в создании собственных данных. Это особенно актуально при решении специализированных задач, для которых трудно найти подходящие открытые источники. Кроме того, генерация данных позволяет восполнить пробелы в слабо представленных областях выборки, скажем добавить в набор, посвященный JavaScript, примеры обработки ошибок. Хотя такие данные можно собирать вручную или с помощью краудсорсинга, подобные подходы, как правило, требуют значительных затрат времени и ресурсов. Генерация синтетических данных с использованием LLM представляет собой более эффективную и масштабируемую альтернативу. В сочетании с грамотным про-

ектированием промптов (prompt engineering) такая техника позволяет получать данные высокого качества в гораздо больших объемах, устраняя ограничения, присущие ручному процессу генерации.

Генерация синтетических данных, как правило, начинается с подготовки набора тщательно сформулированных промптов (иногда называемого таксономией). Именно они служат основой для создания новых разнообразных примеров. Пять исходных промптов, использованных в наборе данных Alpaca, приведены ниже. Качество синтетических данных во многом зависит от промптов и выбранной методики генерации. Хорошо составленные промпты направляют языковую модель, способствуя формированию разнообразных, релевантных и качественных пар «инструкция — ответ». Такие промпты обычно включают конкретные указания, примеры и ограничения, гарантирующие соответствие сгенерированных данных заданному формату и тематике.

Примеры исходных промптов из оригинального набора данных Alpaca

Инструкции

- Порекомендуй вариант завтрака без яиц, с высоким содержанием белка и общей калорийностью от 700 до 1000 ккал.
- Найди логическое отношение между заданными парами. Вход: Ночь : День :: Право : Лево
- Сформулируй одно предложение, описывающее каждого из следующих людей. Вход: – Барак Обама\n – Илон Маск\n – Тейлор Свифт
- Опиши ситуацию, в которой заданный стереотип может тебе навредить. Вход: Все азиаты — умные!
- Придумай уместный заголовок для следующего электронного письма. Вход: «Здравствуйте, [имя],\n\nЯ пишу, чтобы узнать, готовы ли вы принять участие в нашей панельной дискуссии в рамках воркшопа по мультимодальности на конференции CVPR. Это мероприятие состоится 20 июня 2023 года.\n\nС уважением,\n[мое имя]»

Многие пайплайны генерации синтетических данных предусматривают несколько этапов, направленных на обеспечение их высокого качества. Как правило, сначала формируется исходный набор вопросов или инструкций, а затем — соответствующие ответы. В некоторых системах предусмотрены дополнительные стадии валидации, в ходе которых другая модель (или набор правил) проверяет сгенерированные пары на точность, релевантность и соответствие заданным критериям.

Важной особенностью генерации синтетических данных является возможность управлять различными характеристиками создаваемого контента, к которым относятся сложность инструкций, длина ответов, тональность или стиль

используемого языка, а также тематика или предметная область. Тонкая настройка этих параметров позволяет формировать наборы данных, адаптированные под конкретные обучающие цели либо дополняющие уже существующие. Структурированная генерация с использованием таких библиотек, как Outlines, также может быть полезна для соблюдения заданного формата.

Кроме того, генерация синтетических данных помогает устранить предвзятости и пробелы в существующих наборах. Тщательное проектирование этого процесса позволяет создавать сбалансированные и инклюзивные наборы, отражающие широкий спектр точек зрения, тематик и языковых стилей. Это способствует обучению языковых моделей, проявляющих большую степень справедливости и способных эффективно взаимодействовать с разнообразной аудиторией.

Тем не менее генерация синтетических данных сопряжена и с рядом сложностей. Одна из главных проблем — вероятность наследования предвзятости или ошибок, присущих языковой модели, используемой в процессе генерации. Для минимизации таких рисков применяются подходы, предполагающие участие человека, разнообразные промпты и дополнительные фильтры, гарантирующие качество и корректность создаваемых данных.

Еще одним важным аспектом является необходимость обеспечения достаточного разнообразия и сложности данных. Слишком простые или однообразные примеры не позволят обучить по-настоящему надежную модель. Продвинутые методы генерации синтетических данных ориентированы на создание разнообразных и детализированных пар «инструкция — ответ», которые способствуют обучению моделей, умеющих решать более сложные задачи.

Аугментация данных

В данном контексте под аугментацией данных понимается процесс увеличения как количества, так и качества обучающих примеров. В отличие от генерации данных на этом этапе в качестве исходных используются уже существующие пары «инструкция — ответ». Хотя количество таких пар можно было бы просто увеличить, основной целью аугментации является повышение качества имеющихся образцов. Особое внимание здесь уделяется двум аспектам: разнообразию и сложности.

Одним из инновационных подходов в этой области является метод Evol-Instruct, в рамках которого LLM используются для преобразования простых инструкций в более комплексные. Такие усовершенствованные инструкции затем служат основой для генерации ответов с использованием мощных языковых моделей. В основе этого метода лежат две стратегии: улучшение инструкций в глубину и в ширину.

Улучшение в глубину (in-depth evolving) направлено на повышение сложности уже имеющихся инструкций. Для реализации этой стратегии применяются следующие техники.

- *Добавление ограничений* — введение дополнительных требований или условий к исходной инструкции, усложняющих ее выполнение.
- *Углубление* — замена поверхностных вопросов на более содержательные, требующие развернутых и осмысленных ответов.
- *Конкретизация* — замена обобщенных понятий более точными и детализированными формулировками.
- *Увеличение числа шагов рассуждения* — модификация инструкции таким образом, чтобы она требовала поэтапного логического вывода.
- *Усложнение входных данных* — добавление к инструкции более сложных форматов и структур данных, таких как XML, JSON или фрагменты кода.

Улучшение в ширину (in-breadth evolving) направлено на увеличение разнообразия инструкций. Его суть заключается в генерации новых формулировок на основе уже существующих с акцентом на редкие или слабо представленные примеры из той же предметной области.

Примером практической реализации автоматизированного улучшения в глубину служит следующий промпт. Достаточно передать на вход исходную инструкцию, и мощная модель, такая как GPT-4o, возвратит ее более сложную версию.

Промпт Evol LLM из статьи Automatic Instruction Evolving for Large Language Models (Zeng et al., 2024)

Ты — редактор инструкций, задача которого состоит в преобразовании предоставленной #Инструкции# в более сложную версию. Пожалуйста, выполни перечисленные ниже шаги для усложнения #Инструкции#.

- Шаг 1. Внимательно прочитай #Инструкцию# и перечисли все возможные методы ее усложнения (цель — сделать ее более трудной для выполнения такими ИИ-ассистентами, как ChatGPT и GPT-4). Не включай методы, предполагающие изменение языка инструкции.
- Шаг 2. На основе полученного #Списка методов# составь подробный план усложнения #Инструкции#. План должен включать несколько методов из списка.
- Шаг 3. Выполни план шаг за шагом и представь результат в виде #Переписанной инструкции#. При этом в исходную #Инструкцию# можно добавить всего 10–20 слов.
- Шаг 4. Внимательно проверь #Переписанную инструкцию# на предмет наличия в ней не имеющих смысла фрагментов. Убедись в том, что #Переписанная инструкция# действительно представляет собой усложненную, но логически эквивалентную версию исходной #Инструкции#. В финальном ответе приведи только #Итоговую версию переписанной инструкции# без дополнительных пояснений.

Пожалуйста, используй следующий формат ответа:

Шаг 1 #Список методов#:

Шаг 2 #План#:

Шаг 3 #Переписанная инструкция#:

Шаг 4 #Итоговая версия переписанной инструкции#:

#Инструкция#:

{Инструкция}

Метод UltraFeedback представляет собой еще один инновационный подход, направленный на повышение качества ответов моделей с использованием обратной связи от ИИ. В отличие от Evol-Instruct, усложняющего инструкции, UltraFeedback опирается на множество разнообразных инструкций и моделей для генерации широкого спектра ответов.

Затем для оценки полученных ответов привлекаются продвинутые языковые модели вроде GPT-4, которые выполняют их критический разбор и присваивают численные оценки по нескольким критериям, таким как следование инструкции, достоверность, честность и полезность.

На основе этих идей можно разработать собственные методы аугментации для создания более комплексных и разнообразных наборов данных с инструкциями. Путем уточнения и усложнения существующих инструкций и ответов можно формировать наборы данных, позволяющие научить модели эффективнее решать сложные, многоэтапные задачи и повысить их качество работы в более широком спектре приложений.

Создание собственного набора данных с инструкциями

В этом разделе мы создадим собственный набор данных с инструкциями на основе материалов, собранных с помощью краулеров в главе 3. Чтобы получить качественный результат, нам необходимо преодолеть такие проблемы, как неструктурированность исходных данных и ограниченное количество доступных статей.

Неструктурированность данных объясняется тем, что мы имеем дело с сырым текстом (например, статьями), а не с парами «инструкция — ответ». Для преобразования таких данных мы будем использовать LLM. В частности, мы применим комбинацию обратного перевода и перефразирования. Под обратным переводом здесь понимается процесс, при котором на входе модели подается ожидаемый ответ и она генерирует соответствующую инструкцию. Однако

использование фрагмента текста в качестве ответа не всегда целесообразно. Поэтому мы будем перефразировать исходный текст, чтобы гарантировать соответствие ответов модели заданному формату. Кроме того, мы попросим модель придерживаться стиля автора исходного абзаца, чтобы сохранить близость к оригиналу. Хотя такой подход требует продуманного проектирования промптов, он вполне поддается автоматизации и масштабированию.

Вторая проблема, а именно ограниченное число доступных статей, довольно типична для практических сценариев. Небольшой объем извлекаемых материалов ограничивает размер создаваемого набора данных с инструкциями. В нашем случае чем больше примеров мы соберем, тем лучше модель сможет имитировать стиль авторов. Чтобы увеличить объем данных, мы разобьем статьи на фрагменты и для каждого из них сгенерируем по три пары «инструкция — ответ». Такой прием позволит приумножить число образцов при сохранении их тематического разнообразия. В рамках этого примера мы применим модель GPT-4o-mini от OpenAI, однако вы вполне можете задействовать модели с открытым исходным кодом.

Следует отметить, что LLM не всегда дают надежные результаты при генерации структурированных ответов. Даже если используются четкие шаблоны и инструкции, модель не всегда соблюдает заданный формат, что приводит к необходимости в дополнительном синтаксическом анализе строк.

Чтобы упростить этот процесс и обеспечить корректную структуру результатов, можно воспользоваться методами структурированной генерации, которые позволяют заставить модель придерживаться шаблона, например заданного с помощью формата JSON, классов Pydantic или регулярных выражений. В дальнейшем мы будем работать с режимом JSON, реализованным в системах OpenAI, который обеспечивает более надежную генерацию корректных JSON-объектов и снижает необходимость в постобработке.

Все этапы предлагаемого пайплайна генерации синтетических данных представлены на рис. 5.6.

Теперь реализуем этот процесс на языке Python. Вы можете встроить его в существующий LLMOps-пайплайн или оформить в виде отдельного скрипта.

1. Первым делом необходимо убедиться в том, что у вас установлены все нужные библиотеки. Библиотека `openai` позволит взаимодействовать с языковой моделью для генерации набора инструкций, `datasets` — преобразовывать результат в формат, совместимый с Hugging Face, а `tqdm` — визуализировать прогресс генерации.

```
openai==1.37.1
datasets==2.20.0
tqdm==4.66.4
```

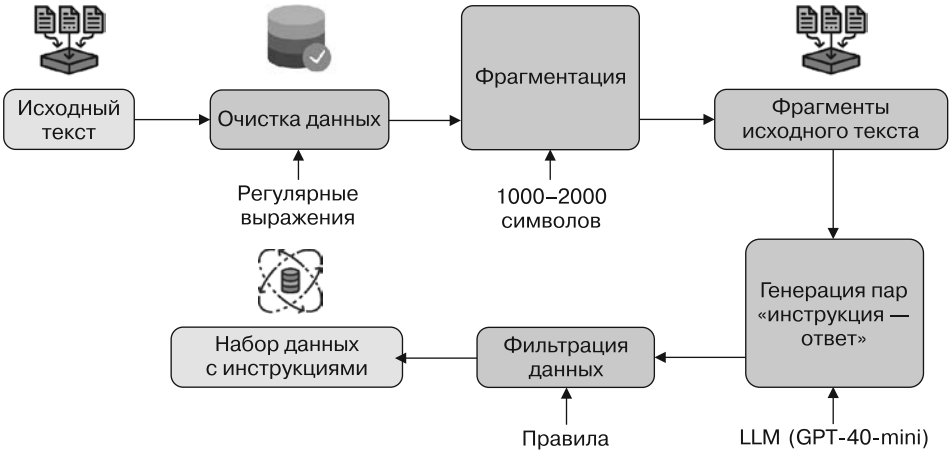


Рис. 5.6. Пайплайн генерации синтетических данных

2. Теперь импортируем все необходимые библиотеки.

```
import concurrent.futures
import json
import random
import re
from concurrent.futures import ThreadPoolExecutor

from typing import List, Tuple
from datasets import Dataset
from openai import OpenAI
from pydantic import BaseModel, Field
from tqdm.auto import tqdm
```

3. Исходные данные представлены в виде JSON-файла. Мы создаем на его основе набор данных Hugging Face, извлекая из каждой статьи следующие поля: `id`, `content`, `platform`, `author_id`, `author_name`, `link`.

```
def load_articles_from_json(file_path: str) -> Dataset:
    with open(file_path, "r") as file:
        data = json.load(file)

    return Dataset.from_dict(
        {
            "id": [item["id"] for item in data["artifact_data"]],
            "content": [item["content"] for item in data["artifact_data"]],
            "platform": [item["platform"] for item in data["artifact_data"]],
            "author_id": [item["author_id"] for item in data["artifact_data"]],
            "author_full_name": [item["author_full_name"] for item in
            ➥data["artifact_data"]],
            "link": [item["link"] for item in data["artifact_data"]],
        }
    )
```

Если мы просто загрузим этот набор в виде таблицы данных pandas, то получим следующую структуру.

	id	content	platform	author_id	author_full_name	link
0	ab2f9e2e-5459-4dd6-97d6-c291de4a7093	The Importance of Data Pipelines in the Era of...	medium	e6b945ba-6a9a-4cde-b2bf-0890af79732b	Alex Vesa	https://medium.com/decodingml/the-importance-o...
1	ccfe70f3-d324-40b6-ba38-86e72786dcf4	Change Data Capture: Enabling Event-Driven Arc...	medium	e6b945ba-6a9a-4cde-b2bf-0890af79732b	Alex Vesa	https://medium.com/decodingml/the-3nd-outof-1...
2	4c9f68ae-ec8b-4534-8ad5-92372bf8bb37	The Role of Feature Stores in Fine-Tuning LLMs...	medium	e6b945ba-6a9a-4cde-b2bf-0890af79732b	Alex Vesa	https://medium.com/decodingml/the-role-offeat...
...	...	...	...	...	...	...
73	68795a4d-26c2-43b7-9900-739a80b9b7dc	DML: 4 key ideas you must know to train an LLM...	decodingml.substack.com	1519b1d1-1a5d-444c-a880-926c9eb6539e	Paul Iusztin	https://decodingml.substack.com/p/dml-4-key-id...
74	d91b17c0-05d8-4838-bf61-e2abc1573622	DML: How to add real-time monitoring & metrics...	decodingml.substack.com	1519b1d1-1a5d-444c-a880-926c9eb6539e	Paul Iusztin	https://decodingml.substack.com/p/dml-how-to-a...
75	dcf55b28-2814-4480-a18b-a77d014d4f5f	DML: Top 6 ML Platform Features You Must Know ...	decodingml.substack.com	1519b1d1-1a5d-444c-a880-926c9eb6539e	Paul Iusztin	https://decodingml.substack.com/p/dml-top-6-ml...

4. При проверке содержимого статей заметно, что в некоторых из них встречаются специальные символы и избыточные пробелы, которые можно удалить с помощью простого регулярного выражения.

   Сначала мы используем выражение `[^\w\s.,!?']`, чтобы удалить все символы, кроме буквенно-цифровых, апострофов, точек, запятых, восклицательных

и вопросительных знаков. Затем мы используем выражение `\s+`, чтобы заменить несколько идущих подряд пробельных символов одним пробелом.

Наконец, мы применяем функцию `strip()`, чтобы удалить пробелы в начале и в конце строки.

```
def clean_text(text):
    text = re.sub(r"[^\w\s.,!?']", " ", text)
    text = re.sub(r"\s+", " ", text)
    return text.strip()
```

5. После загрузки статей необходимо разбить их на фрагменты перед преобразованием в пары «инструкция — ответ». В идеале для фрагментации следует использовать заголовки или абзацы, чтобы обеспечить семантически осмысленное разбиение текста.

   Однако, как и в большинстве реальных сценариев, в нашем случае сырые данные плохо структурированы. Из-за неадекватного форматирования извлечь заголовки и абзацы из всех статей невозможно. Вместо этого мы разобьем тексты на фрагменты с помощью регулярных выражений, выделяя предложения и объединяя их в блоки размером от 1000 до 2000 символов. Данные значения можно скорректировать в зависимости от плотности информации, содержащейся в тексте.

   Функция `extract_substrings` очищает текст каждой статьи и затем разбивает его на предложения. Далее она конкатенирует эти предложения вплоть до достижения заданной длины фрагмента (1000–2000 символов).

```
def extract_substrings(dataset: Dataset, min_length: int = 1000,
➥max_length: int = 2000) -> List[str]:
    extracts = []
    sentence_pattern = r"(?<!\w\.\w.)(?<![A-Z][a-z]\.)(?<=\.|\?|\!)\s"

    for article in dataset["content"]:
        cleaned_article = clean_text(article)
        sentences = re.split(sentence_pattern, cleaned_article)

        current_chunk = ""
        for sentence in sentences:
            sentence = sentence.strip()
            if not sentence:
                continue

            if len(current_chunk) + len(sentence) <= max_length:
                current_chunk += sentence + " "
            else:
                if len(current_chunk) >= min_length:
                    extracts.append(current_chunk.strip())
                current_chunk = sentence + " "

        if len(current_chunk) >= min_length:
            extracts.append(current_chunk.strip())

    return extracts
```

6. Теперь нам нужно сформировать пары «инструкция — ответ» на основе полученных текстовых фрагментов. Для эффективного управления этими парами мы создаем класс `InstructionAnswerSet`, позволяющий создавать экземпляры непосредственно из JSON-строк, что особенно полезно при парсинге результатов, возвращаемых API OpenAI.

```
class InstructionAnswerSet:
    def __init__(self, pairs: List[Tuple[str, str]]):
        self.pairs = pairs

    @classmethod
    def from_json(cls, json_str: str) -> 'InstructionAnswerSet':
        data = json.loads(json_str)
        pairs = [(pair['instruction'], pair['answer'])
            for pair in data['instruction_answer_pairs']]
        return cls(pairs)

    def __iter__(self):
        return iter(self.pairs)
```

7. Теперь, когда у нас есть фрагменты статей подходящей длины, можно использовать LLM для преобразования каждого из них в пару «инструкция — ответ». Причем для этого допускается использовать модель как с открытым, так и с закрытым исходным кодом. Поскольку вывод формируется строго на основе предоставленного контекста, для решения нашей задачи не требуются сложное рассуждение или высокопроизводительные модели.

   В данном примере мы используем экономичную и достаточно эффективную модель GPT-4o mini. На текущем этапе наибольшее значение имеет проектирование промптов, и для получения ожидаемого результата, как правило, требуется несколько итераций. Мы рекомендуем начинать с простых промптов, а при необходимости постепенно увеличивать их сложность, изменять стиль или просить сгенерировать несколько ответов.

   В примере мы хотим создавать инструкции вида «Напиши абзац на тему X» и соответствующие ответы, содержащие достоверную информацию и имитирующие стиль конкретного автора. Для этого нужно предоставить фрагмент текста, который модель возьмет за основу при генерации ответов. Для большей эффективности мы будем генерировать по пять пар «инструкция — ответ» на каждый фрагмент. Ниже приведено начало функции генерации инструкций вместе с используемым промптом.

```
def generate_instruction_answer_pairs(
    extract: str, client: OpenAI
) -> List[Tuple[str, str]]:
    prompt = f"""На основе следующего фрагмента сгенерируй пять пар
«инструкция — ответ». Каждая инструкция должна предлагать написать текст
на определенную тему, содержащуюся в контексте. Каждый ответ должен содержать
соответствующий абзац, основанный на информации из контекста.
Используй для генерации инструкций только те понятия, которые есть в контексте.
Инструкции ни в коем случае не должны явно упоминать контекст, систему,
```

```
курс или фрагмент. Инструкции должны быть самодостаточными и общими. Ответы
должны имитировать стиль написания, характерный для контекста.
Пример инструкции: объясни понятие LLM-двойника (LLM Twin).
Пример ответа: LLM-двойник — это, по сути, ИИ-персонаж, который имитирует
ваш стиль письма, личность и голос.
Он создан, чтобы писать так же, как вы, интегрируя эти элементы в языковую
модель.
Идея заключается в создании цифровой копии ваших писательских привычек
с помощью передовых методов ИИ.
Предоставь ответ в формате JSON со следующей структурой:
    {{
        "instruction_answer_pairs": [
            {{"instruction": "...", "answer": "..."}},
            ...
        ]
    }}

    Фрагмент:
    {extract}
    """
```

8. В дополнение к пользовательскому промпту можно также задать системный промпт, который будет направлять модель в процессе генерирования ожидаемых инструкций.

   Результат конкатенации системного и пользовательского промптов передается в API OpenAI, использующий модель GPT-4o-mini в режиме JSON, с ограничением на длину ответа в 1200 токенов. Для обеспечения разнообразия ответов мы возьмем стандартное значение параметра `temperature = 0.7`. Сгенерированный ответ незамедлительно разбирается с помощью класса `InstructionAnswerSet`, возвращающего итоговые пары «инструкция — ответ».

```
    completion = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {
                "role": "system", "content": "Ты — полезный ассистент, который
                 генерирует пары «инструкция — ответ» на основе заданного
                 контекста. Предоставь ответ в формате JSON.",
            },
            {"role": "user", "content": prompt},
        ],
        response_format={"type": "json_object"},
        max_tokens=1200,
        temperature=0.7,
    )

    # Разбор структурированного вывода
    result = InstructionAnswerSet.from_json(completion.choices[0].
    ➥message.content)

    # Преобразование в список кортежей
    return result.pairs
```

9. Теперь создадим основную функцию для автоматизации всего процесса. Она должна извлекать подстроки из исходного набора данных, а затем с помощью средств конкурентной обработки на базе класса Python `ThreadPoolExecutor` генерировать пары «инструкция — ответ».

   Мы используем стандартное значение параметра `max_workers = 4`, поскольку более высокие могут привести к превышению лимитов API OpenAI и вызвать ошибки при выполнении запросов или замедление их обработки.

```
def create_instruction_dataset(
    dataset: Dataset, client: OpenAI, num_workers: int = 4
) -> Dataset:
    extracts = extract_substrings(dataset)
    instruction_answer_pairs = []
    with concurrent.futures.ThreadPoolExecutor(max_workers=num_workers)
    ➥as executor:
        futures = [executor.submit(generate_instruction_answer_pairs,
        ➥extract, client)
            for extract in extracts
        ]
        for future in tqdm(concurrent.futures.as_completed(futures),
        ➥total=len(futures)
        ):

            instruction_answer_pairs.extend(future.result())

    instructions, answers = zip(*instruction_answer_pairs)
    return Dataset.from_dict(
        {"instruction": list(instructions), "output": list(answers)}
    )
```

10. Вызов этой функции позволяет нам сформировать набор инструкций. Генерация такого набора на основе сырых данных с использованием модели GPT-4o-mini обходится менее чем в $0,50.

11. Теперь можно создать функцию, которая будет координировать работу всего пайплайна. Она должна загружать исходные данные, формировать набор инструкций, разбивать его на обучающую и тестовую (10 % от общего количества) выборки, а затем отправлять результат на платформу Hugging Face Hub.

```
def main(dataset_id: str) -> Dataset:
    client = OpenAI()

    # 1. Загрузка исходных данных
    raw_dataset = load_articles_from_json("cleaned_documents.json")
    print("Raw dataset:")
    print(raw_dataset.to_pandas())

    # 2. Создание набора данных с инструкциями
    instruction_dataset = create_instruction_dataset(raw_dataset, client)
        print("Instruction dataset:")
        print(instruction_dataset.to_pandas())
```

```
    # 3. Разбиение на обучающую и тестовую выборки и экспорт
    filtered_dataset = instruction_dataset.train_test_split(test_size=0.1)
    filtered_dataset.push_to_hub("mlabonne/llmtwin")

    return filtered_dataset

Dataset({
    features: ['instruction', 'output'],
    num_rows: 3335
})
```

В результате этого процесса было получено 3335 пар «инструкция — ответ». Итоговый набор данных доступен по адресу https://huggingface.co/datasets/mlabonne/llmtwin. Платформа Hugging Face Hub предоставляет удобный интерфейс для просмотра данных (рис. 5.7), который позволяет проверить инструкции и ответы на наличие очевидных ошибок. Ввиду ограниченного объема набора данных полноценная кластеризация тем и углубленный анализ нам не требуются.

instruction string · lengths 57–67 28.2%	output string · lengths 561–628 21.7%
Explain how list comprehensions improve iteration speed in Python.	List comprehensions serve as a highly efficient mechanism for iterating over lists in Python, enabling operations to be expressed in a single line. This method eliminates the need to explicitly call the .append method during iteration, which contributes to its speed. Python processes list comprehensions similarly to traditional for loops at the bytecode level, yet the overhead of function calls is reduced, resulting in a performance boost. As evidenced by benchmarks, list comprehensions can achieve speeds that are 200 times faster than conventional looping methods.
Define what an LLM Twin is and its purpose.	An LLM Twin is an AI-driven character designed to replicate your own writing style, personality, and voice. By leveraging large language…
Describe how audio narrations can enhance the reading experience.	Audio narrations can significantly enhance the reading experience by adding an auditory dimension to the text. This feature allows readers t…
Describe the role of a vector database in embedding storage.	A vector database serves as a crucial repository for storing embeddings along with their associated metadata, such as the embedded text, source…
Explain the concept of quantization in the context of LLMs.	Quantization in the context of LLMs refers to the process of reducing the precision of the numbers used to represent the model's weights. Thi…

Рис. 5.7. Обучающий набор с инструкциями mlabonne/llmtwin на платформе Hugging Face Hub

Как было сказано ранее, этот набор данных можно доработать, повысив разнообразие и сложность примеров. Более продвинутая настройка промптов также способствует улучшению качества сгенерированных данных, в частности, за счет включения примеров ожидаемых результатов. Наконец, оценка качества позволяет отфильтровать неудачные образцы путем их индивидуального анализа. Однако, исходя из соображений краткости и наглядности, в рамках текущего примера мы придерживаемся относительно простого подхода. Более продвинутые методы будут рассмотрены в главе 6, где мы займемся созданием набора данных о предпочтениях.

В следующем разделе мы познакомимся с техниками тонкой настройки с учителем и связанными с ней понятиями.

Тонкая настройка с учителем и ее методы

Тонкая настройка с учителем (SFT) предполагает калибровку предварительно обученной модели с помощью небольшого набора пар «инструкция — ответ». Цель SFT заключается в преобразовании базовой модели, способной лишь предсказывать следующий токен, в полезного ассистента, умеющего отвечать на вопросы и следовать инструкциям. Такая тонкая настройка также может применяться для повышения общей производительности базовой модели (универсальная тонкая настройка), внедрения новых знаний (например, языков или предметных областей), адаптации под решение конкретных задач, задания определенного стиля речи и не только.

В этом разделе мы обсудим ситуации, в которых целесообразно применять тонкую настройку, а также рассмотрим связанные с ней понятия, включая форматы хранения данных для дообучения и шаблоны диалогов. Кроме того, мы познакомимся с тремя популярными подходами к реализации SFT, такими как полная тонкая настройка (full fine-tuning), *LoRA* (Low-Rank Adaptation — «низкоранговая адаптация») и *QLoRA* (Quantization-aware LoRA — «низкоранговая адаптация с учетом квантования»).

Когда стоит проводить тонкую настройку

В большинстве случаев рекомендуется начинать не с тонкой настройки, а с проектирования промптов. Причем такой подход применим как к моделям с открытыми весами, так и к моделям с закрытым исходным кодом. Используя методы вроде обучения на малом числе примеров (few-shot) или *генерации, дополненной поиском* (RAG), можно эффективно решать широкий круг задач без необходимости в SFT. Кроме того, проектирование промптов позволяет построить надежный пайплайн оценки, в рамках которого измеряются не только точность ответов модели, но и стоимость, а также задержка отклика. Если результаты не соответствуют требованиям, можно подумать о создании набора данных с инструкциями, как описано в предыдущем разделе. При наличии достаточного объема данных тонкая настройка становится реальной альтернативой (рис. 5.8).

Помимо решения технических задач, тонкая настройка с учителем позволяет удовлетворить потребности, связанные с контролем над данными (мы точно знаем, какие данные туда вошли) и возможностью кастомизации (дообученная модель становится уникальной, адаптированной под конкретные задачи). Вместо того чтобы строить приложение вокруг чат-бота, SFT позволяет создавать более разнообразные сценарии взаимодействия с языковыми моделями, включая аналитику, модерацию и поддержку дополнительного контекста. Хотя

в книге основное внимание уделяется моделям с открытыми весами, следует отметить, что ряд провайдеров LLM предоставляет услуги автоматической тонкой настройки. Такие решения обеспечивают меньший уровень контроля и кастомизации по сравнению с самостоятельной организацией пайплайна тонкой настройки, однако в определенных случаях (например, при ограниченных ресурсах команды ML-инженеров) могут представлять собой разумный компромисс.

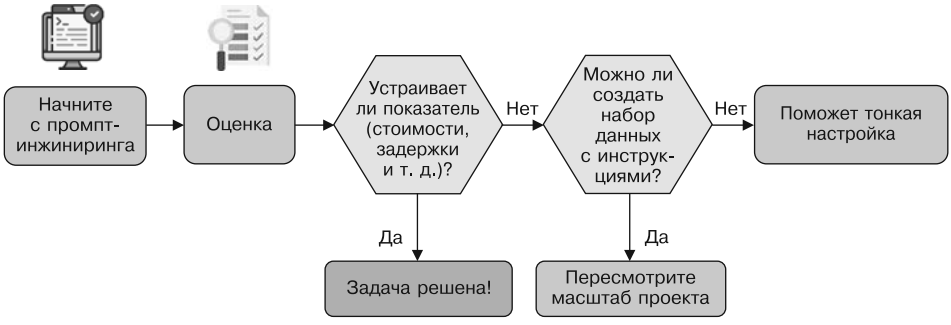


Рис. 5.8. Базовая блок-схема, помогающая оценить техническую целесообразность тонкой настройки модели

Несмотря на все преимущества, тонкая настройка имеет и свои ограничения. В общем случае SFT предполагает использование уже имеющихся знаний, содержащихся в весах базовой модели, и калибровку параметров под решение конкретной задачи. Это влечет за собой ряд последствий. Прежде всего, знания для дообучения, находящиеся слишком далеко от тематики набора, задействованного на этапе предварительного обучения (например, редкие или незнакомые языки), усваиваются моделью с трудом.

Более того, согласно одному из исследований, тонкая настройка модели на новых знаниях может привести к учащению галлюцинаций. И в зависимости от применяемой техники SFT существует риск так называемого катастрофического забывания, то есть утраты знаний, ранее присутствовавших в базовой модели.

Форматы наборов данных с инструкциями

Обучающие наборы данных с инструкциями сохраняются в определенном формате, структурирующем инструкции и ответы. Обычно каждый пример в наборе представляет собой словарь Python, где ключами являются типы частей промпта (например, `system`, `instruction`, `output`), а значениями — соответствующие

тексты. Наиболее распространенными форматами являются: Alpaca, ShareGPT и OpenAI. В табл. 5.2 показаны их типовые структуры.

Таблица 5.2. Примеры форматов хранения наборов данных с инструкциями

Название	Формат JSONL
Alpaca	`{"instruction": "...", "input": "...", "output": "..."}` `{"instruction": "...", "output": "..."}`
ShareGPT	`{"conversations": [{"from": "...", "value": "..."}, ...]}`
OpenAI	`{"conversations": [{"role": "...", "content": "..."}, ...]}`
OASST	`{"INSTRUCTION": "...", "RESPONSE": "..."}`
Сырой текст	`{"text": "..."}`

Обратите внимание, что в формате Alpaca ключ `"input"` является необязательным. Если он присутствует, его содержимое просто добавляется к значению `"instruction"`. Мы включили в таблицу формат «сырой текст», чтобы показать, что тонкая настройка с учителем концептуально не отличается от предварительного обучения. Тонкая настройка модели на сырых текстах обычно называется *непрерывным предварительным обучением* (Continual Pretraining, CPT).

Набор данных, созданный в предыдущем разделе, предусматривает два столбца — `"instruction"` и `"output"` — и соответствует формату Alpaca, который подходит для организации одиночных взаимодействий типа «одна инструкция — один ответ». Если вам нужны диалоги, предполагающие многократный обмен репликами, лучше использовать форматы ShareGPT или OpenAI, которые позволяют сохранить беседу произвольной длины, представляя каждое сообщение в виде словаря внутри списка.

Ваш выбор между этими взаимодействиями напрямую влияет на формат хранения данных и зависит от целевого применения модели.

Шаблоны ведения диалога

После разбора пар «инструкция — ответ» из набора данных их необходимо привести к шаблону ведения диалога. Такие шаблоны обеспечивают единый способ подачи инструкций и ответов на вход модели.

Обычно шаблон включает специальные токены, обозначающие начало и конец сообщений, а также указывающие на их автора. Поскольку базовые модели

не предназначены для следования инструкциям, они не имеют встроенного шаблона ведения диалога. Это означает, что при тонкой настройке такой модели вы можете задействовать любой шаблон. Если же вы тонко настраиваете модель с инструкциями (что, как правило, не рекомендуется), то важно использовать тот же шаблон, который применялся на этапе ее предварительного обучения, иначе вы рискуете понизить уровень ее производительности.

Как и в случае с форматами наборов данных с инструкциями, существует множество шаблонов ведения диалога: ChatML, Llama 3, Mistral, Gemma и не только. В сообществе разработчиков и пользователей ПО с открытым исходным кодом особенно популярен ChatML (разработанный OpenAI). Он использует специальные токены `<|im_start|>` и `<|im_end|>`, чтобы указать на автора (роль) и его сообщение. Далее приведен пример пары «инструкция — ответ» из табл. 5.1 в формате ChatML.

Пример из табл. 5.1, оформленный с использованием шаблона ChatML

<|im_start|>system

Ты — полезный помощник, способный объяснить все что угодно. Представь, что отвечаешь пятилетнему ребенку.<|im_end|>

<|im_start|>user

Понятия: здание, магазин, город/

Составь предложение, в котором используются все эти слова.<|im_end|>

<|im_start|>assistant

В нашем городе есть магазин внутри большого здания, куда люди приходят для того, чтобы купить любимые игрушки и конфеты.<|im_end|>

Как видите, структура по-прежнему состоит из трех частей: `system`, `user`, `assistant`. Каждая часть начинается с токена `<|im_start|>` и заканчивается `<|im_end|>`. Автор фразы задается строкой (например, `"system"`), которая токенизируется и передается модели в качестве входных данных на стадии тонкой настройки.

Однако на этапе инференса мы не можем предоставить ожидаемый ответ. В таком случае мы подаем только части `system` и `user` и побуждаем модель сгенерировать ответ, добавляя токен: `<|im_start|> assistant\n`.

Поскольку модель была тонко настроена с использованием этого шаблона, она понимает, что следующим должен быть ответ, согласующийся с пользовательской инструкцией и контекстом системного промпта. Именно так модели обучаются следовать инструкциям.

В табл. 5.3 приведены примеры подобных шаблонов, включая Alpaca, который является одновременно форматом набора данных с инструкциями и шаблоном ведения диалога.

Таблица 5.3. Примеры популярных шаблонов ведения диалога

Название	Шаблон Jinja

Alpaca	`### Инструкция: Какой город является столицей Франции?`
	
	`### Ответ: Столицей Франции является Париж.<EOS>`

ChatML	`<|im_start|>user`
	`Какой город является столицей Франции?<|im_end|>`
	`<|im_start|>assistant`
	`Столицей Франции является Париж.<|im_end|>`

Llama 3	`<|begin_of_text|><|start_header_id|>user<|end_header_id|>`
	
	`Какой город является столицей Франции?<|eot_id|><|start_header_id|>assistant<|end_header_id|>`
	
	`Столицей Франции является Париж.<|eot_id|>`

Phi-3	`<|user|>`
	`Какой город является столицей Франции?<|end|>`
	`<|assistant|>`
	`Столицей Франции является Париж.<|end|>`

Gemma	`<bos><start_of_turn>user`
	`Какой город является столицей Франции?<end_of_turn>`
	`<start_of_turn>model`
	`Столицей Франции является Париж.<end_of_turn>`

Шаблонизатор Jinja реализует циклы и условия, что позволяет использовать один и тот же шаблон как для обучения, так и для инференса (`add_generation_prompt`).

Эффективные методы тонкой настройки параметров

Несмотря на многообразие техник тонкой настройки с учителем (SFT), на практике чаще всего применяются три основных подхода: полная тонкая настройка, LoRA и QLoRA (рис. 5.9). Далее мы кратко опишем каждый из них и обсудим их преимущества и ограничения в зависимости от предполагаемого сценария использования.

D_{in}, D_{int} и D_{out} — размерности входного, промежуточного и выходного представлений соответственно.

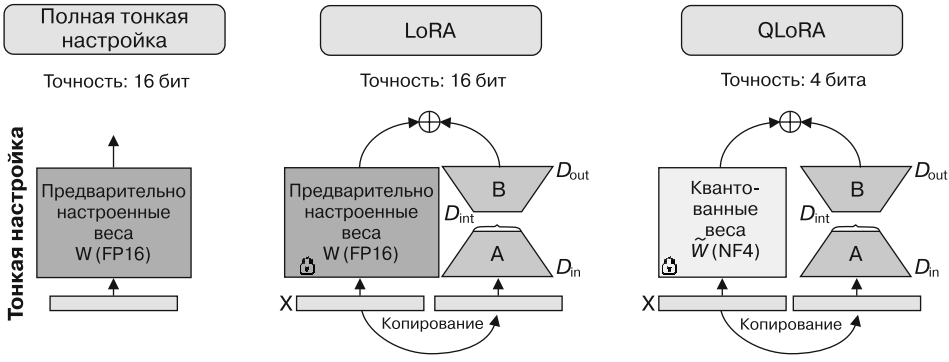


Рис. 5.9. Архитектурные различия трех основных техник тонкой настройки на уровне модулей

Полная тонкая настройка

Полная тонкая настройка (full fine-tuning) представляет собой самый прямолинейный метод SFT, предполагающий повторное обучение всех параметров базовой модели. Как и в случае с предварительным обучением, методы SFT направлены на предсказание следующего токена. Это означает, что главное различие между непрерывным предварительным обучением и полной тонкой настройкой заключается не в самой задаче, а в структуре используемого набора данных.

Данный метод нередко обеспечивает наилучшие результаты, но требует значительных вычислительных ресурсов. Объем потребляемой памяти зависит от множества факторов, включая размер модели, особенности применяемой техники обучения и методы оптимизации. В простейшем случае (с использованием одного графического процессора) объем необходимой памяти можно оценить с помощью следующей формулы:

$$\text{объем памяти} = \text{параметры} + \text{градиенты} + \text{состояния оптимизатора} + \\ + \text{активации}.$$

Для базовой настройки с использованием *32-битного формата с плавающей запятой (FP32)* объем потребляемой памяти можно приблизительно оценить следующим образом.

- *Параметры* — обучаемые веса и смещения нейросети. В больших языковых моделях это, как правило, веса в механизмах внимания, слоях прямого распространения и векторизации. Оценка затрат: 4 байта на параметр (FP32) или 2 байта на параметр (FP16/BF16).

- *Градиенты* — частные производные функции потерь по каждому параметру модели. Они показывают, насколько и в каком направлении следует изменить параметры для минимизации ошибки. Во время обучения градиенты вычисляются для каждого параметра методом обратного распространения ошибки и используются для обновления весов. Оценка затрат: 4 байта на параметр.
- *Состояния оптимизатора* — дополнительные значения, поддерживаемые оптимизаторами вроде Adam или AdamW. Обычно включают скользящие средние прошлых градиентов и квадратов градиентов, позволяя адаптировать скорость обучения для всех параметров. Например, алгоритм Adam хранит два дополнительных значения для каждого параметра (импульс и дисперсию). Оценка затрат: 8 байт на параметр (для оптимизатора Adam).
- *Активации* — промежуточные выходы каждого слоя нейросети во время прямого прохода. В трансформерах это выходы механизмов внимания, слоев прямого распространения и нормализации. Активации необходимо сохранять в памяти во время прямого прохода, чтобы затем использовать их при вычислении градиентов в процессе обратного прохода. Однако этого можно избежать, применив метод сохранения контрольных точек активаций. Оценка затрат: переменная, но обычно незначительная при небольших размерах пакетов.

Таким образом, приблизительный объем потребляемой памяти равен 16 байтам на один параметр, что в пересчете на видеопамять составляет 112 Гбайт для модели с 7 млрд параметров и 1120 Гбайт для модели с 70 млрд. Однако эти значения часто оказываются заниженными, поскольку в них не учитываются дополнительные затраты памяти и издержки, связанные с применением различных методов обучения.

Существует ряд техник, позволяющих снизить потребление памяти при тонкой настройке LLM. Например, параллелизм модели (model parallelism) распределяет вычислительную нагрузку между несколькими графическими процессорами, хотя и сопряжен с определенными накладными расходами. Накопление градиентов (gradient accumulation) позволяет эффективно использовать более крупные пакеты данных без пропорционального увеличения объема затрачиваемой памяти. Оптимизаторы с низким потреблением памяти, такие как 8-битный Adam, позволяют снизить объем памяти, необходимой для хранения состояний оптимизатора. Сохранение контрольных точек активаций (activation checkpointing) помогает сэкономить память за счет повторного вычисления некоторого количества активаций. В совокупности эти техники могут существенно сократить объем используемой памяти. Например,

комбинирование таких методов, как обучение со смешанной точностью (mixed precision) и модельный параллелизм, позволяет сократить затраты с исходных 16 байт на параметр до 14–15 байт. Тем не менее даже с учетом оптимизации требования к памяти при работе с крупными моделями остаются весьма значительными.

Кроме того, полная тонкая настройка напрямую модифицирует веса модели, полученные на этапе предварительного обучения, что делает данный процесс деструктивным по своей природе. Если обучение пойдет не так, как ожидалось, то это может привести к «катастрофическому забыванию» — утрате моделью ранее приобретенных знаний и навыков. Та же проблема рискует возникнуть и при непрерывном предварительном обучении. Именно по этой причине, а также из-за высокой вычислительной стоимости полной тонкой настройки во многих случаях предпочтение отдается техникам настройки параметров, особенно при создании моделей, ориентированных на конкретные задачи или предметные области.

LoRA

LoRA (Low-Rank Adaptation — «низкоранговая адаптация») — это эффективная техника тонкой настройки больших языковых моделей. Этот метод был разработан для преодоления вычислительных сложностей, связанных с дообучением массивных нейросетей, и быстро стал одним из ключевых инструментов тонкой настройки LLM.

Основная цель LoRA заключается в том, чтобы обеспечить возможность тонкой настройки LLM при значительно меньших вычислительных затратах за счет добавления обучаемых низкоранговых[1] матриц, которые изменяют поведение модели, не затрагивая ее исходные параметры. Основными преимуществами LoRA являются:

- существенное снижение объема памяти, используемой во время обучения;
- повышение скорости тонкой настройки;
- сохранение предварительно настроенных весов модели (метод не является деструктивным);
- возможность быстро переключаться между задачами путем подстановки различных весов LoRA.

Эти преимущества делают метод LoRA особенно привлекательным вариантом для исследователей и разработчиков, располагающих ограниченными вычислительными ресурсами, и способствуют демократизации процесса тонкой настройки LLM.

---

[1] Компактные матрицы, которые используют мало памяти. — *Примеч. науч. ред.*

В основе LoRA лежит метод низкорангового разложения, позволяющий эффективно обновлять веса модели. Вместо непосредственного изменения исходной весовой матрицы W LoRA вводит две обучаемые матрицы — A и B, которые вместе формируют низкоранговую поправку к W (рис. 5.10).

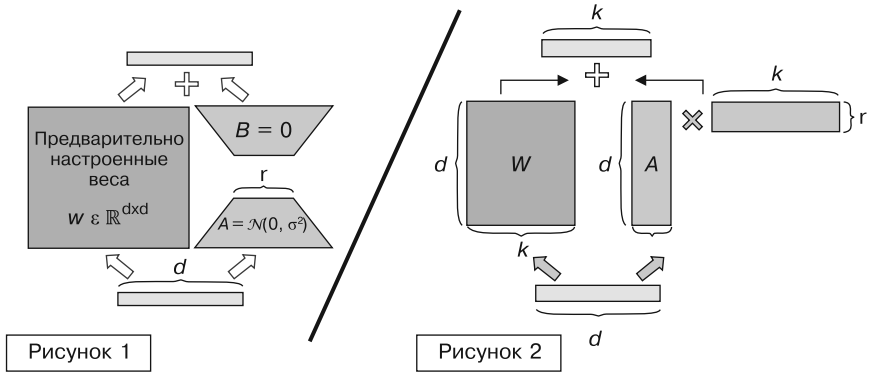


Рис. 5.10. Метод LoRA добавляет обучаемые матрицы A и B, оставляя предварительно настроенные веса модели замороженными

Математически это можно выразить следующим образом:

$$W' = W + BA,$$

где W — это исходная весовая матрица, B и A — обучаемые LoRA-матрицы, а W' — эффективная матрица весов, используемая моделью на этапе инференса.

Размерности матриц A и B подбираются так, чтобы их произведение имело ту же форму, что и матрица W, но существенно меньший ранг *(r)*. Этот ранг является ключевым гиперпараметром LoRA. Во время обучения исходные веса W остаются замороженными, а обновляются только матрицы A и B, что позволяет значительно сократить число обучаемых параметров, уменьшив потребление памяти и ускорив процесс обучения.

Для эффективной реализации метода LoRA необходимо правильно выбрать гиперпараметры и целевые модули. Двумя основными гиперпараметрами являются:

- *ранг (r)* — определяет размерность LoRA-матриц. Распространенной отправной точкой считается значение $r = 8$, однако в ряде случаев удалось получить хорошие результаты при использовании значений вплоть до 256. Большие значения ранга позволяют охватить более широкий спектр задач, но могут привести к переобучению модели;
- *альфа* (α) — коэффициент масштабирования обновления (показывает, как сильно новые матрицы влияют на замороженные веса). На практике

это означает, что к замороженным весам W добавляется обновление, умноженное на α/r. По этой причине распространенным подходом является использование значения α, вдвое превышающего значение r, что эквивалентно применению к обновлению коэффициента масштабирования, равного 2. При переобучении или недостаточном обучении модели рекомендуется поэкспериментировать с другими соотношениями.

Дополнительно можно добавить слой отсева (dropout), чтобы предотвратить переобучение модели. Как правило, коэффициент отсева устанавливается в пределах от 0 до 0,1 и служит дополнительным фактором регуляризации, который слегка снижает скорость обучения.

Метод LoRA можно использовать с различными компонентами архитектуры модели.

Изначально основной областью применения LoRA был механизм внимания, в частности матрицы *запросов (Q)* и *значений (V)* в слоях трансформера. Тем не менее эксперименты показали, что применение LoRA к другим ключевым компонентам модели дает значительные преимущества. Этими дополнительными целевыми модулями являются:

- матрицы *ключей (K)* в слоях внимания;
- выходные проекционные слои в механизмах внимания (часто обозначаемые буквой O);
- блоки прямого распространения или *многослойные персептроны* (Multi-Layer Perceptron, MLP), расположенные между слоями внимания;
- линейные выходные слои.

Однако следует учитывать, что с ростом числа адаптируемых модулей увеличивается количество обучаемых параметров, а значит, и потребление памяти.

С помощью LoRA можно тонко настроить модель с 7 млрд параметров на одном графическом процессоре с объемом видеопамяти от 14 до 18 Гбайт в зависимости от конфигурации. Этот процесс является существенно менее ресурсоемким по сравнению с полной тонкой настройкой, которая обычно требует использования нескольких мощных видеокарт. С точки зрения количества обучаемых параметров LoRA также значительно выигрывает. Например, даже при применении данного подхода ко всем модулям с рангом 16 модель Llama-3-8B содержит всего 42 млн обучаемых LoRA-параметров, что составляет лишь 0,5196 % от общего числа параметров модели, равного 8 млрд.

Что касается качества, то LoRA позволяет достигать сопоставимых, а иногда и лучших результатов по сравнению с полной тонкой настройкой. Весовые матрицы LoRA можно комбинировать для разных задач и предметных областей, что позволяет гибко переключаться между сценариями без повторного обучения

модели. Существуют проекты, специализирующиеся на одновременном обслуживании нескольких LoRA-адаптаций, в частности LoRAX. Эта возможность также поддерживается системами *Text Generation Inference (TGI)* от Hugging Face и *Nvidia Inference Microservices (NIM)*.

QLoRA

Метод QLoRA (Quantization-aware Low-Rank Adaptation — «низкоранговая адаптация с учетом квантования»), предложенный Деттмерсом и соавторами, представляет собой подход к тонкой настройке больших языковых моделей, ориентированный на снижение вычислительных затрат. Объединяя методы квантования и LoRA, он позволяет тонко настраивать модели даже на относительно недорогих и широкодоступных графических процессорах.

В основе подхода QLoRA лежит квантование параметров базовой модели до специального 4-битного типа данных *NormalFloat (NF4)*, что значительно снижает потребление памяти. Как и в случае с LoRA, вместо обновления всех параметров модели в процессе тонкой настройки QLoRA предполагает добавление в определенные слои небольших обучаемых низкоранговых матриц (адаптеров). Во время обучения обновляются только эти адаптеры, тогда как исходные веса модели остаются неизменными. Для дополнительного сокращения объема потребляемой памяти QLoRA использует двойное квантование, при котором этой процедуре подвергаются не только параметры модели, но и сами коэффициенты квантования. Кроме того, данный метод задействует страничные оптимизаторы (paged optimizers), сглаживающие всплески потребления памяти во время обучения за счет технологии унифицированной памяти Nvidia[1].

QLoRA обеспечивает заметную экономию памяти по сравнению с LoRA. Так, для модели с 7 млрд параметров пиковое потребление при инициализации сокращается с 14 до 9,1 Гбайт, то есть на 35 %, а во время тонкой настройки — с 15,6 до 9,3 Гбайт, то есть на 40 %. Однако такая эффективность достигается за счет увеличения продолжительности обучения: при использовании метода QLoRA оно происходит примерно на 30 % медленнее, чем в случае с LoRA. Что касается уровня производительности модели, то различия между этими двумя методами минимальны.

В целом метод QLoRA может оказаться особенно актуальным в ситуациях, когда основным ограничением является объем доступной памяти, например, при работе с очень крупными моделями или на оборудовании с ограниченными ресурсами GPU. Если для вас наиболее важна скорость обучения, а памяти достаточно, то предпочтение можно отдать методу LoRA. При выборе между QLoRA и LoRA

[1] Технология Nvidia, которая позволяет GPU автоматически использовать память CPU, если своей видеопамяти (GPU) недостаточно. — *Примеч. науч. ред.*

следует опираться на конкретные требования проекта, доступную аппаратную инфраструктуру и необходимый баланс между экономией памяти, скоростью обучения и производительностью модели.

Гиперпараметры обучения

При тонкой настройке больших языковых моделей (LLM) на их сходимость, обобщающую способность и итоговую эффективность влияет целый ряд гиперпараметров.

Скорость обучения и планировщик

Одним из важнейших гиперпараметров является скорость (или темп) обучения (learning rate), определяющая степень изменения параметров модели (как сильно меняются параметры при ошибках) в ходе этого процесса. Значения скорости обучения обычно лежат в диапазоне от очень малых (например, `1e-6`) до относительно крупных (например, `1e-3`). Для трансформеров типичным стартовым значением считается `1e-5`. Если скорость обучения слишком мала, данный процесс будет протекать медленно и модель может демонстрировать неоптимальные решения. Напротив, слишком высокая скорость обучения может привести к нестабильности и расходимости, которые также негативно сказываются на производительности. В связи с этим рекомендуется поэкспериментировать с различными значениями, чтобы подобрать оптимальную скорость обучения для конкретной модели и задачи.

Планировщик (learning rate scheduler) постоянно корректирует скорость обучения. Обычно в самом начале устанавливается более высокая скорость, обеспечивающая первичный прогресс, а затем она постепенно снижается для более точной настройки модели. Наиболее популярными являются два типа планировщиков: линейный и косинусный. Линейный планировщик снижает скорость обучения равномерно на протяжении всего процесса, а косинусный использует для этого функцию косинуса, то есть сначала скорость обучения снижается медленно, а ближе к завершению обучения — все быстрее. Например, можно начать со скорости `3e-4` и в процессе обучения плавно снизить ее до `1e-7`. Конкретные значения зависят от модели и набора данных, однако распространенным подходом является использование этапа прогрева (warmup), охватывающего первые примерно 5 % шагов, в ходе которого скорость обучения возрастает от нуля до начального значения, а в ходе оставшихся 95 % шагов происходит ее постепенное затухание (decay). Такой прием позволяет стабилизировать начальную фазу обучения и проводить более точные обновления по мере приближения модели к оптимальной. В целом линейный и косинусный планировщики демонстрируют сопоставимую эффективность.

Размер пакета

Размер пакета (batch size) определяет количество образцов, обрабатываемых до обновления весов модели. Обычно при тонкой настройке LLM размер пакета варьируется от 1 до 32 примеров, а наиболее типичными значениями являются 1, 2, 4, 8 или 16. Большие размеры пакета обеспечивают более стабильные оценки градиентов и могут ускорить обучение, поскольку лучше приближают истинный градиент всего набора данных. Однако увеличение размера пакета сопровождается ростом требований к объему памяти, что может стать ограничивающим фактором при использовании GPU с ограниченным обьемом памяти. Например, пакет размером 16 примеров может эффективно обрабатываться высокопроизводительным GPU с 24 Гбайт памяти, тогда как GPU с 8 Гбайт памяти способен обработать лишь пакет размером 2 или 4 примера.

Чтобы преодолеть ограничения, связанные с памятью, и при этом не потерять преимущества больших размеров пакета, применяется такая техника, как накопление градиентов (gradient accumulation). Ее суть заключается в выполнении нескольких прямых и обратных проходов с мини-пакетами и накоплении градиентов перед применением одного обновления параметров модели. Этот подход особенно полезен при работе с большими моделями или при ограниченном объеме памяти графического процессора. Например, если вы хотите достичь эффективного размера пакета, равного 32 примерам, а ваш GPU способен обрабатывать только 8 примеров за раз, вы можете задать количество шагов накопления градиентов равным 4. В таком случае вы обработаете 4 мини-пакета по 8 примеров каждый, накопите градиенты и затем обновите модель, как если бы обработали все 32 примера одновременно.

Количество шагов накопления градиентов обычно варьируется от 1 (без накопления) до 8 или 16 примеров, в зависимости от желаемого эффективного размера пакета и доступных вычислительных ресурсов. Выбор количества шагов предполагает компромисс между скоростью обучения и доступной памятью. Большее количество шагов накопления позволяет использовать большие эффективные размеры пакета, но увеличивает время, необходимое для выполнения каждого обновления. Вот простая формула для определения эффективного размера пакета:

эффективный размер пакета = размер мини-пакета × число GPU × число шагов накопления градиентов.

Например, если вы используете 2 GPU, каждый из которых обрабатывает пакет из 4 образцов, и выполняете 4 шага накопления градиентов, то эффективный размер пакета составит $4 \times 2 \times 4 = 32$ образца.

Максимальная длина последовательности и упаковка

Максимальная длина последовательности (maximum sequence length) определяет, сколько токенов обрабатывает модель за один раз. Обычно она устанавливается в диапазоне от 512 до 4096 токенов, но может достигать 128 000 и более, в зависимости от задачи и доступной видеопамяти. Например, длина последовательности, равная 2048 токенам, часто используется при решении задач генерации текста, тогда как RAG-приложения могут требовать от 8192 токенов и больше. Если входная последовательность превышает установленный предел, лишние токены обрезаются либо с начала (left truncation), либо с конца (right truncation). Например, если максимальная длина составляет 1024 токена, а входная последовательность содержит 1500 токенов, то 476 токенов будет удалено. Этот параметр напрямую влияет на размер пакета и использование памяти: пакет из 12 образцов при максимальной длине 1024 токена содержит 12 288 токенов (12×1024), а при длине 512 токенов — только 6144 токена. Для оптимизации производительности и потребления ресурсов важно сбалансировать этот параметр с возможностями вашего GPU и особенностями обучающих данных.

Упаковка (packing) позволяет максимально эффективно использовать каждый обучающий пакет. Вместо того чтобы назначать один образец каждому из пакетов при упаковке, несколько более коротких образцов объединяются в один пакет, что увеличивает объем данных, обрабатываемых за итерацию. Например, если максимальная длина последовательности составляет 1024 токена, а многие образцы содержат лишь 200–300 токенов, упаковка позволяет уместить 3–4 образца в один пакет. Такой подход существенно повышает эффективность обучения, особенно при работе с наборами данных, содержащими множество коротких последовательностей. Однако осуществлять упаковку следует с осторожностью, чтобы механизм внимания модели не пересекал границы упакованных образцов (например, если в батче три примера, то модель может подумать, что они являются продолжением друг друга, хотя это не так). Обычно это достигается с помощью масок внимания (attention masks), которые запрещают модели обращаться к токенам из разных образцов внутри одной упакованной последовательности.

Количество эпох

Количество эпох — это еще один важный параметр, обозначающий число полных циклов, за которые модель проходит по всему обучающему набору данных. Для тонкой настройки LLM обычно используют от 1 до 10 эпох, причем успешные результаты часто достигаются при 2–5 эпохах. Оптимальное значение этого параметра зависит от таких факторов, как сложность задачи, размер набора данных и архитектура модели. Большее количество эпох позволяет модели точнее подстроиться под данные, что может улучшить ее производительность. Однако

здесь важно найти баланс, поскольку слишком малое число эпох, скорее всего, приведет к недостаточному обучению модели, а слишком большое — к переобучению. Например, крупной LLM, тонко настраиваемой на незначительном наборе данных, достаточно всего 1–3 эпох, в то время как компактной, обучающейся на большем объеме данных, для достижения лучших результатов может понадобиться 5–10 эпох. При этом рекомендуется отслеживать производительность модели на валидационном наборе данных и применять технику ранней остановки (early stopping), если производительность LLM стабилизируется или начинает ухудшаться. Такой подход позволяет динамически определять оптимальное количество эпох и предотвращает переобучение.

Оптимизаторы

Оптимизаторы помогают модели подстраивать свои веса, уменьшая ошибку и улучшая результаты. При тонкой настройке LLM рекомендуется использовать оптимизатор AdamW (Adaptive Moment Estimation with Weight Decay — «адаптивная оценка моментов[1] с затуханием весов»), особенно его 8-битную версию, которая демонстрирует производительность, сопоставимую с 32-битной версией при меньшем потреблении видеопамяти (правда, это не ускоряет сам процесс обучения). AdamW сочетает адаптивную скорость обучения с затуханием весов, что часто повышает стабильность обучения и производительность модели.

При серьезных ограничениях памяти альтернативой может стать оптимизатор AdaFactor, специально разработанный для эффективного использования памяти. Он хорошо работает без явной настройки скорости обучения, что особенно удобно в условиях ограниченных вычислительных ресурсов. Однако в некоторых случаях AdaFactor может уступать в производительности оптимизатору AdamW. При работе с чрезвычайно крупными моделями, а также при острой нехватке памяти GPU можно задействовать страничные версии оптимизаторов, такие как 8-битный страничный оптимизатор AdamW, которые уменьшают потребление памяти за счет частичного переноса данных в оперативную память CPU. Если ресурсы позволяют, а приоритетом является максимальная производительность, то предпочтение следует отдать неквантованной версии оптимизатора — `adamw_torch`.

Затухание весов

Затухание весов (weight decay) — это метод регуляризации в машинном обучении и нейросетях, который заставляет модель удерживать значения весов на минимально возможном уровне. Предполагает добавление штрафа за большие значения весов в функцию потерь, побуждающее модель изучать более простые

[1] Каждый параметр модели меняется с собственной скоростью в зависимости от того, как он вел себя раньше. — *Примеч. науч. ред.*

и обобщаемые закономерности. Это помогает избежать чрезмерной зависимости от отдельных входных признаков, что улучшает производительность модели на новых, ранее не встречавшихся данных. Обычно значения коэффициента затухания варьируются в диапазоне от 0,01 до 0,1, причем значение 0,01 часто используется в качестве отправной точки. С этого значения можно начать, например, при работе с оптимизатором AdamW.

Хотя затухание весов часто оказывается полезным, если вы выставите слишком высокое значение этого параметра, модели будет сложно улавливать важные закономерности в данных. С другой стороны, слишком низкое значение может привести к переобучению. Оптимальное значение коэффициента зависит от конкретной архитектуры модели и набора данных, поэтому на практике оно часто подбирается экспериментально.

Контрольные точки градиентов

Контрольные точки градиентов (gradient checkpointing) — это техника, позволяющая сократить потребление памяти во время обучения за счет сохранения лишь части промежуточных активаций (результатов вычислений одного слоя LLM), сгенерированных на этапе прямого прохода[1]. При стандартной процедуре обучения все промежуточные активации сохраняются в памяти для облегчения вычисления градиентов при обратном проходе[2]. Однако при обучении очень глубоких нейросетей, таких как LLM, такой подход быстро становится непрактичным из-за аппаратных ограничений, особенно при использовании маломощных GPU.

Контрольные точки градиентов решают данную проблему за счет выборочного сохранения активаций в отдельных слоях сети. Для слоев, в которых активации не сохраняются, они пересчитываются во время обратного прохода по мере необходимости для вычисления градиентов. Этот метод представляет собой компромисс между временем вычисления и использованием памяти: он позволяет существенно снизить требования к объему памяти, но может увеличить общее время обучения из-за необходимости пересчитывать некоторые активации.

Существуют и другие параметры и техники, но по сравнению с теми, что обсуждались ранее, они играют менее значительную роль. В следующем разделе мы рассмотрим процесс выбора и настройки этих параметров на конкретном примере.

---

1 Модель пропускает вход через все слои, получает предсказание и, в отличие от инференса, сохраняет промежуточные результаты для обратного прохода и вычисления ошибки. — *Примеч. науч. ред.*

2 В этом случае вычисляется разница между предсказанием во время прямого прохода и ответом из обучающих данных. — *Примеч. науч. ред.*

Тонкая настройка на практике

Теперь давайте тонко настроим модель с открытыми весами на пользовательском наборе данных. В этом разделе мы рассмотрим пример, в котором для повышения вычислительной эффективности реализованы подходы LoRA и QLoRA. В зависимости от доступного аппаратного обеспечения вы можете выбрать ту технику, которая лучше всего соответствует конфигурации вашего оборудования.

Существует множество эффективных моделей с открытыми весами, которые можно адаптировать под конкретную задачу или предметную область. При выборе наиболее подходящей LLM следует учитывать три ключевых параметра.

- *Лицензия*. Некоторые лицензии допускают использование моделей только в некоммерческих целях, что становится проблемой при тонкой настройке модели для нужд компании. Однако в этой сфере встречаются и специальные лицензии, например разрешающие работать с моделями только организациям с определенной численностью.
- *Бюджет*. Модели с меньшим числом параметров (менее 10 млрд) обходятся значительно дешевле как на этапе тонкой настройки, так и на этапе инференса, поскольку могут запускаться на более доступных GPU и обрабатывать больше токенов в секунду.
- *Производительность*. Важно оценить базовую модель с помощью универсальных бенчмарков или бенчмарков, отражающих специфику задачи или предметной области. Это позволяет гарантировать, что после тонкой настройки модель будет обладать необходимыми возможностями для выполнения целевых задач.

В текущей главе мы воспользуемся LLM с открытыми весами Llama-3.1-8B, чья лицензия (Llama 3.1 Community License Agreement) допускает коммерческое применение. Эта компактная модель, имеющая 8 млрд параметров, запускается на большинстве GPU и демонстрирует высокие показатели производительности в сравнении с конкурентами, что подтверждается данными Open LLM Leaderboard и другими бенчмарками, указанными в карточке модели.

Для тонкой настройки языковых моделей существуют специализированные инструменты и библиотеки. Мы рекомендуем следующие.

- *TRL*. Эта библиотека, разработанная и поддерживаемая Hugging Face, предназначена для обучения LLM с применением таких методов, как SFT и выравнивание по предпочтениям (preference alignment). Она довольно популярна и считается одной из самых актуальных с точки зрения реализуемых алгоритмов. TRL поддерживает обучение модели как на одном, так и на

нескольких GPU с FSDP (Fully Sharded Data Parallel — шардирование параметров модели между GPU) и DeepSpeed (набор инструментов для ускоренного и экономичного обучения больших моделей).

- *Axolotl*. Этот инструмент, разработанный Вингом Лианом, упрощает процесс тонкой настройки LLM с помощью повторно используемых конфигурационных файлов в формате YAML. Он построен на базе TRL, но включает множество дополнительных функций, таких как автоматическое объединение наборов данных в различных форматах. Как и TRL, он поддерживает обучение модели как на одном, так и на нескольких GPU с FSDP и DeepSpeed.
- *Unsloth*. Разработанный Дэниелом и Майклом Ханом, этот инструмент использует пользовательские ядра (custom kernels) для ускорения обучения (в 2–5 раз) и снижения потребления памяти (до 80 %). Он также основан на TRL и предлагает множество дополнительных функций, включая автоматическое преобразование моделей в формат квантования GGUF (GPT-Generated Unified Format — компактный формат файлов для хранения и запуска больших языковых моделей на обычном «железе»). На момент написания книги он поддерживал обучение модели только на одном GPU.

Чтобы максимизировать эффективность, мы выполним тонкую настройку модели с помощью библиотеки Unsloth. Приведенный далее код является частью нашего LLMOps-пайплайна, но может использоваться и в качестве самостоятельного скрипта. Он без проблем запускается в различных средах, таких как Amazon SageMaker, облачные GPU (например, Lambda Labs или RunPod), Google Colab и др. Мы протестировали этот код на различных графических процессорах, включая A40, A100 и L4.

Библиотеку Unsloth и все ее зависимости рекомендуется устанавливать непосредственно из GitHub-репозитория книги (https://github.com/PacktPublishing/LLM-Engineering) или из официального репозитория Unsloth (https://github.com/unslothai/unsloth), так как инструкции по установке регулярно обновляются с учетом возможных конфликтов зависимостей.

1. Для начала необходимо получить доступ к закрытой модели и (при необходимости) подготовить возможность загрузки дообученной модели на платформу Hugging Face (https://huggingface.co/). Для этого потребуется авторизация в аккаунте. Если у вас еще нет аккаунта, создайте его и сохраните API-ключ (Settings ▸ Access Tokens ▸ Create new token (Настройки ▸ Токены доступа ▸ Создать новый токен)) в файле `.env`:

   ```
   HF_TOKEN = YOUR_API_KEY
   ```

2. Убедитесь, что в файле `.env` также содержится ваш API-ключ Comet ML:

   ```
   COMET_API_KEY = YOUR_API_KEY
   ```

3. Импортируйте все необходимые пакеты:

```
import os
import torch
from trl import SFTTrainer
from datasets import load_dataset, concatenate_datasets
from transformers import TrainingArguments, TextStreamerfrom unsloth
import FastLanguageModel, is_bfloat16_supported
```

4. Теперь загрузим модель для тонкой настройки и соответствующий ей токенизатор. Мы воспользуемся классом `FastLanguageModel` из библиотеки Unsloth с методом `.from_pretrained()`. Помимо имени модели, необходимо указать максимальную длину последовательности (в данном случае — 2048 токенов). Аргумент `load_in_4bit` указывает, будет ли применяться метод QLoRA или LoRA.

   В этом примере мы прибегнем к методу LoRA, поскольку он обеспечивает более высокое качество и скорость обучения. Однако при дефиците видеопамяти вы можете легко переключиться на QLoRA.

```
max_seq_length = 2048
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="meta-llama/Meta-Llama-3.1-8B",
    max_seq_length=max_seq_length,
    load_in_4bit=False,
)
```

5. Теперь, когда модель загружена, давайте определим конфигурацию LoRA. Мы задаем значение ранга, равное 32, — в нашем случае этого достаточно, чтобы модель могла имитировать стиль написания текстов и перенимать знания из обучающих примеров. Если полученные результаты окажутся неудовлетворительными, вы сможете увеличить данный параметр до 64 или 128. Мы задаем значение параметра `alpha` равным 32 и, чтобы ускорить обучение, отключаем параметр `dropout` и не используем смещение (`bias`). Наконец, мы применяем тонкую настройку ко всем линейным слоям, чтобы максимизировать качество этого процесса.

```
model = FastLanguageModel.get_peft_model(
    model,
    r=32,
    lora_alpha=32,
    lora_dropout=0,
    target_modules=["q_proj", "k_proj", "v_proj", "up_proj",
    ➥"down_proj", "o_proj", "gate_proj"],
)
```

6. Затем нам нужно подготовить данные в формате, пригодном для тонкой настройки. В нашем случае в наборе `llmtwin` содержится всего около 3000 образцов, чего может оказаться недостаточно для того, чтобы модель корректно усвоила структуру шаблона ведения диалога. Чтобы решить эту проблему, мы дополним его высококачественным универсальным набором данных `FineTome` — отфильтрованной версией набора `arcee-ai/The-Tome`, полученной

с помощью классификатора `fineweb-edu-classifier`. Вместо того чтобы использовать все 100 000 образцов из набора, мы явно укажем, что нам нужны только первые 10 000. Затем мы объединяем эти два набора в один.

```
dataset1 = load_dataset("mlabonne/llmtwin")
dataset2 = load_dataset("mlabonne/FineTome-Alpaca-100k",
➥split="train[:10000]")
dataset = concatenate_datasets([dataset1, dataset2])
```

7. После объединения данных их необходимо отформатировать с помощью шаблона ведения диалога. Для удобства выберем шаблон Alpaca. Он не требует добавления специальных токенов, что снижает вероятность возникновения ошибок (хотя может чуть сильнее повлиять на производительность по сравнению с ChatML). Здесь мы сопоставляем все инструкции и ответы с шаблоном Alpaca и вручную добавляем *токен конца последовательности* (`EOS_TOKEN`) к каждому сообщению, чтобы модель научилась завершать свои ответы и не генерировала их бесконечно.

```
alpaca_template = """Ниже приведена инструкция, описывающая задачу.
Напиши ответ, который правильно дополняет запрос.

                  ### Инструкция:
                  {}

                  ### Ответ:
                  {}"""

EOS_TOKEN = tokenizer.eos_token
dataset = dataset.map(format_samples, batched=True, remove_
➥columns=dataset.column_names)
```

8. Готовый набор данных необходимо разделить на обучающую и тестовую выборки: 95 % данных мы выделим для обучения модели, а оставшиеся 5 % — для ее проверки, чтобы во время обучения понимать, улучшается модель или нет.

```
dataset = dataset.train_test_split(test_size=0.05)
```

9. Теперь модель готова к обучению. Класс `SFTTrainer()` хранит все гиперпараметры, используемые в ходе этого процесса. Кроме того, ему передаются модель, токенизатор, конфигурация LoRA и подготовленные наборы данных. В соответствии с рекомендациями из предыдущего раздела мы устанавливаем скорость обучения (`learning rate`) равной `3e-4`, задействуем линейный планировщик (`linear scheduler`) и задаем максимальную длину последовательности, равную 2048 токенам. Модель обучается в течение трех эпох на пакетах размером 2 с накоплением градиентов за 8 шагов, что эквивалентно эффективному размеру пакета, равному 16. В качестве оптимизатора мы выбрали `adamw_8bit` с коэффициентом затухания `weight decay = 0.01`. Наконец, мы отправляем информацию о запуске процесса обучения в систему Comet ML для отслеживания эксперимента.

```
trainer = SFTTrainer(
    model=model,
    tokenizer=tokenizer,
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    dataset_text_field="text",
    max_seq_length=max_seq_length,
    dataset_num_proc=2,
    packing=True,
    args=TrainingArguments(
        learning_rate=3e-4,
        lr_scheduler_type="linear",
        per_device_train_batch_size=2,
        gradient_accumulation_steps=8,
        num_train_epochs=3,
        fp16=not is_bfloat16_supported(),
        bf16=is_bfloat16_supported(),
        logging_steps=1,
        optim="adamw_8bit",
        weight_decay=0.01,
        warmup_steps=10,
        output_dir="output",
        report_to="comet_ml",
        seed=0,
    ),
)

trainer.train()
```

Обучение такой модели на объединенном наборе данных занимает до нескольких часов. Например, на графическом процессоре A100 на это ушло около 50 минут.

10. После обучения можно провести быструю проверку. Ее целью является не полноценная оценка качества тонко настроенной модели, а базовая проверка на отсутствие ошибок, связанных с токенизатором или шаблоном ведения диалога.

    Для быстрого инференса мы применяем метод `FastLanguageModel.for_inference()` из библиотеки Unsloth и форматируем инструкцию с использованием шаблона Alpaca. Обратите внимание на пустой ответ, добавляемый в конец пользовательской инструкции, — он служит для вставки заголовка помощника (`### Response:`). Это побуждает модель дать ответ на инструкцию вместо того, чтобы продолжить ее. Мы также задействуем потоковый инференс, чтобы отображать текст по мере его генерации, не дожидаясь завершения этого процесса.

```
FastLanguageModel.for_inference(model)
message = alpaca_prompt.format("Напиши абзац, в котором объясняется,
➥что такое тонкая настройка с учителем.", "")
inputs = tokenizer([message], return_tensors="pt").to("cuda")
text_streamer = TextStreamer(tokenizer) _ = model.generate(**inputs,
➥streamer=text_streamer, max_new_tokens=256, use_cache=True)
```

11. Вот пример ответа, сгенерированного нашей моделью:

```
Тонкая настройка с учителем — это метод, который используется для улучшения
языковой модели путем предоставления ей тщательно отобранного набора данных
с инструкциями и соответствующими ответами. Этот процесс направлен на то,
чтобы ответы модели соответствовали ожиданиям человека, тем самым повышая ее
точность и релевантность. Цель состоит в том, чтобы модель могла эффективно
отвечать на широкий спектр запросов, что делает ее ценным инструментом для
таких приложений, как чат-боты и виртуальные помощники.
```

    Этот ответ корректен и отформатирован в соответствии с шаблоном ведения диалога Alpaca.

12. Теперь тонко настроенную модель можно сохранить локально и/или загрузить на платформу Hugging Face Hub с помощью следующих функций.

```
model.save_pretrained_merged("model", tokenizer,
➥save_method="merged_16bit")
model.push_to_hub_merged("mlabonne/TwinLlama-3.1-8B", tokenizer,
➥save_method="merged_16bit")
```

Поздравляем, вы тонко настроили базовую модель! В процессе обучения в системе Comet ML можно отслеживать такие метрики, как значение функции потерь на обучающей и валидационной выборках, а также множество других показателей. Важно следить за тем, чтобы они демонстрировали ожидаемое поведение. На рис. 5.11 показаны метрики в системе Comet ML, полученные в ходе запуска приведенного выше кода.

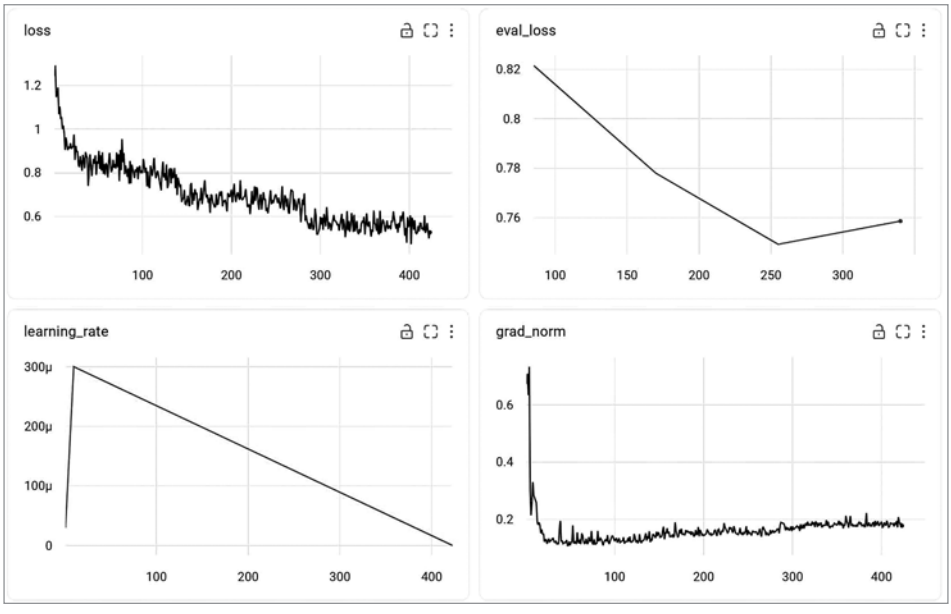


Рис. 5.11. Четыре метрики, отслеживаемые в Comet ML при тонкой настройке модели

Наиболее важными являются следующие показатели.

- *Значение функции потерь на обучающей выборке (training loss)*. Эта метрика показывает, насколько успешно модель справляется с задачей на обучающих данных. В идеале среднее значение должно постепенно уменьшаться. Как правило, на начальных этапах обучения наблюдается резкое снижение потерь, за которым следует длительное плато. Резкие скачки или устойчивый рост значения этой метрики могут указывать на проблемы с обучением. В таких случаях стоит проверить качество данных, корректность работы токенизатора, а также скорректировать значения скорости обучения и размера пакета. На рис. 5.11 (график loss) отчетливо видны три фазы, соответствующие трем эпохам обучения.
- *Значение функции потерь на валидационной выборке (validation loss)*. Эта метрика отражает потери не на обучающем, а на валидационном наборе данных. Для хорошо обученной модели характерно постепенное снижение и стабилизация обеих метрик и небольшой разрыв между ними. Он не должен быть нулевым, так как модель почти всегда показывает лучшие результаты на обучающих данных. Если потери на обучающих данных продолжают снижаться, а на валидационной выборке начинают расти, то это является признаком переобучения. Если же оба показателя остаются стабильно высокими, то это свидетельствует о недостаточном обучении. Универсальных «правильных» значений потерь не существует, так как они зависят от конкретной задачи и используемой функции потерь. Однако важно, чтобы обе кривые демонстрировали сходимость и стабилизацию. На рис. 5.11 (график eval_loss) заметен небольшой рост на шаге 340 — он все еще является приемлемым, но может указывать на начавшееся переобучение модели.
- *Норма градиента (gradient norm)*. Норма градиента отражает величину вектора градиента на каждом шаге обучения. Слишком большие значения могут свидетельствовать, например, о переобучении, особенно если они сопровождаются расхождением между значениями функции потерь на обучающей и валидационной выборках. Напротив, стабильное или убывающее значение нормы градиента обычно указывает на схождение модели к локальному оптимуму. Для снижения рисков, связанных с чрезмерно большими значениями этого показателя, можно использовать такую технику, как обрезка градиентов (gradient clipping). Она предполагает задание верхнего порога для нормы градиента, который ограничивает величину обновлений параметров.

Часто бывает полезно поэкспериментировать с различными скоростями обучения и выбрать модель, демонстрирующую наименьшие потери. Однако следует помнить, что это лишь косвенные показатели эффективности модели. Процесс полноценной оценки ее качества мы рассмотрим в одной из следующих глав.

Резюме

В этой главе были рассмотрены ключевые теоретические и практические аспекты тонкой настройки больших языковых моделей. Мы обсудили пайплайн подготовки высококачественных данных с инструкциями и процесс создания их наборов — от подготовки до аугментации. На каждом этапе этого пайплайна существуют возможности для оптимизации, особенно в части оценки качества, генерации и улучшения данных. Вы можете легко адаптировать пайплайн под свой сценарий использования, выбрав наиболее релевантные этапы и методы.

Мы применили описанный подход к реальным данным из главы 3, преобразовав исходный текст в пары «инструкция — ответ» с помощью LLM. Затем мы подробно рассмотрели такие техники тонкой настройки с учителем (SFT), как полная тонкая настройка, LoRA и QLoRA, включая их преимущества и ограничения, а также способы хранения и разбора наборов инструкций с использованием шаблонов ведения диалога. Мы сравнили эти методы с точки зрения потребления памяти, эффективности обучения и качества инференса. В конце главы был описан процесс тонкой настройки модели Llama-3.1-8B на пользовательском наборе данных с инструкциями. Этот пример позволил подчеркнуть ключевые этапы и важные детали реализации, обеспечивающие успешную тонкую настройку.

В следующей главе применим техники выравнивания по предпочтениям для создания новой версии модели TwinLlama-3.1-8B. Мы сгенерируем новый набор данных, содержащий как выбранные, так и отклоненные ответы, что позволит откалибровать тип ожидаемых от модели ответов. Мы также подробно рассмотрим множество практических приложений этого подхода и обсудим способы его реализации.

Список используемых источников

- *Tahori, Gulrajani, Zhang, Dubois et al.* Alpaca: A Strong, Replicable Instruction-Following Model. crfm.stanford.edu. https://crfm.stanford.edu/2023/03/13/alpaca.html
- *Mukherjee S. et al.* Orca: Progressive Learning from Complex Explanation Traces of GPT-4. arXiv preprint arXiv:2306.02707.
- *Lian W., Goodson B., Pentland E., Cook A., Vong C., Teknium.* Open-Orca/OpenOrca. huggingface.co. https://huggingface.co/datasets/Open-Orca/OpenOrca.
- *Zeng W. et al.* Automatic Instruction Evolving for Large Language Models. arXiv preprint arXiv:2406.00770.
- *Zhou C. et al.* LIMA: Less Is More for Alignment. arXiv preprint arXiv:2305.11206.

- 01.AI. Yi: Open Foundation Models by 01.AI. arXiv preprint arXiv:2403.04652.
- *Birch A.* LLM finetuning memory requirements. blog.scottlogic.com. https://blog.scottlogic.com/2023/11/24/llm-mem.html.
- *Anthony Q. et al.* Transformer Math 101. blog.eleuther.ai. https://blog.eleuther.ai/transformer-math/.
- *Hu E. J. et al.* LoRA: Low-Rank Adaptation of Large Language Models. arXiv preprint arXiv:2106.09685.
- *Dettmers T. et al.* QLoRA: Efficient Finetuning of Quantized LLMs. arXiv preprint arXiv:2305.14314.

6 Тонкая настройка с учетом предпочтений

Тонкая настройка с учителем (SFT) сыграла важную роль в адаптации LLM к выполнению конкретных задач. Однако данный подход плохо учитывает нюансы человеческих предпочтений и не охватывает широкий спектр взаимодействий, с которыми может столкнуться модель. Эти ограничения привели к разработке более совершенных методов согласования ИИ-систем с предпочтениями людей.

Методы *согласования предпочтений* (preference alignment) преодолевают ограничения SFT за счет включения в процесс обучения обратной связи непосредственно от человека или другой модели. Такой прием позволяет лучше улавливать тонкие различия в предпочтениях, особенно в сложных ситуациях, где обычного обучения с учителем оказывается недостаточно. Хотя для согласования предпочтений существует множество методов, в этой главе основное внимание будет уделено *прямой оптимизации предпочтений* (Direct Preference Optimization, DPO), как наиболее простому и эффективному подходу.

Мы рассмотрим типы данных, необходимых для использования таких алгоритмов согласования предпочтений, как DPO, и создадим собственный набор данных, с помощью которого скорректируем стиль письма модели, сделав его более естественным. Для этого мы реализуем алгоритм DPO и применим его к модели, тонко настроенной в главе 5.

К концу главы вы научитесь создавать наборы данных о предпочтениях и задействовать различные техники для согласования поведения моделей с этими предпочтениями.



Все примеры кода из этой главы доступны в репозитории GitHub: https://github.com/PacktPublishing/LLM-Engineering.

Особенности наборов данных о предпочтениях

Принципы создания качественных наборов данных о предпочтениях аналогичны тем, что были рассмотрены в главе 5 в контексте обсуждения наборов инструкций. В данном случае мы также стремимся к максимизации точности, разнообразия и сложности примеров. Для этого мы снова выполняем такие действия, как подготовка данных, удаление дубликатов, очистка данных от пересечений, оценка качества, исследование, генерация и аугментация (рис. 6.1).

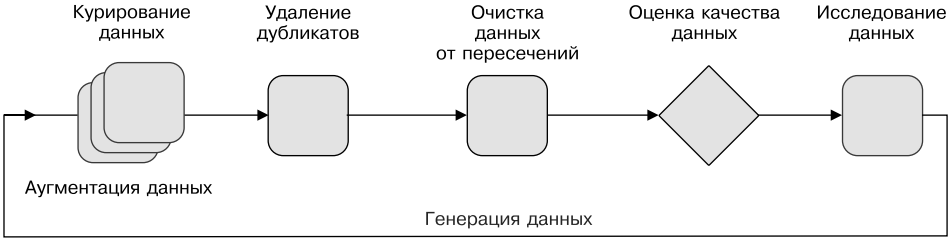


Рис. 6.1. Обзор пайплайна подготовки данных, рассматриваемого в этой главе

Чтобы избежать повторов, мы сосредоточим внимание на основных различиях между наборами данных с инструкциями и наборами данных о предпочтениях. Мы рассмотрим структуру примеров, отражающих предпочтения, а также оптимальный размер таких наборов. Затем подробно обсудим два этапа, которые больше всего отличаются от соответствующих этапов создания обучающих наборов инструкций: генерацию данных и их оценку.

Данные о предпочтениях

В отличие от наборов данных с инструкциями наборы данных о предпочтениях не имеют единой стандартной структуры, поскольку алгоритмы обучения предъявляют к ним разные требования. Такие наборы состоят из множества ответов на одну и ту же инструкцию, упорядоченных по степени предпочтительности, как правило, на основе оценок человека или языковой модели. Поскольку в этой главе основное внимание уделяется алгоритму DPO, мы рассмотрим именно тот формат данных, который требуется для его реализации.

Как показано в табл. 6.1, структура наборов данных для DPO довольно проста: каждая инструкция сопровождается одним предпочтительным и одним отклоненным ответом. Цель заключается в том, чтобы научить модель генерировать предпочтительный ответ вместо отклоненного.

Таблица 6.1. Пример из набора данных о предпочтениях mlabonne/orpo-dpo-mix-40k

Инструкция	
Расскажи шутку про осьминогов	
Предпочтительный ответ	**Отклоненный ответ**
Почему осьминоги не играют в карты в казино?	Сколько щекоток нужно, чтобы рассмешить осьминога?
Потому что они не умеют считать больше восьми	Десять щекоток. (Игра слов: ten tickles («десять щекоток») созвучно слову tentacles («щупальца»))

В наборах данных о предпочтениях отклоненный ответ так же важен, как и выбранный: без него перед нами был бы просто набор с инструкциями. Отклоненные ответы представляют то поведение, от которого мы стремимся отучить модель. Это делает наборы данных о предпочтениях весьма гибким инструментом, пригодным для множества сценариев. Ниже перечислены ситуации, в которых использование таких наборов оказывается более эффективным по сравнению с одной лишь тонкой настройкой с учителем.

- *Чат-боты.* В случае диалогового ИИ качество ответа часто зависит от субъективных факторов вроде естественности, вовлеченности и уместности в контексте. Набор данных о предпочтениях позволяет модели усвоить эти нюансы путем сравнения удачных и неудачных ответов. При простом дообучении с учителем она может не уловить тонкости, определяющие предпочтительность конкретного ответа в той или иной ситуации.
- *Модерация контента.* Определение того, соответствует ли контент установленным требованиям или нарушает их, зачастую требует весьма изощренных суждений. Наборы данных о предпочтениях помогают модели научиться различать пограничные случаи путем сравнения примеров допустимого и недопустимого контента. Этот подход оказывается более эффективным по сравнению с обучением модели бинарной классификации методами SFT, поскольку позволяет модели понять логику, лежащую в основе решений, связанных с модерацией контента.
- *Резюмирование.* Качество краткого изложения часто оценивается по таким критериям, как сжатость, релевантность и связность. Наборы данных о предпочтениях помогают моделям генерировать более информативные и полезные людям резюме. Простая тонкая настройка с учителем может привести к получению технически корректных, но менее предпочтительных с точки зрения читателя результатов.
- *Генерация кода.* Для задач в области программирования часто существует несколько корректных решений, но некоторые из них являются более эффективными, удобочитаемыми или соответствующими лучшим практикам.

Наборы данных о предпочтениях позволяют модели научиться улавливать эти качественные нюансы, что довольно трудно сделать в рамках простой проверки корректности решения, предусматриваемой методами SFT.

- *Художественный текст*. При генерации прозы или поэзии результат оценивается с помощью множества субъективных критериев. Наборы данных о предпочтениях лучше отражают человеческие суждения о стиле, креативности и эмоциональном воздействии, чем наборы данных с инструкциями, которые чаще ориентированы на техническую корректность и соответствие промптам.
- *Перевод*. Хотя традиционные метрики, такие как BLEU (Bilingual Evaluation Understudy — измерение различий между автоматическим переводом и эталонными переводами того же предложения, выполненными людьми), позволяют измерить точность перевода, они не всегда отражают степень естественности языка. Благодаря наборам данных о предпочтениях модель может научиться генерировать такие переводы, которые носители языка воспринимают как более качественные по сравнению с другими технически корректными вариантами.

Во всех перечисленных сценариях наборы данных о предпочтениях помогают реализовать более тонкий подход к обучению. Они фиксируют субъективные оценки качества и человеческие предпочтения, выходящие за рамки простой корректности или следования инструкциям. Этот метод позволяет научить модель генерировать результаты, которые не только технически точны, но и лучше согласуются с суждениями людей и предпочтениями в плане решения сложных, открытых задач.

В отличие от наборов данных с инструкциями, у этих наборов не существует стандартных форматов вроде Alpaca или ShareGPT. Большинство из них имеет структуру, аналогичную той, что показана в табл. 6.1, то есть содержит три столбца: для инструкции, предпочтительного ответа и отклоненного ответа. Многоэтапные диалоги в задачах согласования предпочтений встречаются крайне редко. На момент написания книги основные библиотеки для тонкой настройки не поддерживали полноценные многоэтапные взаимодействия и, как правило, извлекали из диалога только первое или последнее сообщение.

Объем данных. Для наборов данных, используемых в рамках DPO, как правило, требуется меньше примеров, чем для наборов с инструкциями, чтобы существенно повлиять на поведение модели. Как и в случае с инструкциями, необходимое количество примеров зависит от размера модели и сложности задачи. Крупные модели эффективнее работают с обучающими примерами и, соответственно, обходятся меньшим объемом данных, тогда как более сложные задачи требуют дополнительного количества примеров для формирования нужного поведенческого шаблона. При этом качество данных остается ключевым

фактором, и наличие большого числа пар предпочтений обычно положительно влияет на результаты обучения модели.

Для общего (универсального) согласования (general-purpose alignment), выполняемого поставщиками LLM с целью повышения общей производительности тонко настроенных моделей, требуются миллионы примеров. Крупные игроки индустрии ИИ, например Nvidia, задействуют схожие пайплайны постобучения, включающие несколько этапов согласования предпочтений и предусматривающие широкое использование синтетических данных. Это говорит о том, что именно такие методы оказываются наиболее эффективными в плане расширения возможностей языковых моделей.

Сообщество пользователей и разработчиков свободного ПО работает с менее объемными наборами данных, содержащими от 10 000 до 100 000 примеров для повышения производительности моделей. Этот подход доказал свою эффективность не только в плане улучшения результатов бенчмарк-тестов, но и в плане восстановления качества нейросети после слияния[1] моделей, обрезки параметров и других модификаций. В целом DPO считается менее деструктивным методом, по сравнению с SFT, и оказывает более мягкое воздействие на финальную модель.

С другой стороны, для задач, подобных описанным ранее, требуется гораздо меньше пар предпочтений. Ориентированное на решение определенной задачи согласование направлено на улучшение поведения модели при выполнении конкретной функции, такой как изменение стиля письма, отказ от выполнения определенных инструкций и т. д. В подобных случаях часто можно обойтись небольшими наборами данных, содержащими от 100 до 10 000 пар предпочтений в зависимости от сложности решаемой проблемы.

Примером задачи, для которой достаточно ограниченного объема данных, является обучение модели тому, чтобы она утверждала, что не была обучена в OpenAI или другой крупной организации. Это можно реализовать с помощью набора данных о предпочтениях, где отклоненными являются ответы, в которых модель ошибочно указывает альтернативное происхождение, а предпочтительными — ответы, в которых она корректно утверждает, что была обучена вами. Для такой задачи может быть достаточно всего 200–500 пар предпочтений.

Генерация и оценка данных

При создании наборов данных о предпочтениях генерация и оценка данных тесно связаны между собой. Сначала мы формируем ответы, а затем оцениваем их для составления итогового набора данных. В этом разделе оба этапа рассматриваются не как отдельные шаги, а как единый процесс.

---

[1] Процесс обьединения моделей; например, когда первая модель хороша в одной области, а вторая — в другой. — *Примеч. науч. ред.*

Генерация данных о предпочтениях

Прежде чем приступать к созданию собственного набора данных о предпочтениях, полезно ознакомиться с уже существующими открытыми наборами. Хотя их количество меньше, чем наборов с инструкциями, на платформе Hugging Face Hub вы найдете качественные наборы данных, которые подходят для решения конкретных задач или могут дополнить ваш собственный. Среди известных наборов данных о предпочтениях выделяется Anthropic HH-RLHF (Helpful-Harmless Reinforcement Learning from Human Feedback — полезное и безопасное обучение с подкреплением на основе человеческой обратной связи), содержащий оценки полезности и безвредности ответов ИИ. Стоит также обратить внимание на OpenAI Summarize from Human Feedback (суммаризация на основе человеческой обратной связи), предназначенный для краткого изложения статей.

Наборы данных для DPO могут создаваться разными способами, каждый из которых представляет собой определенный компромисс между качеством, стоимостью и масштабируемостью. Эти методы можно адаптировать под конкретные прикладные задачи, и они различаются степенью участия человека в процессе. Мы делим эти методы на четыре основные категории.

- *Наборы, созданные и оцененные людьми*. Этот метод предполагает привлечение специалистов для создания ответов на промпты и последующей оценки их качества. Такой подход позволяет зафиксировать нюансы человеческих предпочтений и идеально подходит для сложных задач. Однако он является крайне затратным и слабо масштабируемым, поэтому используется преимущественно крупными ИИ-компаниями, располагающими значительными ресурсами.
- *Наборы, созданные людьми, но оцененные LLM*. Данный метод может пригодиться, если у вас уже есть большой объем контента, созданного людьми. Однако на практике он применяется редко из-за неэффективности: он по-прежнему требует значительных усилий со стороны человека для генерации ответов, но не гарантирует улавливание нюансов предпочтений на этапе оценки, выполняемой моделью.
- *Наборы, созданные LLM и оцененные людьми*. Этот подход обеспечивает хороший баланс между качеством и эффективностью. Модели генерируют несколько ответов на промпт, после чего люди их ранжируют. Такой метод часто оказывается предпочтительным, поскольку люди, как правило, лучше справляются с оценкой ответов, чем с их написанием. Он позволяет быстро генерировать разнообразные ответы и эффективно учитывать человеческие предпочтения. Однако при таком подходе реже удается получить креативные или нестандартные формулировки, свойственные человеку.
- *Наборы, созданные и оцененные LLM*. Полностью синтетические наборы, сгенерированные и оцененные моделями, становятся все более распространенными благодаря их масштабируемости и низкой стоимости. Этот метод

позволяет быстро создавать крупные наборы данных, качество которых улучшается по мере развития моделей. Однако он требует тщательного проектирования промптов для обеспечения качества и разнообразия данных и может закреплять предвзятости или ограничения, присущие LLM, которая их генерирует.

На практике наборы данных, созданные вручную, являются дорогостоящими, плохо масштабируются и не всегда обладают наивысшим качеством. В то же время оценка со стороны человека весьма ценна, но ее также сложно масштабировать — именно поэтому крупные наборы данных часто выигрывают от автоматической оценки с помощью LLM. Помимо этих общих соображений, следует учитывать способ получения данных и предполагаемый сценарий их использования. Так, в приложениях с большим числом пользователей можно встроить механизм получения обратной связи, позволяющий фиксировать предпочтения людей, например, с помощью простой оценки «нравится/не нравится» или путем сбора развернутых текстовых отзывов.

Следует отметить, что оценка не всегда является обязательной, так как предпочтения могут формироваться естественным образом в процессе генерации. Например, можно использовать качественную модель для генерации предпочтительных ответов, а менее точную или намеренно ограниченную — для генерации отклоненных ответов. Это создает четкое различие между примерами в наборе данных о предпочтениях и позволяет более эффективно обучать ИИ отличать и воспроизводить качественные ответы. Например, именно таким способом был создан набор данных `Intel/orca_dpo_pairs`, доступный на платформе Hugging Face Hub.

Другой подход заключается в сравнении ответов, сгенерированных моделью, с текстами, написанными человеком, что может дать представление о том, насколько хорошо поведение модели соответствует реальным предпочтениям людей, и выявить области, требующие улучшений. Этот метод также можно использовать для копирования определенного стиля и придания ответам модели более естественного и аутентичного звучания.

Рекомендации по генерации данных

Наборы данных о предпочтениях генерируются так же, как и наборы с инструкциями. Промпты следует формулировать таким образом, чтобы они стимулировали модель генерировать разнообразные и сложные ответы. Создавая промпты, которые явно требуют применения различных подходов или стилей изложения, мы обеспечиваем генерацию широкого спектра результатов, отражающих многообразие человеческих предпочтений.

Например, при резюмировании текста можно специально запрашивать различные варианты: краткие пересказы, подробные обобщения или резюме, отража-

ющие только ключевые моменты. Такой подход не только позволяет получить богатый набор данных, но и помогает понять, как те или иные стили и способы изложения соотносятся с предпочтениями людей.

Еще одним важным аспектом создания синтетических наборов данных о предпочтениях является привнесение вариативности в выходные данные. Этого можно добиться с помощью настройки таких параметров генерации, как температура, или применения других методов выборки. При высоких значениях параметра температуры LLM, как правило, генерирует более креативные и разнообразные ответы, а при низких — более сфокусированные и детерминированные. Это позволяет сбалансировать разнообразие и согласованность в зависимости от того, какие именно данные мы хотим получить. Например, для генерации кода требуется низкая креативность и, соответственно, низкая температура, тогда как для написания статей можно задать высокое значение этого параметра.

Использование нескольких LLM для генерации ответов может дать лучший результат по сравнению с применением только одной модели. Некоторые LLM лучше справляются с определенными задачами, поэтому такой подход способствует большему разнообразию получаемых результатов. Именно так были созданы популярные открытые наборы данных вроде `argilla/Capybara-Preferences`: в них сочетаются ответы, сгенерированные GPT-4 и моделями с открытыми весами. После генерации данных следует этап оценки, на котором выбираются предпочтительный и отклоненный ответы.

Оценка предпочтений

Оценка данных может выполняться как с участием людей, так и автоматически с помощью LLM. Второй способ предполагает разработку четких критериев, формулирование промпта, ясно передающего эти критерии модели, и использование LLM для выбора предпочтительного и отклоненного ответа. Такой подход отличается высокой масштабируемостью и обеспечивает единообразие применения заданных правил, однако качество его результатов напрямую зависит от производительности модели и четкости предоставленных инструкций. При его использовании могут быть упущены тонкости человеческих предпочтений или культурные нюансы. Тем не менее по мере развития языковых моделей их способность делать более тонкие суждения также растет, что способствует созданию более качественных наборов данных.

Оценка предпочтений с помощью LLM может осуществляться двумя способами: с использованием абсолютной шкалы оценок или методом попарного сравнения. В первом случае модель присваивает каждому ответу числовую или категориальную оценку на основе заданных критериев. Этот метод прост в реализации, но его результаты могут быть непоследовательными при использовании разных промптов и в рамках разных сеансов оценки. В случае попарного сравнения модели предъявляются два ответа, из которых она должна выбрать лучший или

ранжировать их. Такой подход ближе к формату оценки с участием человека и, как правило, дает более стабильные и воспроизводимые результаты.

При использовании абсолютной шкалы создается промпт, описывающий критерии и запрашивающий оценку по заданной шкале (например, от 1 до 5 или по категориям «плохо/удовлетворительно/хорошо/отлично»). Например: «Оцените следующий ответ по шкале от 1 до 5, учитывая релевантность, связность и полезность: `[ВСТАВЬТЕ ОТВЕТ]`». В случае попарного сравнения промпт может выглядеть так: «Сравните следующие два ответа. Какой из них лучше с точки зрения релевантности, связности и полезности? Ответ А: `[ВСТАВЬТЕ ОТВЕТ А]` Ответ Б: `[ВСТАВЬТЕ ОТВЕТ Б]`».

Сравнительный характер наборов данных о предпочтениях делает попарное сравнение оптимальным способом их оценки. Этот метод, как правило, обеспечивает более точные и согласованные с человеческими суждениями результаты, чем применение абсолютных шкал. Он имитирует свойственный человеку способ сравнения вариантов и поэтому упрощает задачу как людям, так и моделям.

Точность попарного сравнения можно повысить, предоставив модели эталонный ответ и используя цепочки рассуждений. Такой подход побуждает модель учитывать несколько аспектов каждого ответа и четко формулировать свои выводы, что способствует более обоснованной и тщательной оценке. Если эталон недоступен, можно сформулировать промпт, побуждающий модель создать оценочную заметку с описанием ожидаемого ответа. Этот метод особенно эффективен в ситуациях, когда модель недостаточно хорошо разбирается в теме, поскольку он заставляет ее сначала сформулировать критерии оценки, а затем применить их к анализу ответов.

Ниже приведен пример промпта для реализации попарного сравнения с помощью LLM-судьи.

Пример промпта для реализации попарного сравнения с помощью LLM-эксперта

Инструкция

Ты выступаешь в роли оценщика ответов. Твоя задача — сравнить ответы А и Б. Необходимо определить, какой из них лучше соответствует инструкции с точки зрения релевантности, точности, полноты, ясности, структуры и сжатости.

Инструкция: {инструкция}

Ответ А: {ответ_а}

Ответ Б: {ответ_б}

Пошагово обоснуйте свою оценку и укажите букву, соответствующую лучшему ответу, используя следующий формат.

Обоснование: (сравнение двух ответов)

Лучший ответ: (А или Б)

При этом важно помнить о том, что оценка с помощью LLM может быть подвержена нескольким видам предвзятостей.

- *Позиционная (position bias)*. При сравнительной оценке LLM-эксперты склонны отдавать предпочтение первому из двух предоставленных ответов. Это искажает результаты и может приводить к неправильным выводам.
- *Предвзятость в пользу длинных ответов (length bias)*. Как и люди, LLM часто предпочитают более длинные ответы, недооценивая короткие и лаконичные, но качественные варианты.
- *Внутрисемейная (family bias)*. LLM могут отдавать предпочтение ответам, сгенерированным ими самими или другими моделями из того же семейства, вероятно, из-за сходства языковых шаблонов или баз знаний.

Компенсировать указанные виды предвзятости и повысить качество наборов данных о предпочтениях можно несколькими способами. Чтобы справиться с позиционной предвзятостью, можно сделать случайным порядок предъявления ответов А и Б в ходе сравнения. Другой важной стратегией является few-shot-подход, в рамках которого модели предоставляется несколько примеров, демонстрирующих сбалансированное распределение оценок. Эти примеры помогают откалибровать внутренние механизмы оценки LLM-эксперта и позволяют справиться с двумя другими видами предвзятости, показывая, что короткие или сгенерированные моделями из других семейств ответы тоже могут быть качественными. Кроме того, использование нескольких моделей в качестве «жюри» вместо одного LLM-эксперта может значительно повысить надежность процесса оценки. Такой подход позволяет уравновесить предвзятости, присущие отдельным моделям, что способствует всесторонней и более точной оценке ответов.

В следующем разделе мы создадим собственный набор данных о предпочтениях. Мы будем опираться на процесс генерации, при котором предпочтительные ответы создаются человеком, а отклоненные — LLM.

Создание собственного набора данных о предпочтениях

В настоящий момент наша модель умеет писать абзацы текста на темы, связанные с машинным обучением, однако ее стиль изложения не совпадает со стилем авторов оригинальных статей. Мы можем выполнить согласование предпочтений и изменить стиль письма модели так, чтобы он максимально соответствовал стилю исходных текстов. Важно отметить, что на практике метод DPO нередко делает ответы модели более многословными и побуждает ее к использованию чрезмерно формального языка. Поэтому в нашем случае данный подход следует применять предельно избирательно, чтобы модель избежала этой ловушки и усвоила более неформальный стиль, характерный для оригинальных статей.

В текущем разделе мы создадим набор данных, состоящий из предпочтительных ответов, представляющих собой фрагменты исходного текста, и отклоненных ответов, сгенерированных моделью. Для этого мы модифицируем код из главы 5, предназначенный для генерации наборов с инструкциями.

Как мы уже видели в предыдущем разделе, наборы данных о предпочтениях создаются по тому же принципу, что и наборы с инструкциями, только вместо пар «инструкция — ответ» они содержат тройки вида «инструкция, ответ 1, ответ 2». В нашем распоряжении уже есть эталонные ответы, взятые из исходных текстовых фрагментов. Это означает, что мы не нуждаемся в сложной процедуре оценки с использованием LLM-эксперта. Чтобы убедиться в том, что выбранные фрагменты действительно являются качественными, мы реализуем два дополнительных фильтра — на основе длины и пунктуации. Рисунок 6.2 иллюстрирует весь процесс в целом.

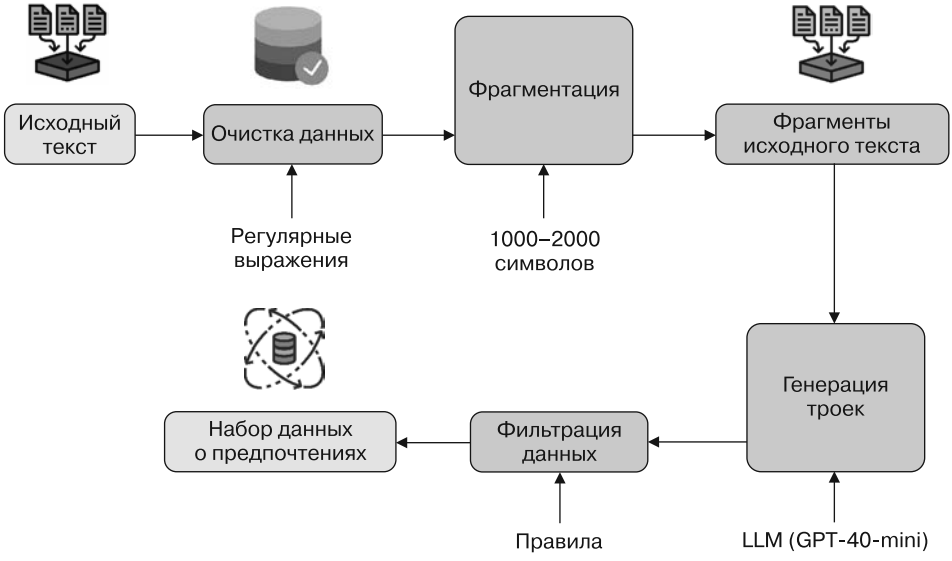


Рис. 6.2. Пайплайн генерации синтетических данных

Теперь мы готовы реализовать пайплайн генерации данных о предпочтениях.

1. Сначала импортируем необходимые библиотеки:

```
import concurrent.futures
import json
import re
from typing import List, Tuple
from datasets import Dataset
from openai import OpenAI
from tqdm.auto import tqdm
```

2. Вместо класса `InstructionAnswerSet` теперь используется класс `PreferenceSet`. Он предназначен для обработки троек, включающих в себя инструкцию, сгенерированный моделью (отклоненный) ответ и извлеченный из текста (предпочтительный) ответ:

```
class PreferenceSet:
    def __init__(self, triples: List[Tuple[str, str, str]]):
        self.triples = triples

    @classmethod
    def from_json(cls, json_str: str) -> 'PreferenceSet':
        data = json.loads(json_str)
        triples = [(triple['instruction'], triple['generated_answer'],
        ➥triple['extracted_answer'])
            for triple in data['preference_triples']]
        return cls(triples)

    def __iter__(self):
        return iter(self.triples)
```

3. Функции `load_articles_from_json`, `clean_text` и `extract_substrings` остаются такими же, как и в исходном коде. Функция `load_articles_from_json` принимает на вход JSON-файл (`cleaned_documents.json`), содержащий статьи, и возвращает набор данных Hugging Face с текстом и метаданными (ID, платформа, ID автора, полное имя автора, ссылка):

```
def load_articles_from_json(file_path: str) -> Dataset:
    with open(file_path, "r") as file:
        data = json.load(file)

    return Dataset.from_dict(
        {
            "id": [item["id"] for item in data["artifact_data"]],
            "content": [item["content"] for item in
            ➥data["artifact_data"]],
            "platform": [item["platform"] for item in
            ➥data["artifact_data"]],
            "author_id": [item["author_id"] for item in
            ➥data["artifact_data"]],
            "author_full_name": [item["author_full_name"] for item in
            ➥data["artifact_data"]],
            "link": [item["link"] for item in data["artifact_data"]],
        }
    )
```

4. Функция `clean_text` удаляет все символы, кроме буквенно-цифровых, апострофов, точек, запятых, восклицательных и вопросительных знаков. Она также заменяет повторяющиеся пробельные символы одним пробелом, чтобы гарантировать корректное форматирование текста:

```
def clean_text(text: str) -> str: text = re.sub(r"[^\w\s.,!?']",
➥" ", text)    text = re.sub(r"\s+", " ", text)
    return text.strip()
```

5. Функция `extract_substrings` разбивает статьи на фрагменты длиной от 1000 до 2000 символов. Чтобы избежать разрыва предложений, способного исказить смысл, мы используем регулярное выражение, допускающее разбиение только после окончания предложения:

```
def extract_substrings(dataset: Dataset, min_length: int = 1000,
➥max_length: int = 2000) -> List[str]:
    extracts = []
    sentence_pattern = r"(?<!\w\.\w.)(?<![A-Z][a-z]\.)(?<=\.|\?|\!)\s"

    for article in dataset["content"]:
        cleaned_article = clean_text(article)
        sentences = re.split(sentence_pattern, cleaned_article)

        current_chunk = ""
        for sentence in sentences:
            sentence = sentence.strip()
            if not sentence:
                continue

            if len(current_chunk) + len(sentence) <= max_length:
                current_chunk += sentence + " "
            else:
                if len(current_chunk) >= min_length:
                    extracts.append(current_chunk.strip())
                    current_chunk = sentence + " "

            if len(current_chunk) >= min_length:
                extracts.append(current_chunk.strip())

        return extracts
```

6. Функция `generate_preference_triples` заменяет прежнюю функцию `generate_instruction_answer_pairs`. Новый промпт адаптирован с учетом необходимости генерации троек вместо пар. Он также содержит общие указания, сообщающие модели о том, какие инструкции нас интересуют, как извлекать ответы из статьи и в каком стиле они должны быть оформлены:

```
def generate_preference_triples(extract: str, client: OpenAI) ->
➥List[Tuple[str, str, str]]:
    prompt = f"""На основе следующего отрывка составь пять троек
«инструкция — ответ». Каждая тройка должна состоять из:
1. Инструкции, в которой задается конкретная тема в контексте.
2. Сгенерированного ответа, который пытается ответить на инструкцию
с учетом контекста.
3. Извлеченного ответа, который представляет собой соответствующий
отрывок, непосредственно из заданного контекста.

Инструкции должны быть самодостаточными и общими, без явного упоминания
контекста, системы, курса или отрывка.
```

```
Важно:
- Убедись, что извлеченный ответ является дословной копией контекста,
включая все знаки препинания и апострофы.
- Не добавляй многоточие в виде (...) или [...] для обозначения
пропущенного текста в извлеченном ответе.
- Если соответствующий текст не является непрерывным, используй два
отдельных предложения из контекста вместо пропуска текста.

Представь свой ответ в формате JSON со следующей структурой:
            {{
                "preference_triples": [
                    {{
                        "instruction": "...",
                        "generated_answer": "...",
                        "extracted_answer": "..."
                    }},
                    ...
                ]
            }}

                Фрагмент:
                {extract}
            """
```

7. В этой же функции мы используем модель GPT-4o-mini в режиме JSON для генерации ответов. В системном промпте мы указываем, что нам требуются именно тройки, а не пары. Ответы в формате JSON разбираются с помощью класса `PreferenceSet`, который возвращает ожидаемый список кортежей:

```
completion = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {
            "role": "system",
            "content": "Ты — полезный помощник, который генерирует
тройки «инструкция — ответ» на основе заданного контекста. Каждая
тройка должна включать инструкцию, сгенерированный ответ и ответ,
извлеченный из контекста. Предоставь ответ в формате JSON.",
        },
        {"role": "user", "content": prompt},
    ],
    response_format={"type": "json_object"},
    max_tokens=2000,
    temperature=0.7,
)

result = PreferenceSet.from_json(completion.choices[0].message.content)
return result.triples
```

8. Помимо всего прочего, в пайплайн были добавлены две новые функции фильтрации: `filter_short_answers` и `filter_answer_format`. Первая отбрасывает слишком короткие ответы, а вторая гарантирует, что ответы начинаются

с заглавной буквы и завершаются корректным знаком препинания. Эти эвристики используются для исключения некачественных примеров.

```
def filter_short_answers(dataset: Dataset, min_length: int = 100) -> Dataset:
    def is_long_enough(example):
        return len(example['chosen']) >= min_length

    return dataset.filter(is_long_enough)

def filter_answer_format(dataset: Dataset) -> Dataset:
    def is_valid_format(example):
        chosen = example['chosen']
        return (len(chosen) > 0 and
            chosen[0].isupper() and
            chosen[-1] in ('.', '!', '?'))

    return dataset.filter(is_valid_format)
```

9. Функция `create_preference_dataset` заменяет прежнюю функцию `create_instruction_dataset`. Она работает с тройками, а не с парами, и использует другие названия столбцов в результирующем наборе данных:

```
def create_preference_dataset(dataset: Dataset, client: OpenAI,
➥num_workers: int = 4) -> Dataset:
    extracts = extract_substrings(dataset)
    preference_triples = []

    with concurrent.futures.ThreadPoolExecutor(max_workers=num_workers)
    ➥as executor:
        futures = [
            executor.submit(generate_preference_triples, extract, client)
            for extract in extracts
        ]
        for future in tqdm(concurrent.futures.as_completed(futures),
        ➥total=len(futures)):
            preference_triples.extend(future.result())

    instructions, generated_answers, extracted_answers =
    ➥zip(*preference_triples)
    return Dataset.from_dict(
        {
            "prompt": list(instructions),
            "rejected": list(generated_answers),
            "chosen": list(extracted_answers)
        }
    )
```

10. Функция `main` обновлена с учетом новых шагов фильтрации и использования функции, предназначенной для генерации набора данных о предпочтениях:

```
def main(dataset_id: str) -> Dataset:
    client = OpenAI()

    # 1. Загрузка исходных данных
    raw_dataset = load_articles_from_json("cleaned_documents.json")
    print("Raw dataset:")
    print(raw_dataset.to_pandas())
```

```
    # 2. Создание набора данных о предпочтениях
    dataset = create_preference_dataset(raw_dataset, client)
    print("Preference dataset:")
    print(dataset.to_pandas())

    # 3. Отбрасывание коротких ответов
    dataset = filter_short_answers(dataset)

    # 4. Отбрасывание ответов на основе формата
    dataset = filter_answer_format(dataset)

    # 5. Экспорт
    dataset.push_to_hub(dataset_id)

    return dataset
```

Функция `create_preference_dataset()` сгенерировала 2970 примеров. Затем этот набор данных был отфильтрован, и в итоговую версию вошли только 1467 образцов — при этом были удалены слишком короткие или неправильно отформатированные ответы (например, начинающиеся не с заглавной буквы или не оканчивающиеся точкой, восклицательным либо вопросительным знаком).

Итоговый набор данных доступен на платформе Hugging Face Hub по адресу https://huggingface.co/datasets/mlabonne/llmtwin-dpo. На рис. 6.3 представлен пример, иллюстрирующий тонкое стилистическое различие между ответами. Оба ответа технически корректны, но *предпочтительный* (chosen) вариант звучит чуть менее формально.



prompt string · lengths 60–79 20.5%	rejected string · lengths 101–136 32.4%	chosen string · lengths 100–155 63.4%
What is the purpose of instruction datasets in guiding language models?	Instruction datasets are designed to efficiently guide a language model toward a specific task, such as news classification.	Instruction datasets offer an efficient way to guide a language model toward a specific task like news classification.
What components will you learn to build in the Hands on LLMs course?	In the Hands on LLMs course, you will learn to build a real-time streaming pipeline, a fine-…	There are 3 components you will learn to build during the course n a real time streaming…
What is the purpose of the function fetch_all_cleaned_content?	The function fetch_all_cleaned_content is used to efficiently retrieve a list of cleaned…	To easily fetch data from a Qdrant database, you can utilize the Python function…
What does the scroll method do in the context of data retrieval?	The scroll method is used to start scrolling through the database, allowing for the…	Initialize the Scroll Start by scrolling through the database using the scroll method,…

Рис. 6.3. Снимок экрана Hugging Face Hub с набором данных mlabonne/llmtwin-dpo

В процессе работы над набором данных мы многократно адаптировали промпт, используя ручную проверку и серию экспериментов, чтобы добиться нужных результатов. Качество промпта играет ключевую роль на этом этапе, поэтому при создании собственного набора данных о предпочтениях рекомендуется применять аналогичную итеративную стратегию.

В следующем разделе мы рассмотрим основные понятия, связанные с такими подходами, как *обучение с подкреплением на основе обратной связи от человека* (RLHF) и прямая оптимизация предпочтений (DPO). Мы также познакомимся с новыми параметрами и идеями, которые будут реализованы в заключительной части главы.

Согласование предпочтений

Согласование предпочтений охватывает совокупность методов тонкой настройки моделей на основе данных о предпочтениях пользователей. В этом разделе мы кратко рассмотрим это направление, а затем подробно обсудим и реализуем на практике такую технику, как *прямая оптимизация предпочтений* (Direct Preference Optimization, DPO).

Обучение с подкреплением на основе обратной связи от человека

Обучение с подкреплением на основе обратной связи от человека (Reinforcement Learning from Human Feedback, RLHF) — это метод, сочетающий алгоритмы *обучения с подкреплением* (Reinforcement Learning, RL) с использованием обратной связи, предоставляемой людьми, для согласования поведения моделей с человеческими предпочтениями и ценностями. Метод RLHF был призван решить проблемы, присущие традиционным методам обучения с подкреплением, в частности, связанные с трудностями при определении функций вознаграждения для сложных задач и риском рассогласования между вручную заданными вознаграждениями и фактическими целями.

Этот подход базируется на идее *обучения с подкреплением на основе предпочтений* (Preference-based Reinforcement Learning, PbRL), которая была описана еще в 2011 году. Метод PbRL был ориентирован на определение целевой функции на основе такой качественной обратной связи, как предпочтения того или иного варианта поведения, вместо использования количественных оценок вознаграждения. Этот подход позволял обойти некоторые ограничения классических методов обучения с подкреплением, в которых определение соответствующих функций вознаграждения бывает затруднено и сопряжено с риском манипуляции системой вознаграждения (reward hacking) и возникновения нежелательного поведения.

Термин RLHF появился позднее — примерно в 2021–2022 годах, когда данный подход получил широкое распространение в сфере обучения больших языковых моделей. Однако основные идеи сформировались задолго до этого. Основополагающая работа, опубликованная в 2017 году, продемонстрировала эффективность обучения моделей вознаграждения на основе человеческих пред-

почтений и их последующего использования для обучения RL-агентов. В ней доказано, что RLHF позволяет модели достигать уровня производительности, сопоставимого и даже превосходящего уровень агентов, обученных на вручную заданных функциях вознаграждения, при значительно меньших усилиях со стороны человека.

В основе метода RLHF лежит итеративная схема обучения, предполагающая поочередное улучшение модели вознаграждения (reward model) и основной модели, генерирующей текст.

- *Обучение модели вознаграждения*. В рамках RLHF модель вознаграждения обучается не на заранее заданной функции вознаграждения, а на основе обратной связи, полученной от человека. Обычно для этого человеку показывают несколько вариантов ответов, из которых он выбирает предпочтительный. Затем эти предпочтения используются для обучения модели, чаще всего с применением модели Бредли — Терри или аналогичных подходов.
- *Оптимизация основной модели*. После обучения модели вознаграждения можно задействовать стандартные алгоритмы обучения с подкреплением для оптимизации политики, которая генерирует новое поведение, чтобы получить максимальное вознаграждение от модели.
- *Итеративное улучшение*. По мере улучшения основной модели она начинает генерировать новые формы поведения, которые могут быть оценены человеком, что позволяет уточнить модель вознаграждения. Этот цикл многократно повторяется, в идеале приводя к формированию такой модели генерации текста, которая хорошо согласуется с человеческими предпочтениями.

Важным новшеством подхода RLHF является решение проблемы дороговизны получения обратной связи от человека. Вместо того чтобы полагаться на постоянный контроль со стороны людей, RLHF допускает асинхронную и редкую (например, на каждый 100-й текст) обратную связь. Обученная модель вознаграждения выступает в роли прокси человеческих предпочтений, что позволяет RL-алгоритму продолжать обучение, не нуждаясь в прямом участии человека на каждом шаге.

На рис. 6.4 представлена общая схема работы алгоритма *оптимизации с использованием модели вознаграждения* (Proximal Policy Optimization, PPO) — одного из наиболее популярных алгоритмов RLHF. На этой схеме модель вознаграждения используется для оценки текста, сгенерированного обучаемой моделью.

Хотя RLHF доказал свою эффективность в согласовании поведения ИИ-систем с предпочтениями людей, он сталкивается с рядом трудностей из-за своей итеративной природы и зависимости от отдельной модели вознаграждения, которая может быть ресурсоемкой и потенциально нестабильной. Несмотря на

теоретическое превосходство, в ходе экспериментов алгоритмы RLHF нередко показывают менее убедительные результаты по сравнению с более простыми подходами. Одним из таких подходов, привлекших значительное внимание, является прямая оптимизация предпочтений.

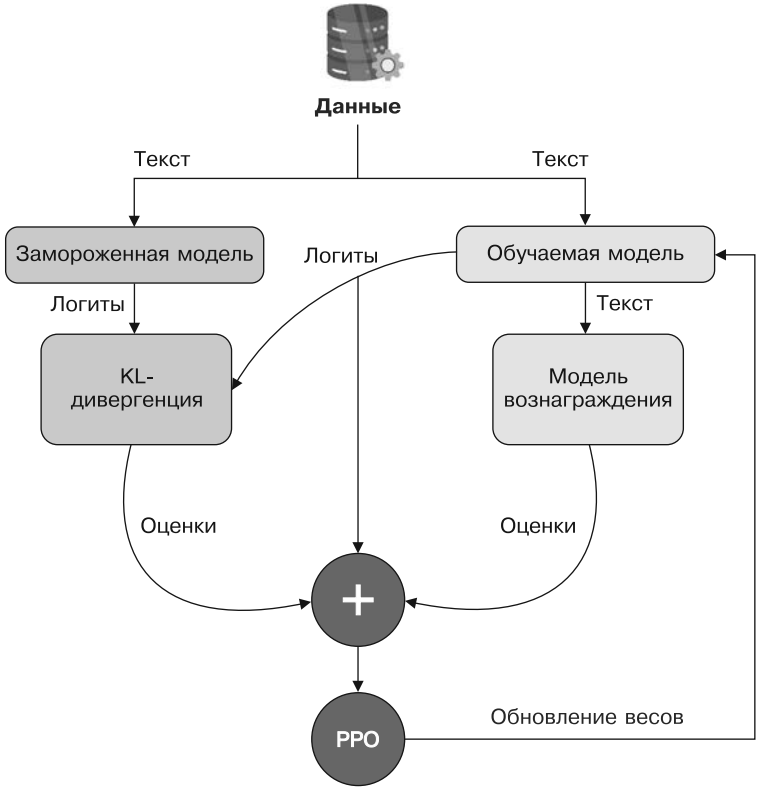


Рис. 6.4. Схема работы алгоритма PPO, используемого для согласования предпочтений

Прямая оптимизация предпочтений

Метод прямой оптимизации предпочтений (Direct Preference Optimization, DPO) был предложен Рафаиловым и соавторами в статье *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*, опубликованной в 2023 году, и представляет собой упрощенную альтернативу традиционным методам RLHF.

Главное новшество DPO заключается в иной формулировке задачи изучения предпочтений. В отличие от RLHF, который предполагает обучение отдельной

модели вознаграждения и последующую тонкую настройку языковой модели с помощью RL-алгоритма (вроде PPO), DPO использует более прямой подход.

Он выводит выражение в замкнутой форме для оптимальной политики в рамках стандартной задачи RLHF, состоящей в максимизации ожидаемого вознаграждения при ограничении на дивергенцию Кульбака — Лейблера относительно эталонной политики. Это позволяет методу DPO формулировать задачу обучения предпочтениям непосредственно в терминах политики, устраняя необходимость в отдельной модели вознаграждения и использовании сложных алгоритмов обучения с подкреплением.

На практике DPO можно реализовать в виде простой функции потерь, такой как бинарная кросс-энтропия, применяемой непосредственно к выходным вероятностям языковой модели. Эта функция побуждает модель присваивать более высокую вероятность предпочтительным ответам и более низкую — отклоненным, сохраняя близость к эталонной (замороженной) модели. Влияние эталонной модели контролируется с помощью параметра β (бета), принимающего значения от 0 до 1. Если $\beta = 0$, эталонная модель полностью игнорируется, что позволяет тонко настроенной модели значительно отклониться от исходной SFT-версии. На практике наиболее часто используется значение $\beta = 0{,}1$, но его можно адаптировать, как будет показано в следующем разделе.

Простота метода DPO позволяет оптимизировать модель с помощью стандартных методов градиентного спуска без необходимости выборки ответов из модели во время обучения и реализации сложных RL-алгоритмов. На рис. 6.5 представлена высокоуровневая схема работы алгоритма DPO, значительно упрощающая процесс обучения в рамках PPO, представленный на рис. 6.4.

Метод прямой оптимизации предпочтений (DPO) обладает рядом преимуществ по сравнению с традиционными подходами RLHF. Как уже упоминалось, он существенно упрощает пайплайн обучения предпочтениям, снижая присущую RLHF инженерную сложность. Поскольку DPO не требует использования отдельной модели вознаграждения и сложных RL-алгоритмов, он является более эффективным с точки зрения вычислений. Это особенно заметно при применении адаптеров (например, в рамках подходов LoRA или QLoRA): в этом случае исходная (замороженная) и тонко настраиваемая модели не разделяются, так как обновляются только параметры новых слоев, а сама модель остается неизменной. Это позволяет экономить видеопамять за счет загрузки только одной модели вместо двух.

Несмотря на простоту, метод DPO зачастую демонстрирует результаты, сопоставимые с более сложными методами RLHF. Кроме того, он лучше поддерживает стабильность в процессе обучения и менее чувствителен к выбору гиперпараметров. Благодаря этой простоте DPO легче реализовать и масштабировать, особенно небольшим командам, не обладающим глубокими знаниями в области обучения с подкреплением.



Рис. 6.5. Схема работы алгоритма DPO, используемого для согласования предпочтений

Хотя RLHF предусматривает итеративное улучшение за счет выполнения множества циклов обучения и динамическую адаптацию к новым предпочтениям, DPO предлагает более прямой путь к достижению сопоставимых результатов. Выбор между DPO и RLHF на основе PPO чаще всего сводится к компромиссу между простотой реализации и возможностью достижения более высокой производительности. При масштабном обучении на миллионах пар предпочтений методы, основанные на PPO, по-прежнему отличаются более высоким пределом производительности. Однако в большинстве прикладных сценариев DPO обеспечивает приемлемый уровень производительности при меньших вычислительных и инженерных затратах.

И RLHF, и DPO в значительной степени выигрывают от использования синтетических данных. По мере роста возможностей LLM они становятся способными генерировать обучающие данные, превосходящие по качеству и разнообразию тексты, созданные людьми. Это создает петлю положительной обратной связи, при которой лучшие модели создают более качественные данные, которые, в свою очередь, способствуют дальнейшему улучшению моделей. Итеративная природа обоих подходов позволяет проводить несколько раундов доработки модели, каждый раз фокусируясь на разных аспектах ее производительности и постепенно расширяя ее возможности в различных областях.

Тем не менее у DPO есть и определенные ограничения. Как и RLHF, он требует данных с оценками о предпочтениях, сбор которых может оказаться дорогостоящим и трудоемким. Кроме того, метод DPO не обладает рядом теоретических гарантий (например, получение LLM максимальной оценки от модели

вознаграждения), присущих подходам на основе обучения с подкреплением. В некоторых случаях дополнительная гибкость (например, самим запускать дообучение LLM, когда это нужно), которую обеспечивает RLHF, может быть полезной, особенно если речь идет о сложных задачах или средах.

Тем не менее DPO является оптимальным решением в большинстве сценариев, включая нашу задачу обучения LLM-двойника. В следующем разделе мы реализуем этот подход с помощью библиотеки Unsloth.

Реализация метода DPO

В этом разделе мы тонко настроим модель TwinLlama-3.1-8B, созданную в главе 5, методом DPO. Для удобства и достижения максимального уровня производительности мы вновь воспользуемся библиотекой Unsloth. В зависимости от доступных ресурсов вы можете выбрать подход LoRA (более высокое качество, скорость и потребление видеопамяти) или QLoRA (менее высокое качество, скорость и потребление видеопамяти). Метод DPO и другие алгоритмы согласования предпочтений также поддерживаются библиотеками TRL и Axolotl.

Данный пример можно рассматривать в качестве продвинутого варианта реализации DPO. Дело в том, что поставленная нами задача — имитировать определенный стиль письма — вступает в некоторое противоречие с естественной склонностью алгоритма DPO поощрять использование более формального языка. Это объясняется тем, что в обучающих наборах предпочтительные ответы, как правило, оказываются более формальными. В таких случаях необходимо выполнить тонкую настройку модели, задав низкую скорость обучения и небольшое число эпох. Для подбора оптимальных гиперпараметров мы обучили более 20 моделей и сравнили их ответы на набор вопросов, включая следующий: «Напиши абзац, объясняющий суть тонкой настройки с учителем». Это позволило нам выбрать модель и параметры, дающие наилучшие результаты при решении поставленной задачи.

В данном случае зависимости совпадают с теми, что использовались в главе 5 при реализации SFT и доступны в GitHub-репозитории книги (https://github.com/PacktPublishing/LLM-Engineering), а также в репозитории Unsloth (https://github.com/unslothai/unsloth).

1. Для начала необходимо получить доступ к закрытой модели и (при необходимости) подготовить возможность загрузки дообученной модели на платформу Hugging Face (https://huggingface.co/). Для этого потребуется авторизация в аккаунте. Если у вас еще нет аккаунта, его можно создать и сохранить API-ключ (Settings ▸ Access Tokens ▸ Create new token (Настройки ▸ Токены доступа ▸ Создать новый токен)) в файле `.env`:

```
HF_TOKEN = YOUR_API_KEY
```

2. Убедитесь, что в файле `.env` также содержится ваш API-ключ Comet ML. В противном случае при запуске процесса обучения выполнение кода завершится с ошибкой.

```
COMET_API_KEY = YOUR_API_KEY
```

3. Перед импортом необходимых библиотек следует применить исправление для класса `DPOTrainer` из библиотеки TRL, которое решает проблему с DPO-логами в Jupyter Notebook и аналогичных средах:

```
from unsloth import PatchDPOTrainer
PatchDPOTrainer()
```

4. Теперь можно импортировать остальные библиотеки. Основное отличие от примера с SFT заключается в использовании классов `DPOConfig` и `DPOTrainer` из TRL, предназначенных специально для обучения моделей с применением метода DPO:

```
import os
import torch
from datasets import load_dataset
from transformers import TrainingArguments, TextStreamer
from unsloth import FastLanguageModel, is_bfloat16_supportedfrom trl
import DPOConfig, DPOTrainer
```

5. На этом этапе мы загружаем модель, тонко настроенную в главе 5, задействуя те же параметры, в том числе максимальную длину последовательности, равную 2048. При необходимости можно активизировать режим QLoRA, задав значение `True` для параметра `load_in_4bit`. В нашем случае мы будем использовать подход LoRA для повышения скорости и качества тонкой настройки:

```
max_seq_length = 2048
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="mlabonne/TwinLlama-3.1-8B",
    max_seq_length=max_seq_length,
    load_in_4bit=False,
)
```

6. Теперь давайте подготовим модель к эффективной тонкой настройке на основе параметров с помощью конфигурации LoRA. Для получения более выразительных результатов мы увеличиваем значения `r` и `lora_alpha` с `32` (как это было в главе 5) до `64`. Параметр `dropout` оставляем равным нулю для повышения скорости и, как обычно, нацеливаемся на все линейные модули:

```
model = FastLanguageModel.get_peft_model(
    model,
    r=64,
    lora_alpha=64,
    lora_dropout=0,
    target_modules=["q_proj", "k_proj", "v_proj", "up_proj",
    ➥"down_proj", "o_proj", "gate_proj"],
)
```

7. Далее мы загружаем тренировочную часть набора данных `llmtwin-dpo`, содержащую промпты, выбранные и отклоненные ответы:

```
dataset = load_dataset("mlabonne/llmtwin-dpo", split="train")
```

8. Процесс подготовки данных в этом примере существенно отличается от описанного в главе 5. Здесь используются тройки, состоящие из промпта, выбранного и отклоненного ответов. В функции `format_samples` мы применяем к промпту шаблон ведения диалога Alpaca. Обратите внимание, что выбранный и отклоненный ответы просто конкатенируются с *токеном конца последовательности* (`EOS_TOKEN`). После этого данные разбиваются на обучающую и тестовую выборки в пропорции 95 к 5 %:

```
alpaca_template = """ Ниже приведена инструкция, описывающая задачу.
                  Напиши ответ, который правильно дополнит запрос.

                  ### Инструкция:
                  {}

                  ### Ответ:
                  """

EOS_TOKEN = tokenizer.eos_token

def format_samples(example):
    example["prompt"] = alpaca_template.format(example["prompt"])
    example["chosen"] = example['chosen'] + EOS_TOKEN
    example["rejected"] = example['rejected'] + EOS_TOKEN
    return {"prompt": example["prompt"], "chosen":
example["chosen"], "rejected": example["rejected"]}

dataset = dataset.map(format_samples)

dataset = dataset.train_test_split(test_size=0.05)
```

9. Теперь, когда модель и данные готовы, можно приступать к тонкой настройке. По сравнению с SFT-примером в данном случае используется несколько новых параметров, в частности `ref_model` и `beta`. Поскольку мы применяем подход LoRA (или QLoRA), напрямую обучаются не все параметры модели, а только адаптеры. Это позволяет нам задействовать исходную модель (без адаптеров) в качестве эталонной и тем самым существенно экономить видеопамять. Параметр `beta` управляет степенью влияния эталонной модели. Обычно для него задается значение `0.1`, которое хорошо работает в большинстве случаев. Однако, исходя из результатов наших экспериментов, мы решили увеличить его до `0.5`. Это связано с тем, что при меньших значениях обученная модель склонна к чрезмерной формальности; повышение значения `beta` позволяет приблизить стиль к эталонному.

   Мы также понизили скорость обучения, по сравнению с SFT-примером, с `3e-4` до `2e-6` и уменьшили число эпох с `3` до `1`. Параметр `max_seq_length`

был разделен на `max_prompt_length` (только для промпта) и `max_length` (для промпта и ответа), а класс `TrainingArguments` — заменен на `DPOConfig`:

```
trainer = DPOTrainer(
    model=model,
    ref_model=None,
    tokenizer=tokenizer,
    beta=0.5,
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    max_length=max_seq_length//2,
    max_prompt_length=max_seq_length//2,
    args=DPOConfig(
        learning_rate=2e-6,
        lr_scheduler_type="linear",
        per_device_train_batch_size=2,
        per_device_eval_batch_size=2,
        gradient_accumulation_steps=8,
        num_train_epochs=1,
        fp16=not is_bfloat16_supported(),
        bf16=is_bfloat16_supported(),
        optim="adamw_8bit",
        weight_decay=0.01,
        warmup_steps=10,
        output_dir="output",
        eval_strategy="steps",
        eval_steps=0.2,
        logging_steps=1,
        report_to="comet_ml",
        seed=0,
    ),
)
trainer.train()
```

10. После завершения обучения мы можем выполнить быструю проверку корректности работы модели. Как и в примере с SFT, на этом этапе модель подготавливается к инференсу и генерирует ответ на заданный промпт:

```
FastLanguageModel.for_inference(model)
message = alpaca_template.format("Напиши абзац, в котором
➥объясняется, что такое тонкая настройка с учителем.", "")
inputs = tokenizer([message], return_tensors="pt").to("cuda")

text_streamer = TextStreamer(tokenizer) _ = model.generate(**inputs,
➥streamer=text_streamer, max_new_tokens=256, use_cache=True)
```

11. Обученная с помощью DPO модель возвращает следующий ответ:

```
Тонкая настройка с учителем — это метод, который используется для повышения
эффективности предварительно обученных языковых моделей с помощью размеченных
данных. Этот метод предполагает использование предварительно обученной модели
и ее доработку для решения конкретной задачи, например создания контента или
обслуживания клиентов. Предоставляя модели соответствующие данные и инструкции,
можно научить ее генерировать результаты, которые больше будут соответствовать
желаемым. Такой подход позволяет создавать более специализированные модели,
способные решать сложные задачи с большей точностью и эффективностью.
```

Мы можем сравнить этот ответ с вариантом, сгенерированным моделью, тонко настроенной методом SFT:

```
Тонкая настройка с учителем — это метод, который используется для улучшения
языковой модели путем предоставления ей тщательно отобранного набора
данных с инструкциями и соответствующими ответами. Этот процесс направлен
на то, чтобы ответы модели соответствовали ожиданиям человека, тем самым
повышая ее точность и релевантность. Цель состоит в том, чтобы модель
могла эффективно отвечать на широкий спектр запросов, что делает ее ценным
инструментом для таких приложений, как чат-боты и виртуальные помощники.
```

Модель, тонко настроенная с помощью DPO, дает ответ, который одновременно является более точным и приближенным к требуемому стилю письма. В частности, она правильно указывает на то, что предварительно обученные языковые модели служат основой для SFT, и акцентирует внимание на тонкой настройке под конкретную предметную область или задачу, а не на абстрактном «соответствии с ожиданиями человека», что точнее отражает суть этапа согласования предпочтений. Кроме того, ответ этой модели менее формален и лучше подходит для публикации в блоге.

12. На заключительном этапе мы сохраняем обученную модель локально и на платформе Hugging Face Hub:

```
model.save_pretrained_merged("model", tokenizer, save_
➥method="merged_16bit")
```

Итак, мы обучили и экспортировали нашу модель, тонко настроенную методом DPO. Теперь она доступна на платформе Hugging Face Hub по адресу https://huggingface.co/mlabonne/TwinLlama-3.1-8B-DPO. В отличие от SFT метод DPO позволяет отслеживать несколько дополнительных метрик в процессе обучения. На рис. 6.6 показан дашборд Comet ML с основными показателями. Ознакомиться с ним можно по ссылке https://www.comet.com/mlabonne/llm-twin-training/.

Рассмотрим эти ключевые метрики.

- *Значение функции потерь на обучающей выборке (training loss)*. Как и прежде, мы ожидаем, что значение функции потерь будет в среднем снижаться по мере обучения модели. Резкое обнуление этого показателя означает, что модель прекратила обучаться. Такое поведение не обязательно свидетельствует о переобучении или плохом качестве модели, но требует контроля.
- *Значение функции потерь на валидационной выборке (validation loss)*. В данном случае мы также ожидаем снижения, при этом разница между значениями потерь на обучающей и валидационной выборках должна оставаться небольшой.
- *Норма градиента (gradient norm)*. Мы ожидаем невысокие значения нормы градиента с редкими всплесками.
- *Вознаграждения (rewards)*. В данном случае мы отслеживаем два вида вознаграждения — за выбранные и отклоненные ответы. Эти вознаграждения представляют собой среднюю разность логарифмов вероятностей, предсказанных обученной и эталонной моделями. Со временем мы ожидаем,

что модель будет все чаще предпочитать выбранные ответы и отвергать отклоненные, что должно приводить к увеличению разрыва между этими значениями. Такая разность напрямую отслеживается с помощью метрики `margins`, которая определяется как разность между вознаграждениями за выбранные и отклоненные ответы. У хорошо обученной модели значение данного показателя быстро растет, а затем стабилизируется.

- *Точность (accuracies)*. Эта метрика отражает процент случаев, в которых модель правильно определяет выбранные ответы. Нам нужно, чтобы точность постепенно увеличивалась в ходе обучения, но не достигала 100 %. Если точность достигает 100 %, особенно на ранних этапах, это может свидетельствовать о том, что набор данных о предпочтениях является слишком простым для модели. В таком случае имеет смысл добавить в него более сложные примеры.

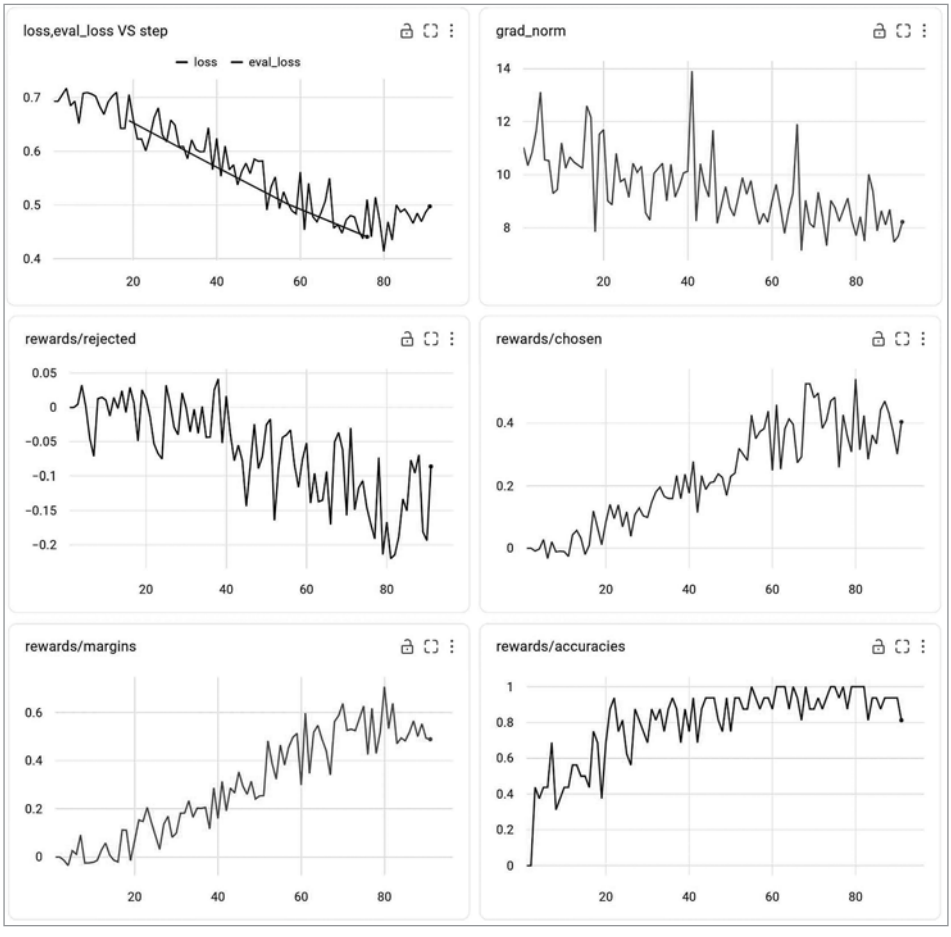


Рис. 6.6. Метрики DPO, отслеживаемые в Comet ML при тонкой настройке модели

В целом метод DPO является чуть более сложным в плане отслеживания и отладки по сравнению с SFT, поскольку предусматривает дополнительный компонент в виде эталонной модели. Однако по сравнению с PPO и другими RLHF-алгоритмами он значительно проще в использовании. При наличии качественного набора данных о предпочтениях и качественной модели вы можете поэкспериментировать с различными значениями таких параметров, как `rank`, `beta`, скорость обучения и число эпох, чтобы найти конфигурацию, позволяющую модели наилучшим образом усвоить нужные вам предпочтения.

Хотя это выходит за рамки текущей главы, стоит отметить, что оценку моделей, имитирующих определенный стиль письма, можно автоматизировать. Один из возможных подходов сводится к сравнению распределений слов в текстах, сгенерированных различными моделями (SFT и DPO), с эталонным набором. В данном примере можно ожидать, что модель, тонко настроенная методом SFT, будет чаще использовать слова, характерные для GPT-4o-mini. Распределение, полученное от модели DPO, должно быть значительно ближе к предпочтительным ответам.

Резюме

В этой главе мы рассмотрели методы согласования предпочтений, направленные на улучшение качества LLM. Мы начали с обсуждения соответствующего набора данных, его структуры и значения для улавливания тонких человеческих предпочтений. Затем мы реализовали пайплайн генерации пользовательских пар предпочтений путем сравнения оригинальных и сгенерированных ИИ текстов. Этот пайплайн можно адаптировать под собственные нужды.

Мы также обсудили эволюцию методов обучения с подкреплением на основе обратной связи от человека (RLHF), которая привела к появлению более простого и эффективного подхода, называемого прямой оптимизацией предпочтений (DPO). В завершение главы мы реализовали метод DPO с помощью библиотеки Unsloth и тонко настроили модель TwinLlama-3.1-8B, созданную в главе 5. Приведенная пошаговая инструкция содержит практические рекомендации по обучению модели и подчеркивает ключевые отличия DPO от SFT. Итоговая версия модели доступна на платформе Hugging Face Hub.

Следующая глава будет посвящена оценке качества больших языковых моделей. Мы обсудим ключевые проблемы и существующие способы анализа производительности LLM, рассмотрим создание предметно-ориентированных оценочных наборов, а также выясним, почему оценка остается одной из наиболее сложных задач в этой сфере. Кроме того, поговорим об использовании мощных моделей (LLM-экспертов) для определения качества более слабых. В завершение главы мы реализуем полноценный пайплайн, обеспечивающий последовательную и эффективную оценку LLM.

Список используемых источников

- *Rafailov R. et al.* Direct Preference Optimization: Your Language Model is Secretly a Reward Model. arXiv preprint arXiv:2305.18290.
- *Kaufmann T. et al.* A Survey of Reinforcement Learning from Human Feedback. arXiv preprint arXiv:2312.14925.
- Anthropic. GitHub — anthropics/hh-rlhf: Human preference data for 'Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback'. github.com. https://github.com/anthropics/hh-rlhf.
- *Stiennon N. et al.* Learning to summarize from human feedback. arXiv preprint arXiv:2009.01325.
- Intel(R) Neural Compressor. Supervised Fine-Tuning and Direct Preference Optimization on Intel Gaudi2. medium.com. https://medium.com/intel-analyticssoftware/the-practice-of-supervised-finetuning-and-direct-preferenceoptimization-on-habana-gaudi2-a1197d8a3cd3.
- Argilla. GitHub — argilla-io/distilabel. github.com. https://github.com/argilla-io/distilabel.
- Databricks. Enhancing LLM-as-a-Judge with Grading Notes. databricks.com. https://www.databricks.com/blog/enhancing-llm-as-a-judge-with-grading-notes.
- *Akrour R., Schoenauer M., Sebag M.* Preference-Based Policy Learning, 2011.
- *Cheng W., Fürnkranz J., Hüllermeier E., Park S.-H.* Preference-Based Policy Iteration: Leveraging Preference Learning for Reinforcement Learning, 2011. P. 312–327. 10.1007/978-3-642-23780-5_30.
- *Christiano P. et al.* Deep reinforcement learning from human preferences. arXiv preprint arXiv:1706.03741.
- *Ouyang L. et al.* Training language models to follow instructions with human feedback. arXiv preprint arXiv:2203.02155.
- *Schulman J. et al.* Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347.
- unslothai. GitHub — unslothai/unsloth: Finetune Llama 3.1, Mistral, Phi & Gemma LLMs 2–5× faster with 80 % less memory. github.com. https://github.com/unslothai/unsloth.

7 Оценка LLM

Оценка больших языковых моделей — это важный процесс, направленный на анализ их производительности и функциональных возможностей. Она может принимать различные формы — от вопросов с несколькими вариантами ответов до выполнения открытых инструкций и сбора обратной связи от реальных пользователей. На сегодняшний день нет единого подхода к измерению качества моделей, однако существуют типовые шаблоны и методы, которые можно адаптировать под определенные задачи.

Хотя наибольшей популярностью пользуются способы оценки общего назначения — такие как бенчмарки *Massive Multi-Task Language Understanding* (MMLU) и LMSYS Chatbot Arena, — для предметно-ориентированных моделей и моделей, направленных на решение конкретных задач, гораздо лучше подходят более специализированные методы. Это особенно актуально в случае полноценных LLM-систем (а не отдельных моделей), например построенных на базе *пайплайна RAG*. В подобных сценариях необходимо расширить рамки оценки, чтобы охватить всю систему в целом, включая такие ее компоненты, как модули поиска и постобработки.

После прочтения этой главы вы будете знать, какие методы оценки LLM наиболее распространены, и научитесь применять различные подходы к оценке как отдельных моделей, так и полноценных RAG-систем.

Оценка моделей

При оценке моделей основная задача — проанализировать возможности отдельной модели без использования инженерии промптов, пайплайнов RAG и других внешних компонентов.

Такая оценка необходима, например, для выбора наиболее подходящей языковой модели или для проверки того, привела ли тонкая настройка к реальному улучшению. В этом разделе мы сравним подходы к оценке LLM и ML-моделей, чтобы выявить ключевые различия между этими направлениями. Затем мы рассмотрим бенчмарки для моделей общего назначения, а также для предметно-ориентированных моделей и направленных на решение конкретных задач.

Оценка LLM и ML-моделей

Оценка ML-моделей обычно сводится к измерению их эффективности при решении таких задач, как предсказание, классификация и регрессия. Если при оценке LLM основное внимание уделяется тому, насколько хорошо модель понимает и генерирует текст, то в случае ML-модели оценивается то, насколько точно и эффективно она обрабатывает структурированные данные для получения конкретных результатов.

Эти различия обусловлены самой природой задач. Модели машинного обучения, как правило, разрабатываются для узкоспециализированных целей, например для предсказания динамики цены акций или выявления выбросов, и чаще всего работают с числовыми или категориальными данными. Это делает процесс их оценки сравнительно прямолинейным. Напротив, LLM предназначены для интерпретации и генерации естественного языка, что привносит в процесс их оценки элемент субъективности. Поэтому, помимо числовых показателей, оценка LLM часто предполагает использование таких качественных критериев, как связность, релевантность и соответствие ответов контексту.

Мы можем выделить три основных различия в принципах работы этих моделей, которые напрямую влияют на процесс их оценки.

- *Числовые метрики*. Оценка ML-моделей обычно основывается на измерении объективных показателей, таких как точность (accuracy), полнота (recall), точность предсказаний (precision) или среднеквадратичная ошибка (MSE), в зависимости от типа задачи. В случае с LLM ситуация менее однозначна, поскольку одна и та же модель может выполнять множество разных задач, требующих разных методов оценки.
- *Генерация признаков*. В традиционном машинном обучении важную роль играет ручной отбор и преобразование признаков перед обучением модели. Успешность этого этапа часто влияет на итоговую оценку модели. В свою очередь, LLM обрабатывают сырой текст напрямую, что устраняет необходимость в дополнительном механизме обработки признаков.
- *Интерпретируемость*. В случае с ML-моделями мы часто способны понять, почему модель сделала те или иные предсказания или классификации, и эта интерпретируемость может являться ключевым элементом ее оценки. Интерпретировать работу LLM гораздо сложнее. Тем не менее мы можем запрашивать у такой модели объяснения в процессе генерации ответов, чтобы получить представление об используемом ею процессе принятия решений.

В следующем разделе мы подробно рассмотрим разные типы LLM. Хотя оценка больших языковых моделей общего назначения существенно отличается от подходов, применяемых в сфере машинного обучения, узкоспециализированные LLM ближе по своей природе к традиционным ML-моделям и оцениваются похожим образом.

Оценка LLM общего назначения

Под оценкой моделей общего назначения понимается применение метрик, специально предназначенных для базовых моделей и тонко настроенных моделей без привязки к конкретной задаче. Эти метрики охватывают широкий спектр способностей, отражающих объем знаний и практическую полезность модели, не фокусируясь на узких сценариях использования. Такой подход позволяет разработчикам получить общее представление о возможностях модели, сравнить ее с конкурентами и выявить ее сильные и слабые стороны. На основе полученных результатов можно скорректировать набор данных, подобрать гиперпараметры или даже изменить архитектуру модели.

Оценку моделей общего назначения можно проводить на трех этапах: во время предварительного обучения, после него и после тонкой настройки.

Во время предварительного обучения важно внимательно отслеживать то, как модель усваивает информацию (как было показано в конце главы 5). На этом этапе используются наиболее простые и низкоуровневые метрики, отражающие сам процесс обучения.

- *Значение функции потерь на обучающей выборке (training loss).* Данная метрика основана на функции потерь, называемой перекрестной (или кросс-) энтропией, и измеряет разницу между предсказанным моделью распределением вероятностей и истинным распределением следующего токена.
- *Значение функции потерь на валидационной выборке (validation loss).* Эта метрика аналогична предыдущей, но вычисляется на отложенной валидационной выборке и используется для оценки того, насколько модель умеет обобщать.
- *Перплексия (perplexity).* Экспонента от перекрестной энтропии, показывающая, насколько модель «удивлена» входными данными (чем ниже это значение, тем лучше).
- *Норма градиента (gradient norm).* Эта метрика позволяет отслеживать величину градиентов во время обучения, способствуя выявлению нестабильности, а также исчезающих или резко растущих градиентов.

На этом этапе также можно задействовать внешние бенчмарки, такие как HellaSwag (проверка на здравый смысл[1]), однако важно учитывать риск переобучения модели на этих тестах.

---

[1] В контексте ИИ это означает способность модели делать логические выводы и предсказания, основываясь на неявных знаниях о физическом мире, человеческом поведении и причинно-следственных связях, которые обычно не требуют специального объяснения. Например, знание о том, что, если был дождь, значит, асфальт должен быть мокрым. — *Примеч. науч. ред.*

После завершения предварительного обучения обычно используется набор тестов для оценки качества базовой модели. Он может включать в себя как внутренние, так и публичные бенчмарки. Ниже приведен неполный перечень наиболее распространенных публичных бенчмарков:

- *MMLU (знания)* — проверяет знания модели в 57 предметных областях (от школьного до профессионального уровня), на основе вопросов с выбором ответа;
- *HellaSwag (рассуждения)* — предлагает модели выбрать наиболее правдоподобное завершение заданной ситуации из нескольких вариантов;
- *ARC-C (рассуждения)* — оценивает знания модели по естественным наукам школьного уровня на основе вопросов с выбором ответа, требующих понимания причинно-следственных связей;
- *Winogrande (рассуждения)* — оценивает способность модели к рассуждению с помощью задач на разрешение местоимений в сложно сформулированных фразах;
- *PIQA (рассуждения)* — проверяет понимание базовых физических законов на основе вопросов о повседневных физических действиях.

Многие из перечисленных выше наборов данных также используются для оценки тонко настроенных моделей общего назначения. В этом случае показатели такой модели сравниваются с метриками базовой LLM. Например, неудачная тонкая настройка может привести к снижению уровня знаний модели, что отразится в виде ухудшения ее результата в тесте MMLU. Напротив, качественная тонкая настройка способна повысить уровень знаний и улучшить результат модели в этом тесте.

Такая оценка также помогает выявить проблему, связанную с тонкой настройкой модели на данных, слишком близких к тестовой выборке. Например, повышение оценки модели в тесте MMLU на десять баллов в ходе тонкой настройки является маловероятным и может указывать на пересечение набора инструкций с тестовой выборкой.

Помимо вышеперечисленных метрик, для оценки тонко настроенных моделей предусмотрены дополнительные бенчмарки. Здесь речь идет о моделях, прошедших *тонкую настройку с учителем* (SFT) и согласование предпочтений. Эти бенчмарки оценивают способность таких моделей понимать вопросы и формулировать осмысленные ответы. Особое внимание уделяется следованию инструкциям, ведению сложного диалога и демонстрации агентных навыков:

- *IFEval (выполнение инструкций)* — оценивает способность модели следовать инструкциям с конкретными ограничениями (например, «не использовать запятые в ответе»);
- *Chatbot Arena (ведение диалога)* — система сравнительной оценки, в которой пользователи голосуют за лучший ответ на инструкцию, сопоставляя реплики двух моделей;

- *AlpacaEval (выполнение инструкций)* — автоматическая оценка тонко настроенных моделей, сильно коррелирующая с результатами Chatbot Arena;
- *MT-Bench (ведение диалога)* — проверяет, как модель ведет сложные диалоги, оценивая ее способность сохранять контекст и давать связные ответы;
- *GAIA (агентные навыки)* — оценивает широкий спектр способностей модели, таких как использование инструментов и веб-поиск, в многоэтапных сценариях.

Понимание того, как устроены и применяются различные методы оценки, играет ключевую роль при выборе оптимальной языковой модели для конкретного случая. Например, перед выполнением тонкой настройки важно выбрать такую базовую модель, которая обладает наилучшим сочетанием знаний и навыков рассуждения при заданном размере. Это позволит вам сравнить производительность различных LLM и выбрать ту, которая обеспечит наиболее прочную основу для последующей тонкой настройки.

Однако даже если вы не собираетесь тонко настраивать модель, бенчмарки вроде Chatbot Arena или IFEval могут пригодиться для сравнения различных моделей, работающих по инструкции. Например, если вы разрабатываете чат-бот, вам будут критически важны хорошие навыки ведения диалога. Однако если ваша модель предназначена для извлечения информации из неструктурированных документов, то приоритет смещается в сторону способности точно следовать инструкциям и корректно выполнять поставленные задачи.

Хотя такие бенчмарки широко используются и приносят ощутимую пользу, они не лишены недостатков. Например, открытые тесты можно «обмануть», обучив модель на тестовых данных или на примерах, близких к заданиям из бенчмарка. Даже оценка человеком далека от идеала и часто смещена в сторону длинных и уверенных ответов, особенно если они хорошо отформатированы (например, с использованием Markdown). С другой стороны, закрытые тестовые выборки подвергались меньшей проверке со стороны сообщества и могут содержать собственные искажения или предвзятости.

Это означает, что бенчмарки не следует рассматривать как абсолютный источник истины. Скорее, они служат индикаторами: если несколько разных методов оценки дают схожие результаты, это позволяет с большей уверенностью судить о реальных возможностях модели.

Оценка предметно-ориентированных LLM

Предметно-ориентированные LLM обладают более узкой специализацией по сравнению с моделями общего назначения, поэтому для их оценки следует использовать соответствующие бенчмарки.

Выбор конкретного бенчмарка для моделей этой категории полностью зависит от предметной области. Для оценки моделей, ориентированных на конкретный

язык или генерацию кода, рекомендуется подбирать соответствующие тесты и даже целые наборы бенчмарков, разработанных с учетом воспроизводимости результатов. Благодаря охвату различных аспектов предметной области они позволяют точнее оценить производительность LLM в целевой сфере.

Ниже приведен список таких бенчмарков с таблицами лидеров, доступными на платформе Hugging Face Hub.

- *Open Medical-LLM Leaderboard* — оценивает способность LLM отвечать на вопросы в области медицины. Предусматривает девять метрик и охватывает 1273 вопроса из экзаменов на получение медицинской лицензии в США (MedQA), 500 вопросов из статей базы PubMed (PubMedQA), 4183 вопроса из индийских вступительных экзаменов по медицине (MedMCQA), а также 1089 вопросов из шести подкатегорий бенчмарка MMLU (клинические знания, медицинская генетика, анатомия, профессиональная медицина, знания по биологии и медицине на уровне колледжа).
- *BigCodeBench Leaderboard* — оценивает производительность LLM, предназначенных для работы с кодом, и включает две основные категории тестов: BigCodeBench-Complete (автодополнение на основе структурированной документации к коду) и BigCodeBench-Instruct (генерация кода на основе инструкций, написанных на естественном языке). Модели ранжируются по метрике Pass@1, рассчитанной с использованием жадного декодирования, а для категории Complete дополнительно рассчитывается рейтинг Elo. Данный бенчмарк охватывает широкий спектр сценариев написания программного кода и проверяет способности моделей к композиционному мышлению и следованию инструкциям.
- *Hallucinations Leaderboard* — оценивает склонность LLM к генерации ложной или неподтвержденной информации в рамках 16 различных задач, охватывающих пять категорий навыков: *умение отвечать на вопросы* (наборы данных NQ Open, TruthfulQA и SQuADv2), *понимание прочитанного* (TriviaQA и RACE), *резюмирование* (HaluEval Summ, XSum и CNN/DM), *ведение диалога* (HaluEval Dial и FaithDial), *проверка фактов* (MemoTrap, SelfCheckGPT, FEVER и TrueFalse). Кроме того, данный бенчмарк анализирует способность модели следовать инструкциям с помощью теста IFEval.
- *Enterprise Scenarios Leaderboard* — оценивает производительность LLM в шести корпоративных сценариях, охватывающих разнообразные задачи, характерные для бизнес-приложений. В этот набор входят следующие тесты: FinanceBench (100 вопросов с финансовым контекстом), Legal Confidentiality (100 промптов из LegalBench для оценки способности к юридической аргументации), Writing Prompts (оценка креативных способностей к написанию текстов), Customer Support Dialogue (оценка релевантности ответов в контексте взаимодействия с клиентами), Toxic Prompts (проверка устойчивости модели к генерации токсичного или вредоносного контента) и Enterprise PII (оценка способности модели обеспечивать защиту конфи-

денциальной информации в рамках бизнес-сценариев). Некоторые тестовые наборы являются закрытыми, что исключает подгонку моделей под конкретные тесты. Оценка точности ответов, способности к юридической аргументации, креативности, контекстуальной релевантности и мер безопасности обеспечивает всестороннюю проверку пригодности LLM для использования в корпоративной среде.

Подход к построению рейтингов часто зависит от предметной области. Например, BigCodeBench Leaderboard резко выделяется среди других, так как опирается всего на две метрики, которые позволяют практически полностью охватить предметную область. Напротив, Hallucinations Leaderboard объединяет 16 метрик, включая множество универсальных оценок. Это говорит о том, что использование универсальных тестов в дополнение к специализированным бенчмаркам позволяет сформировать более полноценный тестовый набор.

В частности, для оценки LLM, ориентированных на конкретный язык, нередко задействуют переведенные версии тестовых наборов. Их можно дополнять оригинальными оценочными заданиями, составленными на соответствующем языке. Хотя в ряде случаев применяются результаты машинного перевода, для повышения качества оценки лучше использовать тексты, переведенные людьми. Ниже перечислены три тематических рейтинга, служащих примером того, как можно построить собственную систему оценки.

- *OpenKo-LLM Leaderboard* — оценивает производительность LLM для корейского языка по девяти показателям. Эти метрики включают в себя как переведенные на корейский язык бенчмарки общего назначения (GPQA, Winogrande, GSM8K, EQ-Bench и IFEval), так и специализированные тесты для оценки знаний, социальной ценности, безвредности и полезности ответов модели (Knowledge, Social Value, Harmlessness, Helpfulness).
- *Open Portuguese LLM Leaderboard* — оценивает производительность LLM для португальского языка с помощью девяти разнообразных бенчмарков. Среди них: образовательные тесты (ENEM с 1430 вопросами и BLUEX с 724 вопросами из вступительных экзаменов в университет), профессиональные экзамены (OAB Exams с более чем 2000 вопросами), задания на понимание языка (ASSIN2 RTE and STS, FAQUAD NLI), а также задания на анализ контента в соцсетях (HateBR с 7000 комментариями в Instagram, PT Hate Speech с 5668 твитами и tweetSentBR).
- *Open Arabic LLM Leaderboard* — оценивает производительность LLM для арабского языка с помощью обширного набора бенчмарков, включающего как оригинальные задачи на арабском языке, так и переведенные наборы данных. Среди оригинальных бенчмарков — AlGhafa и Arabic-Culture-Value-Alignment. В набор также входят 12 переведенных бенчмарков, охватывающих различные предметные области, включая MMLU, ARC-Challenge, HellaSwag и PIQA.

Как универсальные, так и предметно-ориентированные бенчмарки разрабатываются с учетом трех ключевых принципов. Во-первых, они должны быть сложными, то есть ставить перед моделью нетривиальные задачи и побуждать ее отличать хорошие ответы от плохих. Во-вторых, они должны быть разнообразными и охватывать как можно больше тем и сценариев. Если одного бенчмарка оказывается недостаточно, можно подключить дополнительные. И наконец, бенчмарки должны быть практичными и удобными в применении. Этот аспект в большей степени связан с библиотеками для проведения оценки, которые могут различаться по уровню сложности. Мы рекомендуем использовать lm-evaluation-harness от EleutherAI (https://github.com/EleutherAI/lm-evaluation-harness) и lighteval от Hugging Face (https://github.com/huggingface/lighteval).

Оценка узкоспециализированных LLM

Хотя универсальные и предметно-ориентированные бенчмарки позволяют судить о базовом уровне и качестве обученных моделей, они не дают представления о том, насколько хорошо эти модели справляются с прикладными задачами. Для этого требуются специальные тестовые наборы.

Поскольку узкоспециализированные модели имеют узкую специализацию, для их оценки редко используются готовые оценочные наборы данных. Однако их выходные данные, как правило, являются более структурированными, поэтому их легче проанализировать с помощью классических метрик машинного обучения. Например, для оценки эффективности решения задачи резюмирования может применяться метрика *ROUGE* (Recall-Oriented Understudy for Gisting Evaluation), измеряющая степень совпадения между сгенерированным и эталонным текстом с помощью *n*-грамм.

Эффективность решения задачи классификации тоже можно легко оценить с помощью следующих традиционных метрик.

- *Точность (accuracy)* — доля правильно предсказанных примеров от их общего числа. Эта метрика особенно полезна в случае задач с четкой разницей между правильными и неправильными ответами, например при *распознавании именованных сущностей* (Named Entity Recognition, NER).
- *Точность предсказаний (precision)* — отношение числа истинно положительных предсказаний к общему числу положительных предсказаний модели.
- *Полнота (recall)* — отношение числа истинно положительных предсказаний к общему числу положительных примеров.
- *F1-мера (F1 score)* — гармоническое среднее между точностью предсказаний и полнотой, позволяющее сбалансировать обе метрики. Особенно полезна в задачах классификации и извлечения сущностей.

Если задача не сводится к классической задаче машинного обучения, можно создать собственный бенчмарк, похожий на наборы тестов для оценки универсальных или предметно-ориентированных моделей. Одним из распространенных и успешных подходов является использование вопросов с выбором ответа. В этом случае инструкция представляет собой вопрос с несколькими вариантами. Рассмотрим пример такого вопроса из набора MMLU (посвященного общей алгебре).

Пример из набора данных MMLU[1]

Инструкция

Найди степень расширения поля Q(sqrt(2), sqrt(3)) над Q.

A. 0

Б. 4

В. 2

Г. 6

Ответ

Б

В рамках этого подхода существует два основных способа оценки моделей — на основе генерации текста и через анализ вероятностей.

В первом случае модель генерирует текстовый ответ, который затем сравнивается с заданными вариантами. Например, модель выдает в качестве ответа одну из букв (А, Б, В или Г), которая проверяется на соответствие правильному варианту. Этот метод позволяет оценить способность модели формулировать связные и точные ответы в формате, приближенном к реальным условиям использования.

Оценка на основе вероятностей, в свою очередь, предполагает анализ предсказанных моделью вероятностей для различных вариантов ответа без необходимости генерировать текст. В случае с MMLU библиотека lm-evaluation-harness сравнивает вероятности, присвоенные полным формулировкам каждого из вариантов ответа. Этот подход позволяет получить более детальную оценку понимания модели, поскольку отражает ее относительную уверенность в разных вариантах, даже если сама модель не сгенерировала правильный ответ.

Если вы стремитесь к простоте, мы рекомендуем использовать способ оценки на основе генерации текста, имитирующий поведение человека при прохождении тестов. Он проще в реализации и, как правило, позволяет эффективнее отличать

[1] Здесь и далее примеры из англоязычных наборов данных приводятся в переводе. — *Примеч. пер.*

хорошие модели от плохих, так как низкокачественные модели нередко показывают завышенные результаты при проведении оценки на основе вероятностей. Эту технику можно адаптировать для тестирования моделей на конкретной задаче и даже расширить его на новые предметные области.

Однако в случае с открытыми задачами классические метрики машинного обучения и способ оценки путем выбора из нескольких вариантов ответов могут оказаться неприменимыми. В таких случаях можно привлечь LLM в качестве эксперта, когда оценку качества ответов осуществляет сама модель, о чем мы говорили в главе 5. Если у вас есть эталонные ответы, их включение в контекст может повысить точность оценки. В противном случае имеет смысл заранее определить различные критерии (например, релевантность или токсичность), чтобы сделать результаты более интерпретируемыми.

Для проведения оценки рекомендуется использовать крупные модели и многократно уточнять промпт. При этом важную роль играют объяснения, которые генерирует модель: они помогают выявить ошибки в ее рассуждениях и скорректировать их с помощью дополнительной настройки промпта.

Чтобы упростить разбор ответов, можно задать структуру прямо в промпте или использовать форматирование в запросе (вроде Outlines или режима JSON от OpenAI). Далее приведен пример инструкции с заданной структурой.

Пример универсального промпта для LLM-эксперта, оценивающего качество ответа

```
Ты — оценщик, который анализирует качество ответов.
Твоя задача — выставить оценку, отражающую степень соответствия ответа заданной
инструкции.
Ты будешь использовать шкалу от 1 до 4, где каждый балл означает следующее:
Ответ не соответствует инструкции.
Ответ соответствует инструкции, но бесполезен.
Ответ соответствует инструкции и полезен, но мог бы быть более подробным.
Ответ соответствует инструкции, полезен и достаточно подробен.
Пожалуйста, оформи свою оценку в следующем формате:
##Оценка##
Обоснование: (проанализируй релевантность, полезность и степень детализации
ответа)
Итоговая оценка: (целое число от 1 до 4)
```

Инструкция:

{инструкция}

Ответ:

{ответ}

##Оценка##

Обоснование:

Разумеется, вы можете изменить эту шкалу, добавить эталонный ответ к промпту и адаптировать его под собственные задачи.

Однако LLM-эксперты могут быть предвзяты, например отдавать предпочтение менее точным, но убедительно звучащим или многословным ответам. Недостаток знаний в специализированных областях приводит к ошибкам в суждениях. Кроме того, одна и та же модель по-разному оценивает схожие ответы или предпочитает определенные стилевые особенности текста, никак не связанные с качеством оцениваемого ответа. Чтобы снизить влияние этих факторов, можно комбинировать оценки LLM с другими метриками, использовать несколько LLM-экспертов и тщательно составлять промпты с учетом потенциальных искажений.

Модели, прошедшие проверку и демонстрирующие корректную работу, можно встроить в более обширные системы, об оценке которых мы поговорим в следующем разделе.

Оценка RAG-систем

Если традиционные способы оценки LLM направлены на проверку способностей модели, то оценка RAG-систем требует более комплексного подхода, учитывающего как генеративные возможности модели, так и ее взаимодействие с внешними источниками.

RAG-системы объединяют сильные стороны LLM с механизмами поиска информации, что позволяет им генерировать ответы, которые не только являются связными и контекстуально уместными, но и опираются на актуальные внешние данные. Это делает RAG-системы особенно ценными в областях, где критически важна точность и своевременность получения сведений, таких как новостная журналистика, научные исследования и обслуживание клиентов.

Оценка RAG-систем не ограничивается проверкой изолированной LLM, а требует анализа всех аспектов ее работы и включает в себя следующее.

- *Точность поиска (retrieval accuracy)*. Насколько хорошо система извлекает релевантную информацию?
- *Качество интеграции (integration quality)*. Насколько эффективно извлеченная информация встраивается в сгенерированный ответ?
- *Соответствие фактам и релевантность (factuality and relevance)*. Отвечает ли выходной текст на запрос, гармонично объединяя извлеченный и сгенерированный контент?

Ключевыми метриками при оценке RAG-систем служат точность и полнота извлеченной информации. Кроме того, критически важными являются качество

интеграции извлеченных данных в сгенерированный текст, а также общее соответствие фактам и связность ответа.

Чтобы проиллюстрировать процесс применения этих метрик на практике, рассмотрим RAG-систему, разработанную для чат-бота службы поддержки клиентов в сфере электронной коммерции. В этом сценарии пользователь задает вопрос: «Какая политика возврата действует в отношении ноутбуков, купленных во время праздничной распродажи?» Пайплайн RAG находит соответствующие документы, содержащие политику возврата электроники и условия праздничной распродажи. Эти дополнительные материалы добавляются к вопросу и используются моделью при формулировании ответа.

Пример ответа, сгенерированного RAG-системой, предназначенной для службы поддержки клиентов

```
Для ноутбуков, приобретенных во время нашей праздничной распродажи, действует
продленный срок возврата — 60 дней с даты покупки. Это дольше, чем стандартный
срок возврата для электроники, составляющий 30 дней. Чтобы вернуть средства
в полном объеме, пожалуйста, убедитесь, что ноутбук возвращается в оригинальной
упаковке и с полным комплектом аксессуаров.
```

В рамках этого пайплайна мы можем оценить, соответствуют ли ожиданиям извлеченные документы (точность поиска). Мы также способны измерить разницу между ответами, сгенерированными при наличии дополнительного контекста и без него (качество интеграции). И наконец, мы можем проверить, соответствует ли итоговый ответ исходному запросу и основывается ли он на извлеченных данных (соответствие фактам и релевантность).

В этом разделе будут представлены два метода, позволяющие оценить, насколько эффективно модели RAG интегрируют внешнюю информацию в свои ответы.

Инструментарий Ragas

Ragas (Retrieval-Augmented Generation Assessment) — это инструментарий с открытым исходным кодом, предназначенный для всесторонней оценки и оптимизации RAG-систем. В его основе лежит методология разработки *MDD* (metrics-driven development), суть которой сводится к использованию данных для принятия обоснованных решений и постоянному мониторингу ключевых метрик с целью получения ценной информации о работе приложения. Таким образом, Ragas позволяет разработчикам объективно оценивать свои RAG-системы, выявлять области для улучшения и отслеживать влияние изменений.

Одной из ключевых возможностей Ragas является синтетическая генерация разнообразных и сложных тестовых наборов данных. Это особенно важно, поскольку ручное составление сотен вопросов, ответов и контекстов требует больших усилий и времени. Вместо этого Ragas применяет эволюционный подход, вдохновленный такими методами, как Evol-Instruct, для создания вопросов,

различающихся степенью сложности рассуждений, наличием условных элементов и требований к анализу множества контекстов. Такой подход обеспечивает комплексную оценку всех компонентов пайплайна RAG.

Кроме того, Ragas может генерировать примеры диалогов, моделирующие взаимодействие в чате, что позволяет разработчикам тестировать свои системы в рамках более реалистичных сценариев.

Как показано на рис. 7.1, инструментарий Ragas предоставляет набор метрик, позволяющих объективно измерить различные аспекты производительности RAG-систем. К ним относятся следующие метрики.

- *Достоверность (faithfulness)* — показывает фактическую точность сгенерированного ответа относительно предоставленного контекста. Для этого ответ разбивается на отдельные утверждения и проверяется, можно ли подтвердить каждое из них на основе контекста. Достоверность вычисляется как отношение количества утверждений, которые подтверждаются контекстом, к общему числу утверждений в ответе.
- *Релевантность ответа (answer relevancy)* — показывает соответствие сгенерированного ответа исходному вопросу. Для этого используется инновационный подход, в рамках которого LLM генерирует несколько вопросов на основе ответа, а затем вычисляется среднее косинусное сходство между этими вопросами и исходным вопросом. Такой прием позволяет выявить ответы, которые фактически верны, но неполны или не относятся к теме.
- *Точность контекста (context precision)* — показывает корректность ранжирования всех релевантных элементов, присутствующих в извлеченном контексте. При этом учитывается положение значимых фрагментов информации: системы, которые располагают наиболее важные сведения в верхней части контекста, получают более высокую оценку.
- *Полнота контекста (context recall)* — показывает, насколько полно извлеченный контекст соответствует эталонному ответу. Для этого анализируется каждое утверждение из эталонного ответа и проверяется возможность его обоснования на базе извлеченного контекста. Это дает представление о полноте извлеченной информации.

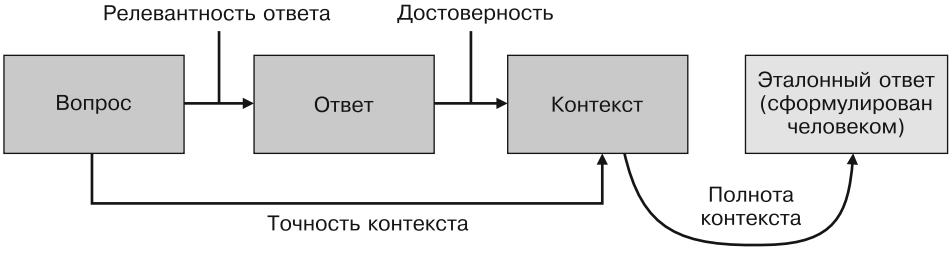


Рис. 7.1. Обзор системы оценки Ragas

В заключение следует отметить, что Ragas также предоставляет инструменты для мониторинга качества RAG в продакшене, что позволяет непрерывно улучшать эти системы, например, за счет настройки алгоритмов поиска, корректировки стратегии составления промптов или оптимизации соотношения между объемом извлекаемого контекста и долей контента, генерируемого LLM.

Кроме того, применение Ragas можно сочетать с подходом на основе пользовательских классификаторов.

Фреймворк ARES

ARES (Automated RAG Evaluation System) — это комплексный инструмент, предназначенный для оценки RAG-систем. Он реализует автоматизированный процесс, который сочетает синтетическую генерацию данных с использованием тонко настроенных классификаторов с целью анализа различных аспектов производительности RAG-систем, включая релевантность контекста, достоверность ответа и соответствие ответа исходному запросу (рис. 7.2).

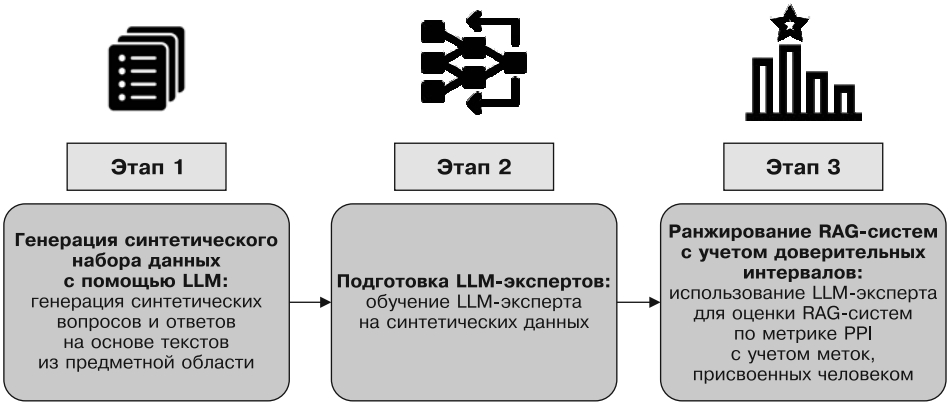


Рис. 7.2. Обзор рабочего процесса фреймворка ARES

Процесс работы фреймворка ARES предусматривает три этапа: генерацию синтетических данных, обучение классификаторов и оценку RAG-системы. Каждый этап допускает настройку, что позволяет пользователям адаптировать процесс оценки под свои задачи и наборы данных.

На этапе генерации синтетических данных ARES создает наборы данных, имитирующие реальные сценарии применения, что обеспечивает более надежное тестирование RAG-систем. Пользователь может управлять этим процессом, указав пути к файлам с документами, few-shot-промптам и папке для сохранения синтетических запросов. Для выполнения этой задачи фреймворк позволяет использовать различные предварительно обученные языковые модели

(по умолчанию установлена google/flan-t5-xxl). Кроме того, можно контролировать количество выбираемых документов и другие параметры, что дает возможность найти баланс между полнотой охвата и вычислительной эффективностью.

На этапе обучения классификаторов создаются высокоточные модели, предназначенные для определения релевантности и достоверности результатов работы RAG-системы. Пользователь может указать набор данных для классификации (как правило, полученный на предыдущем этапе), тестовую выборку, названия столбцов с метками и выбранную модель. По умолчанию в ARES применяется модель microsoft/deberta-v3-large, однако платформа поддерживает и другие модели из библиотеки Hugging Face. Параметры обучения, такие как количество эпох и скорость обучения, можно настраивать для повышения качества работы классификаторов.

На заключительном этапе ARES использует обученные классификаторы и синтетические данные для анализа производительности RAG-модели. Пользователь должен предоставить эталонные наборы данных, few-shot-примеры для управления логикой оценки, контрольные точки обученных классификаторов и пути к эталонным меткам (gold labels).

ARES поддерживает разнообразные метрики оценки и способен рассчитывать доверительные интервалы[1] для своих результатов.

Данный фреймворк предоставляет гибкие возможности запуска моделей, поддерживая как облачное, так и локальное выполнение с помощью интеграции vLLM. Кроме того, ARES умеет работать с различными типами артефактов, включая фрагменты кода, документы, HTML-страницы, изображения и т. д., что делает его пригодным для оценки различных выходных данных RAG-системы.

В целом Ragas и ARES дополняют друг друга благодаря различным подходам к оценке и генерации наборов данных. Сильной стороной Ragas является мониторинг в продакшене и использование метрик, рассчитываемых с помощью LLM, а ARES реализует гибко настраиваемый процесс оценки на основе классификаторов. Ragas позволяет проводить более детализированный анализ, опираясь на возможности больших языковых моделей, а ARES обеспечивает стабильную и потенциально более быструю оценку после обучения классификаторов. Совместное использование этих инструментов дает возможность построить комплексную систему оценки, сочетающую быструю итеративную проверку с помощью Ragas и глубокую, гибко настраиваемую оценку с помощью ARES, проводимую на ключевых этапах разработки.

В следующем разделе мы создадим собственную систему для оценки нашей узкоспециализированной модели TwinLlama-3.1-8B.

---

[1] Доверительный интервал — это диапазон, в который попадает точность нашей модели, с учетом колебаний. — *Примеч. науч ред.*

Оценка модели TwinLlama-3.1-8B

В предыдущих главах мы тонко настроили две модели для генерации качественных постов и статей: TwinLlama-3.1-8B и TwinLlama-3.1-8B-DPO. Теперь нам нужно оценить их способность к написанию точных и качественных текстов. Универсальные модели, прошедшие тонкую настройку, демонстрируют высокую точность благодаря обширным знаниям, но при этом часто используют излишне формальный язык и отличаются многословием. Наша цель заключалась в том, чтобы сделать стиль модели более естественным и приближенным к оригинальным статьям из обучающего набора.

Поскольку эта задача имеет открытый характер, для оценки качества сгенерированного текста мы задействуем LLM-эксперта, который получает на вход инструкцию и ответ, а затем выставляет оценку от 1 до 3 по двум критериям:

- *точность* — степень фактической корректности и полноты представленной в ответе информации;
- *стиль* — уместность тона и стиля изложения для блогов или контента в социальных сетях (отсутствие формальных и академических выражений).

Для оценки воспользуемся тестовым подмножеством нашего набора данных с инструкциями. Тестовые инструкции будут подаваться на вход моделям, а те, в свою очередь, сгенерируют ответы на их основе. Затем мы передадим эти ответы LLM-эксперту (GPT-4o-mini) для оценки по заданным в промпте критериям. Наконец, мы проанализируем качественные и количественные оценки, чтобы сделать окончательные выводы.

Генерация ответов

Первым шагом является генерация ответов на каждую инструкцию из тестового набора. В дополнение к нашим двум моделям мы задействуем meta-llama/Meta-Llama-3.1-8B-Instruct — официальную версию Llama-3.1-8B в качестве контрольной точки, которая позволит нам понять, на какие компромиссы нам пришлось пойти.

Приступаем к реализации первого этапа.

1. Сначала мы импортируем необходимые библиотеки, включая vLLM, которая обеспечивает более высокую скорость генерации в пакетном режиме по сравнению с `Transformers`:

```
from vllm import LLM, SamplingParams
from datasets import load_dataset
from tqdm.auto import tqdm
import gc
```

2. Затем мы определяем функцию `generate_answers`, отвечающую за генерацию ответов на основе выбранной модели и тестового набора. Она принимает два аргумента — идентификатор модели и название набора данных:

```
def generate_answers(model_id, dataset_name):
    dataset = load_dataset(dataset_name, split="test")
```

3. Сырые инструкции необходимо отформатировать, задействовав шаблон ведения диалога, на котором обучались наши модели. Несмотря на то что Llama-3.1-8B-Instruct использовала иной шаблон, она способна следовать и этому упрощенному формату. Для обеспечения единообразия все модели здесь используют один и тот же шаблон. Мы сопоставляем весь тестовый набор с этим шаблоном с помощью функции `format()`:

```
    def format(sample):
        return "Ниже приведена инструкция, описывающая задачу. Напиши ответ,
➥который соответствующим образом завершает запрос.\n\n###
➥Инструкция:\n{}\n\n### Ответ:\n".format(sample["instruction"])
    dataset = dataset.map(lambda sample: {"prompt": format(sample)})
```

4. Далее мы инициализируем объект LLM из библиотеки vLLM с максимальной длиной 4096 токенов. Мы также можем задать гиперпараметры, которые соответствуют переменным в стратегии декодирования. В данном случае используются настройки, обеспечивающие разнообразие ответов (высокое значение `temperature`) и исключающие наименее вероятные токены (`top_p` и `min_p`). Для запуска процесса генерации мы передаем список промптов с помощью `dataset["prompt"]`:

```
    llm = LLM(model=model_id, max_model_len=4096)
    sampling_params = SamplingParams(temperature=0.8, top_p=0.95,
➥min_p=0.05, max_tokens=4096)
    outputs = llm.generate(dataset["prompt"], sampling_params)
```

5. Процесс генерации ответов на 334 инструкции занимает несколько минут. После его завершения мы извлекаем ответы из объекта, возвращенного библиотекой vLLM, и добавляем их в наш набор данных в виде нового столбца. Это позволяет сохранить ответы для их последующего просмотра и анализа:

```
    answers = [output.outputs[0].text for output in outputs]
    dataset = dataset.add_column("answers", answers)
```

6. Полученные результаты загружаются на платформу Hugging Face Hub для последующего использования. После этого мы очищаем память GPU для корректного запуска следующей модели:

```
    print(f"Uploading results for {model_id}")
    dataset.push_to_hub(f"mlabonne/{model_id.split('/') [-1]}-results")
    gc.collect()

    return dataset
```

7. Далее мы составляем список из трех моделей, которые хотим протестировать. Затем поочередно запускаем функцию `generate_answers()` для каждой из них. В результате будет создан отдельный набор с ответами каждой модели и загружен на платформу Hugging Face:

```
model_ids = [
    'mlabonne/TwinLlama-3.1-8B',
    'mlabonne/TwinLlama-3.1-8B-DPO',
    'meta-llama/Meta-Llama-3.1-8B-Instruct'
]
for model_id in model_ids:
    generate_answers(model_id, "mlabonne/llmtwin")
```

Теперь, когда генерация ответов завершена, можно переходить к этапу их оценки.

Оценка ответов

Для оценки сгенерированных ответов мы используем модель GPT-4o-mini, которая выступит в роли эксперта. Эта стратегия аналогична той, что применялась при генерации данных. Фактически ее можно адаптировать и для отсеивания некачественных примеров на этапе генерации. Здесь мы будем оценивать ответы каждой модели по двум критериям: точность и стиль. Средние оценки позволят нам судить о качестве тонко настроенных моделей по сравнению с Llama-3.1-8B-Instruct.

1. Сначала мы импортируем необходимые библиотеки, включая `openai`:

```
import json
from typing import List
from datasets import Dataset, load_dataset
from openai import OpenAI
from tqdm.auto import tqdm
import concurrent.futures
```

2. Далее определяем функцию `evaluate_answer()`. В ней задается промпт, который описывает контекст и критерии оценки ответов, такие как точность и стиль:

```
def evaluate_answer(
    instruction: str, answer: str, client: OpenAI
) -> dict:
    prompt = f"""Ты — эксперт. Пожалуйста, оцени качество выданного
ответа на инструкцию на основе двух критериев:
1. Точность: насколько достоверна информация, представленная в ответе?
Ты — технический эксперт в этой теме.
2. Стиль: подходит ли тон и стиль написания для публикации в блоге или
для контента в социальных сетях? Следует использовать простые, но
технические слова и избегать формального или академического языка.
```

3. В этом же промпте задаются шкалы оценки. Мы применим трехбалльные шкалы Лайкерта с четкими описаниями каждого балла:

```
Шкала точности:
1. (Неудовлетворительно): Содержит фактические ошибки или вводящую
в заблуждение информацию.
2. (Хорошо): В основном точная, с незначительными ошибками или пропусками.
3. (Отлично): Высокая точность и полнота изложения.

Шкала стиля:
1. (Неудовлетворительно): Слишком формальный стиль, используется несколько
чрезмерно сложных слов.
2. (Хорошо): Хороший баланс технического содержания и доступности, но при
этом все еще используются официальные слова и выражения.
3. (Отлично): Идеально подходящий язык для блога/социальных сетей, при
необходимости используются простые, но точные технические термины.
```

4. В завершение промпта приводятся два примера, иллюстрирующих то, что подразумевается под «сложной лексикой» (complex words) и «формальным или академическим стилем» (formal or academic language). Мы предоставляем пары «инструкция — ответ» и просим модель вернуть ответ в формате JSON:

```
Пример плохого стиля: Модель Llama2 7B представляет собой заметный
прогресс в области искусственного интеллекта, являясь преемником своей
предшественницы, оригинальной архитектуры Llama.
Пример отличного стиля: Llama2 7B превосходит оригинальную модель Llama по
многим показателям.

Инструкция: {instruction}

Ответ: {answer}

Предоставь свою оценку в формате JSON со следующей структурой:
{{
    "accuracy": {{
        "analysis": "...",
        "score": 0
    }},
    "style": {{
        "analysis": "...",
        "score": 0
    }}
}}
"""
```

5. Этот промпт передается модели GPT-4o-mini в качестве пользовательского запроса. Системный промпт уточняет, что нас интересует оценка точности и стиля ответа:

```
completion = client.chat.completions.create(
    model="gpt-4o-mini",
```

```
    messages=[
        {
            "role": "system",
            "content": "Ты — полезный помощник, который оценивает
             ответы на основе точности и стиля. Предоставь свой ответ
             в формате JSON с кратким анализом и оценкой по каждому критерию.",
        },
        {"role": "user", "content": prompt},
    ],
    response_format={"type": "json_object"},
    max_tokens=1000,
    temperature=0.8,
)
```

6. Как и в предыдущих главах, мы реализуем пакетную обработку запросов для ускорения процесса. Для этого создается функция `evaluate_batch()`, которая возвращает список структурированных результатов с соответствующими индексами. Эти индексы необходимы для корректного упорядочения оценок:

```
def evaluate_batch(batch, start_index):
    client = OpenAI(api_key=OPENAI_KEY)
    return [
        (i, evaluate_answer(instr, ans, client))
        for i, (instr, ans) in enumerate(batch, start=start_index)
    ]
```

7. Затем мы определяем функцию `evaluate_answers()` для оркестрации описанного выше процесса. Она принимает идентификатор модели, количество потоков и размер пакета. Первым делом мы загружаем набор данных с ранее сгенерированными ответами:

```
def evaluate_answers(model_id: str, num_threads: int = 10,
➥batch_size: int = 5) -> Dataset:
    dataset = load_dataset(f"mlabonne/{model_id.split('/')
    ➥[-1]}-results", split="all")
```

8. Далее мы формируем пакеты из пар «инструкция — ответ», каждый из которых содержит количество записей, равное значению параметра `batch_size`:

```
    batches = [
        (i, list(zip(dataset["instruction"][i:i+batch_size],
        ➥dataset["answers"][i:i+batch_size])))
        for i in range(0, len(dataset), batch_size)
    ]
```

9. Оценка пакетов производится в параллельном режиме с использованием нескольких потоков, что позволяет ускорить этот процесс. Каждый пакет отправляется в функцию `evaluate_batch()` с помощью `ThreadPoolExecutor`. Результаты сохраняются в списке `evaluations`:

```
    evaluations = [None] * len(dataset)

    with concurrent.futures.ThreadPoolExecutor(max_workers=num_threads)
   ➥as executor:
```

```
    futures = [executor.submit(evaluate_batch, batch, start_index)
    ➥for start_index, batch in batches]

    for future in tqdm(concurrent.futures.as_completed(futures),
    ➥total=len(futures)):
        for index, evaluation in future.result():
            evaluations[index] = evaluation
```

10. Далее мы создаем в наборе данных новый столбец для сохранения сырых результатов работы модели-эксперта в формате JSON, включая оценки и пояснения:

```
if 'evaluation' in dataset.column_names:
    dataset = dataset.remove_columns(['evaluation'])
dataset = dataset.add_column("evaluation", evaluations)
```

11. Мы можем разобрать этот JSON-объект с помощью `json.loads()` и извлечь сгенерированные оценки за точность и стиль. Генерация выполняется в режиме *best-effort* («в меру возможностей»), то есть получение результатов не гарантируется. Если при разборе объекта возникает ошибка, мы используем значение `None` в качестве запасного варианта:

```
accuracy_scores = []
style_scores = []

for evaluation in dataset['evaluation']:
    try:
        eval_dict = json.loads(evaluation) if
        ➥isinstance(evaluation, str) else evaluation
        accuracy_score = eval_dict['accuracy']['score']
        style_score = eval_dict['style']['score']
        accuracy_scores.append(accuracy_score)
        style_scores.append(style_score)
    except (json.JSONDecodeError, KeyError, TypeError):
        accuracy_scores.append(None)
        style_scores.append(None)
```

12. Далее добавляем два новых столбца для сохранения оценок точности и стиля для их последующего анализа:

```
if 'accuracy' in dataset.column_names:
    dataset = dataset.remove_columns(['accuracy'])
dataset = dataset.add_column('accuracy', accuracy_scores)
if 'style' in dataset.column_names:
    dataset = dataset.remove_columns(['style'])
dataset = dataset.add_column('style', style_scores)
```

13. Затем итоговый набор данных, содержащий ответы, оценки и выставленные баллы, загружается на платформу Hugging Face Hub:

```
dataset.push_to_hub(f"mlabonne/{model_id.split('/') [-1]}-results")
return dataset
```

14. Наконец, вызываем функцию `evaluate_answers()` для трех выбранных моделей:

```
model_ids = [
    'mlabonne/TwinLlama-3.1-8B',
    'mlabonne/TwinLlama-3.1-8B-DPO',
    'meta-llama/Meta-Llama-3.1-8B-Instruct'
]
for model_id in model_ids:
    evaluate_answers(model_id)
```

Благодаря сохранению промежуточных результатов наш фреймворк оказывается более устойчивым к сбоям и легко адаптируемым для других моделей, наборов данных и критериев оценки.

Анализ результатов

По завершении процесса оценки мы можем проанализировать три ключевых компонента: сами ответы моделей, объяснения LLM-эксперта и выставленные оценки.

Этап изучения ответов не масштабируется, но играет важную роль, так как позволяет выявить типичные ошибки, например связанные с использованием неправильного шаблона ведения диалога или выбором неправильной модели. В этом разделе мы сосредоточим внимание на одной инструкции и проанализируем ответы, сгенерированные двумя нашими моделями и моделью Llama-3.1-8B-Instruct (табл. 7.1).

Таблица 7.1. Примеры сгенерированных ответов на заданную инструкцию

Инструкция
Объясни концепцию алгоритмической предвзятости и опиши ее последствия
TwinLlama-31-8B
Алгоритмическая предвзятость — это склонность алгоритмов выдавать искаженные или несправедливые результаты из-за скрытых предпосылок или предвзятости, заложенной в программный код. Это может иметь серьезные последствия, особенно в таких сферах, как машинное обучение и искусственный интеллект, где на алгоритмы возлагается ответственность за принятие критически важных решений. Алгоритм, не прошедший должной проверки на наличие предвзятости, может склоняться к дискриминационным решениям, например, при выдаче кредита человеку из определенной демографической группы или к несправедливым процедурам найма, отдающим предпочтение определенным кандидатам. Поэтому при проектировании и внедрении алгоритмов крайне важно обеспечить их справедливость и точность, чтобы снизить риски подобных искажений

TwinLlama-31-8B-DPO

Алгоритмическая предвзятость — это склонность алгоритмов выдавать искаженные или несправедливые результаты из-за лежащих в их основе допущений или предвзятости, заложенной в программный код. Это может иметь серьезные последствия, особенно в таких сферах, как машинное обучение и искусственный интеллект, где на алгоритмы возлагается ответственность за принятие критически важных решений. Алгоритм, не прошедший должной проверки на наличие предвзятости, может склоняться к дискриминационным решениям, например отказывать в выдаче кредитов представителям определенных демографических групп или закреплять существующие стереотипы. Поэтому крайне важно выявлять и устранять алгоритмическую предвзятость на ранних этапах, чтобы создаваемые нами технологии были справедливыми и честными

Llama-31-8B-Instruct

Алгоритмическая предвзятость — это непреднамеренно привнесенное или изначально присутствующее искажение в моделях машинного обучения, способное повлиять на их производительность, точность и справедливость. Источниками этой предвзятости могут быть несбалансированные обучающие данные, использованные для обучения модели, особенности самого алгоритма, а также распространенные в обществе или культуре предубеждения. Одним из наиболее серьезных последствий алгоритмической предвзятости является ее способность воспроизводить и усиливать уже существующие социальные неравенства. Так, например, программное обеспечение для распознавания лиц менее точно работает с изображениями темнокожих людей, что приводит к ошибочным задержаниям и дискриминации. Аналогичным образом модели обработки естественного языка нередко демонстрируют предвзятость в пользу языка и диалекта доминирующей группы, что затрудняет доступ к информации и услугам представителям меньшинств. Алгоритмическая предвзятость может иметь особенно серьезные последствия в таких областях, как здравоохранение, образование и трудоустройство. Так, алгоритм для медицинской диагностики, обученный на данных, смещенных для определенной демографической группы, может неверно диагностировать или вовсе не распознавать заболевания у представителей других групп, что может привести к задержке лечения или его неэффективности. В сфере образования алгоритмы, ориентированные на конкретные стили обучения, могут несправедливо оценивать учеников с иными когнитивными способностями, создавая неравные условия обучения. Для смягчения последствий алгоритмической предвзятости необходимо выявлять ее коренные причины и внедрять меры, обеспечивающие справедливость и прозрачность моделей машинного обучения. Для этого можно применять такие методы, как очистка данных, регуляризация и устранение искажений, а также использовать разнообразные и репрезентативные наборы данных. Кроме того, крайне важно привлекать к разработке и тестированию алгоритмов широкий круг заинтересованных сторон, включая представителей дискриминируемых сообществ, чтобы гарантировать справедливость и инклюзивность разрабатываемых решений. В заключение следует отметить, что алгоритмическая предвзятость представляет собой серьезную проблему, способную оказывать далекоидущие последствия как на отдельных людей, так и на общество в целом. Осознание потенциальных искажений в моделях машинного обучения и принятие упреждающих мер по их устранению является ключевым условием создания более справедливых систем, приносящих пользу всем людям вне зависимости от их происхождения и принадлежности к определенным демографическим группам

Сразу можно заметить, что ответы наших моделей очень похожи. В свою очередь, ответ модели Llama-3.1-8B-Instruct отличается многословием и тем, что содержит множество примеров. Это вариант корректного, но чрезмерно длинного ответа, который может показаться искусственным и утомительным при чтении. В то же время модель, тонко настроенная методом DPO, немного упростила стиль изложения по сравнению с SFT-версией, сделав его менее академичным. Изменение стиля изложения при сохранении фактической точности ответа является как раз тем поведением, которого мы и добиваемся.

Теперь давайте проанализируем оценки, выставленные моделью GPT-4o-mini каждому из ответов (табл. 7.2).

Таблица 7.2. Оценки ответов, выставленные моделью GPT-4o-mini по таким критериям, как стиль и точность

TwinLlama-3.1-8B	
Точность Ответ содержит точное определение алгоритмической предвзятости и подчеркивает ее значение в таких областях, как машинное обучение и искусственный интеллект. В нем корректно указаны риски дискриминационных решений, принимаемых предвзятыми алгоритмами. Фактических ошибок не обнаружено, что делает данный ответ исчерпывающим и достоверным. Оценка: 3	**Стиль** Текст является информативным и в целом доступным для понимания, хотя в отдельных фрагментах наблюдается склонность к излишней формальности (например, в выражении о «скрытых предпосылках или предвзятости, заложенной в программный код»). Технические термины используются уместно, однако текст можно упростить для охвата более широкой аудитории, особенно если он предназначен для публикации в блоге или социальных сетях. Оценка: 2
TwinLlama-3.1-8B-DPO	
Точность Ответ точно определяет понятие алгоритмической предвзятости и описывает ее последствия в таких ключевых сферах, как машинное обучение и искусственный интеллект. Упоминание потенциально дискриминационных результатов делает изложение темы исчерпывающим и свободным от фактических ошибок. Оценка: 3	**Стиль** Текст ясен, доступен для понимания и эффективно передает суть понятия алгоритмической предвзятости без излишней формальности. Хотя в тексте присутствуют некоторые технические термины, они уместны и объяснены доступным для широкой аудитории языком. Оценка: 3

Llama-3.1-8B-Instruct	
Точность Ответ содержит точное определение алгоритмической предвзятости, а также раскрывает ее причины и последствия в таких сферах, как распознавание лиц, здравоохранение и образование. Корректно обозначены возможные способы борьбы с предвзятостью, включая привлечение различных заинтересованных сторон и использование методов устранения смещений. В целом представленная информация является достоверной и всесторонне раскрывает заданную тему. Оценка: 3	**Стиль** Текст является информативным и доступным, технические термины объяснены понятным языком. Однако некоторые выражения, такие как «непреднамеренно привнесенное или изначально присутствующее искажение» и «использование разнообразных и репрезентативных наборов данных», можно упростить для охвата более широкого круга читателей. В целом текст подходит для публикации в блоге или социальных сетях, но его стиль можно было бы сделать более разговорным. Оценка: 2

Модель-эксперт не выявила проблем с содержанием ответов, поэтому выставила всем им наивысший балл за точность. Однако стиль изложения моделей TwinLlama-3.1-8B (SFT) и Llama-3.1-8B-Instruct был признан чрезмерно формальным и оценен на 2 балла. В свою очередь, стиль модели TwinLlama-3.1-8B-DPO был оценен на «отлично». Согласно LLM-эксперту, ее ответ «передает суть понятия алгоритмической предвзятости без излишней формальности».

Эти оценки вполне согласуются со средними баллами по всем данным, полученными каждой из моделей:

```
TwinLlama-3.1-8B - Accuracy: 2.45
TwinLlama-3.1-8B - Style: 2.04

TwinLlama-3.1-8B-DPO - Accuracy: 2.46
TwinLlama-3.1-8B-DPO - Style: 2.12

Llama-3.1-8B-Instruct - Accuracy: 2.62
Llama-3.1-8B-Instruct - Style: 1.86
```

В плане точности две наши тонко настроенные модели демонстрируют схожие результаты, в то время как наивысший средний балл, 2,62, присваивается модели Llama-3.1-8B-Instruct. Это указывает на небольшое превосходство данной модели в плане передачи достоверной информации. Вероятнее всего, это связано с тем, что на этапе постобучения она была обучена более чем на 10 млн примеров, а наши модели — всего лишь на 13 000.

Однако при оценке стиля наблюдается совсем иная картина. Здесь лидирует модель TwinLlama-3.1-8B-DPO с оценкой 2,12, которой удалось добиться более

доступного и менее формального изложения без потери качества содержания. За ней следует модель TwinLlama-3.1-8B (SFT) с оценкой 2,04, чей стиль признан менее академичным, но не лишенным некоторой формальности. Модель Llama-3.1-8B-Instruct замыкает рейтинг с результатом 1,86 из-за склонности к многословию.

Опираясь на эти оценки и анализ сгенерированных ответов, мы можем выявить ошибки и определить области для улучшения. Это особенно важно для совершенствования процесса генерации данных путем дополнительной фильтрации или пополнения набора данных недостающей информацией. Несмотря на то что первая версия модели уже демонстрирует обнадеживающие результаты, дополнительные итерации с использованием других наборов данных и моделей позволят нам значительно превзойти исходный уровень и приблизиться к созданию оптимальной модели для решения поставленной задачи.

Резюме

В этой главе мы рассмотрели методы оценки различных LLM и RAG-систем. Вы познакомились с такими классическими бенчмарками, как MMLU, позволяющими выбирать наиболее сильные модели как для непосредственного использования, так и для последующей тонкой настройки. Кроме того, мы подробно разобрали методы оценки моделей, ориентированных на конкретные предметные области и задачи, а также способы создания собственных систем оценки на основе общедоступных примеров.

В качестве двух основных подходов к построению пользовательских систем оценки мы сосредоточились на тестах с выбором из нескольких вариантов ответов и оценке LLM при помощи другой модели. Однако в большинстве случаев модели используются в составе более сложных систем, обеспечивающих дополнительный контекст.

Мы также обсудили два фреймворка для оценки RAG-систем — Ragas и ARES. Оба они имеют схожие черты (генерация синтетических данных) и различаются подходом к оценке: Ragas опирается на конкретные метрики, а ARES — на обученные классификаторы. Наконец, мы провели оценку ответов модели TwinLlama-3.1-8B с привлечением LLM в качестве эксперта по трем критериям: релевантность, связность и краткость, что помогло выявить возможные направления для ее улучшения.

В следующей главе обсудим методы оптимизации инференса, позволяющие повысить скорость работы и снизить потребление памяти без значительного ущерба для производительности модели. Мы также рассмотрим приемы использования параллелизма и квантизации.

Список используемых источников

- *Zheng L. et al.* Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. arXiv preprint arXiv:2306.05685.
- *Roucher A.* Using LLM-as-a-judge for an automated and versatile evaluation — Hugging Face Open-Source AI Cookbook. huggingface.co, https://huggingface.co/learn/cookbook/en/llm_judge.
- LangChain. Aligning LLM-as-a-Judge with Human Preferences. blog.langchain.dev. https://blog.langchain.dev/aligning-llm-as-a-judge-with-human-preferences/.
- *Hendrycks D. et al.* Measuring Massive Multitask Language Understanding. arXiv preprint arXiv:2009.03300.
- *Zhou J. et al.* Instruction-Following Evaluation for Large Language Models. arXiv preprint arXiv:2311.07911.
- *Dubois Y. et al.* Length-Controlled AlpacaEval: A Simple Way to Debias Automatic Evaluators. arXiv preprint arXiv:2404.04475.
- *Mialon G. et al.* GAIA: a benchmark for General AI Assistants. arXiv preprint arXiv:2311.12983.
- *Hong G. et al.* The Hallucinations Leaderboard — An Open Effort to Measure Hallucinations in Large Language Models. arXiv preprint arXiv:2404.05904.
- *Es S. et al.* RAGAS: Automated Evaluation of Retrieval Augmented Generation. arXiv preprint arXiv:2309.15217.
- *Saad-Falcon J. et al.* ARES: An Automated Evaluation Framework for Retrieval-Augmented Generation Systems. arXiv preprint arXiv:2311.09476.

8 Оптимизация инференса

Развертывание больших языковых моделей усложняется их требовательностью к вычислительным ресурсам и памяти. Для эффективной работы таких моделей нужны специализированные ускорители, например графические или тензорные процессоры, выполняющие операции в параллельном режиме и обеспечивающие высокую пропускную способность. Некоторые задачи (скажем, генерацию документов) можно выполнять пакетно в ночное время, тогда как другие (например, дополнение кода) требуют минимальной задержки и высокой скорости работы. Поэтому оптимизация инференса, то есть процесса получения предсказаний на основе входных данных, имеет решающее значение для практического применения моделей. Ключевыми задачами в данном случае являются сокращение времени до генерации первого токена, увеличение пропускной способности (числа токенов, генерируемых в секунду) и минимизация потребления памяти.

Базовые подходы к развертыванию приводят к неэффективному использованию оборудования и неадекватным значениям пропускной способности и задержки. К счастью, сегодня существует множество техник, позволяющих значительно ускорить инференс. В этой главе мы рассмотрим такие ключевые методы, как спекулятивное (предположительное) декодирование, модельный параллелизм и квантование весов модели, и покажем, как можно достичь ускорения в 2–4 раза за счет их грамотной реализации. Вы также познакомитесь с тремя популярными движками для инференса (Text Generation Inference, vLLM и TensorRT-LLM) и сравните их возможности в плане оптимизации данного процесса.

После изучения этой главы вы будете знать об основных сложностях, связанных с инференсом, и о таких передовых методах его оптимизации, как модельный параллелизм и квантование весов.



Все примеры кода из этой главы доступны в репозитории GitHub по адресу https://github.com/PacktPublishing/LLM-Engineering.

Стратегии оптимизации модели

Большинство современных больших языковых моделей, включая GPT и Llama, представляют собой трансформер, состоящий только из декодера (decoder-only). Эта архитектура специально разработана для генерации текста. Она предсказывает следующее слово в последовательности на основе предыдущих, что позволяет ей эффективно генерировать связные предложения, подходящие по контексту.

В свою очередь, архитектура, состоящая только из энкодера (encoder-only), как в случае модели BERT, ориентирована на понимание и представление входного текста в виде эмбеддингов. Она демонстрирует отличные результаты при решении задач, требующих глубокого понимания контекста, таких как классификация текста или распознавание именованных сущностей. Наконец, архитектура, состоящая из энкодера и декодера (encoder-decoder), примером которой является модель T5, объединяет оба подхода: энкодер анализирует входной текст и формирует его описание с учетом контекста, которое затем используется декодером для генерации выходного текста. Такая двухкомпонентная структура особенно хорошо подходит для решения задач, связанных с преобразованием последовательностей (перевод или резюмирование текста), в которых понимание входа и формирование релевантного выхода одинаково важны.

В книге мы ограничились обсуждением архитектуры, состоящей только из декодера, поскольку она доминирует в сфере LLM. Взгляните на рис. 8.1.

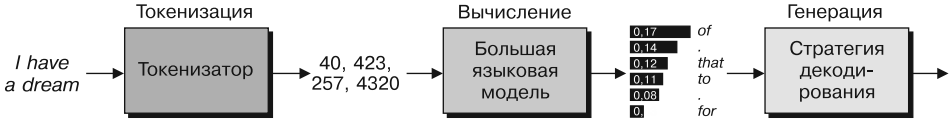


Рис. 8.1. Процесс инференса при использовании модели, состоящей только из декодера. На вход подается фраза I have a dream, а модель генерирует следующий токен — of

Как видно, базовый процесс инференса при использовании модели, состоящей только из декодера, включает следующие этапы.

1. *Токенизация* входного промпта, пропускание токенов через слой эмбеддинга и позиционное кодирование[1] (positional encoding).

[1] Чтобы модель понимала последовательность слов («кошка сидит на ковре» ≠ «ковер сидит на кошке»), каждому токену добавляют позиционный вектор, который отражает его место в предложении. Модель изначально получает батч векторов и не знает их последовательность. — *Примеч. науч ред.*

2. *Вычисление* пар «ключ — значение» для каждого входного токена с использованием механизма многоголового внимания (multi-head attention).
3. *Генерация* выходных токенов (по одному за раз) с использованием ранее вычисленных пар «ключ — значение».

Хотя первые два этапа являются вычислительно затратными, они хорошо распараллеливаются и эффективно выполняются на таких ускорителях, как графические и тензорные процессоры.

Главная проблема возникает на третьем этапе, поскольку генерация токенов — это по определению последовательный процесс. Чтобы сгенерировать очередной токен, необходимо иметь все предыдущие. Поскольку выходная последовательность строится токен за токеном, мы не можем в полной мере задействовать преимущества параллельных вычислений. Устранить это узкое место призваны методы оптимизации инференса.

В текущем разделе мы рассмотрим ряд стратегий, применяемых для ускорения инференса и снижения объема потребляемой *видеопамяти* (VRAM), включая реализацию (статического) кэша ключей и значений, непрерывную пакетную обработку, спекулятивное декодирование и оптимизированные механизмы внимания.

Кэш ключей и значений

Как мы уже видели, LLM генерируют текст токен за токеном. Данный процесс достаточно медленный, поскольку каждое новое предсказание зависит от всего предыдущего контекста. Например, чтобы предсказать 100-й токен, модели требуется информация о токенах с 1-го по 99-й. При генерации 101-го токена она учитывает токены с 1-го по 100-й. Такое повторное вычисление крайне неэффективно.

Кэш ключей и значений (KV-кэш) решает эту проблему, сохраняя пары «ключ — значение», создаваемые слоями самовнимания. Вместо того чтобы пересчитывать эти пары заново для каждого нового токена, модель просто извлекает их из кэша, что существенно ускоряет процесс генерации.

Суть данной техники представлена на рис. 8.2.

При генерации нового токена модель вычисляет только ключ и значение для этого одного токена, после чего добавляет их в кэш. Такая оптимизация реализована практически во всех популярных библиотеках и инструментах. В некоторых случаях для каждого слоя модели поддерживается отдельный KV-кэш.

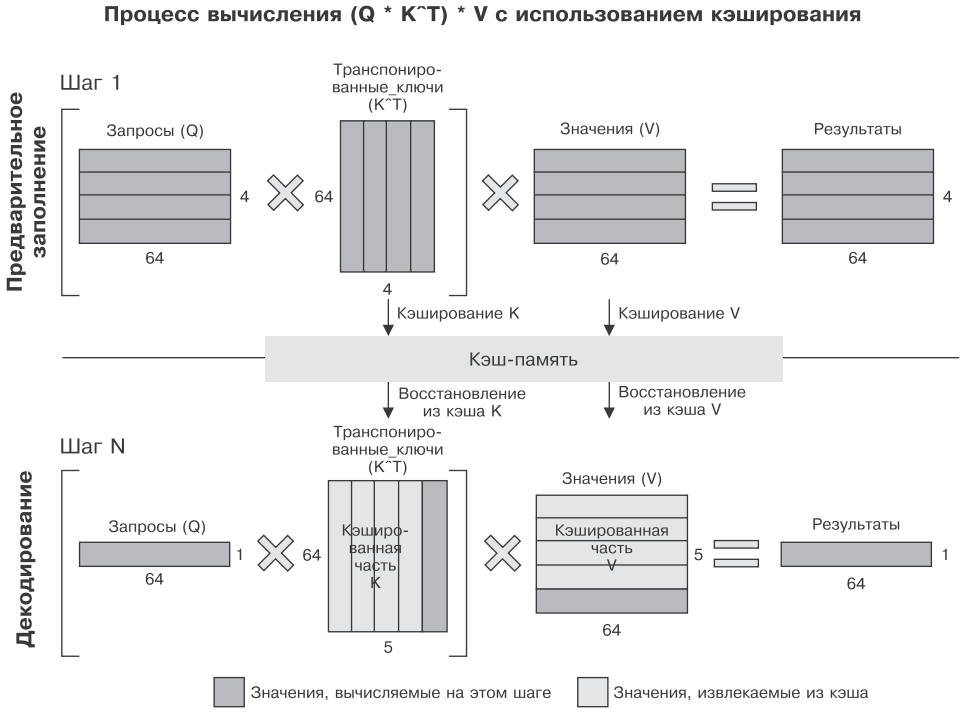


Рис. 8.2. Принцип работы KV-кэша

Размер кэша зависит от количества токенов ($n_{\text{токенов}}$), числа слоев в модели ($n_{\text{слоев}}$), числа голов внимания ($n_{\text{голов}}$), их размерности ($dim_{\text{головы}}$) и точности представления параметров в байтах ($n_{\text{байт}}$):

$$\text{размер(KV-кэша)} = 2n_{\text{токенов}}\, n_{\text{слоев}}\, n_{\text{голов}}\, dim_{\text{головы}}\, n_{\text{байт}}$$

Например, для модели с 7 млрд параметров, использующей 16-битную точность, объем кэша может превышать 2 Гбайт при длине последовательности более 2048 токенов. У более крупных моделей, с большим числом слоев и размерностью эмбеддингов, требования к памяти еще выше.

Поскольку объем KV-кэша растет с каждым шагом генерации и изменяется динамически, мы не можем задействовать `torch.compile` — мощный инструмент оптимизации, преобразующий PyTorch-код в быстрые и оптимизированные ядра. Данную проблему решает статический KV-кэш, для которого заранее выделяется максимальный объем памяти. Это позволяет использовать его совместно с `torch.compile`, получая ускорение вплоть до четырех раз при прямом проходе.

Чтобы настроить модель на применение статического KV-кэша с библиотекой `transformers`, нужно выполнить следующие шаги.

1. Сначала мы импортируем токенизатор и модель, которую требуется оптимизировать:

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

model_id = "google/gemma-2b-it"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id, device_map="auto")
```

2. Чтобы использовать статический кэш, необходимо изменить тип его реализации в конфигурации модели на `static`:

```
model.generation_config.cache_implementation = "static"
```

3. После этого можно скомпилировать модель с помощью `torch.compile`:

```
compiled_model = torch.compile(model, mode="reduce-overhead", fullgraph=True)
```

4. Далее мы токенизируем входной запрос, например `"What is 2 + 2?"`, и сохраняем результат на GPU при его наличии (в противном случае — на CPU):

```
device = "cuda" if torch.cuda.is_available() else "cpu"
inputs = tokenizer("What is 2 + 2?", return_tensors="pt").to(device)
```

5. Теперь давайте используем метод `generate()` для получения выходных данных модели и декодируем их с помощью `batch_decode()` для отображения ответа на экране:

```
outputs = model.generate(**inputs, do_sample=True, temperature=0.7,
➥max_new_tokens=64)
print(tokenizer.batch_decode(outputs, skip_special_tokens=True))
['What is 2 + 2?\n\nThe answer is 4. 2 + 2 = 4.']
```

В результате мы получаем список, содержащий как входные, так и выходные данные модели, корректно отвечающий на заданный вопрос.



Обратите внимание, что статический кэш работает не со всеми архитектурами. Для получения подробной информации о поддерживаемых архитектурах обратитесь к документации по библиотеке Transformers.

Эффективное управление KV-кэшем имеет ключевое значение, поскольку он может быстро исчерпать доступную видеопамять и ограничить допустимый размер пакета. Это стало одной из причин для разработки менее требовательных к памяти механизмов внимания и других техник, которые будут рассмотрены в заключительном разделе главы.

Непрерывная пакетная обработка

Пакетная обработка, то есть одновременное выполнение нескольких запросов к модели, — это стандартный способ повышения пропускной способности. Увеличение размера пакета позволяет распределить затраты памяти на хранение весов модели и передавать на GPU больше данных за раз, тем самым лучше задействуя его возможности в плане параллельных вычислений.

Однако модели с декодером (decoder-only) создают определенные сложности из-за сильной вариативности длины входных промптов и ожидаемых ответов. Одни запросы могут предусматривать короткие промпты, требующие всего одного слова в качестве ответа, а другие — включать объемный контекст и запрашивать развернутый ответ в несколько абзацев.

При традиционной пакетной обработке нам приходится дожидаться завершения обработки самого длинного запроса из пакета, чтобы перейти к следующему. Это приводит к неполной загрузке оборудования, поскольку часть времени ускоритель вынужден простаивать. Непрерывная пакетная обработка (continuous batching, также называемая in-flight batching) позволяет избежать этого простоя, подавая новый запрос в пакет сразу после завершения обработки предыдущего.

Процесс начинается стандартно — с заполнения пакета начальными запросами. Но затем, как только обработка одного из них завершается, он исключается из пакета и его место занимает следующий. Таким образом, ускоритель все время обрабатывает полный пакет, что обеспечивает максимально эффективное использование вычислительных ресурсов. Еще одним важным аспектом является необходимость периодически приостанавливать процесс генерации, чтобы выполнить предзаполнение, то есть векторизацию и кодирование ожидающих запросов. Для эффективной работы системы требуется найти оптимальный баланс между генерацией и предобработкой, подобрав подходящее значение гиперпараметра `waiting_served_ratio` (отношение ожидающих запросов к обрабатываемым).

Непрерывная пакетная обработка по умолчанию реализована в большинстве современных движков для инференса, включая Text Generation Inference (TGI) от Hugging Face, vLLM и NVIDIA TensorRT-LLM.

Спекулятивное декодирование

Еще одной мощной техникой оптимизации является *спекулятивное декодирование (speculative decoding)*, также называемое ассистированной генерацией. Основная идея заключается в том, что даже при применении непрерывной пакетной обработки пошаговая генерация токенов не позволяет в полной мере

задействовать вычислительные мощности ускорителя. Спекулятивное декодирование призвано использовать избыточные ресурсы для одновременного предсказания нескольких токенов с помощью более компактной вспомогательной модели (рис. 8.3).

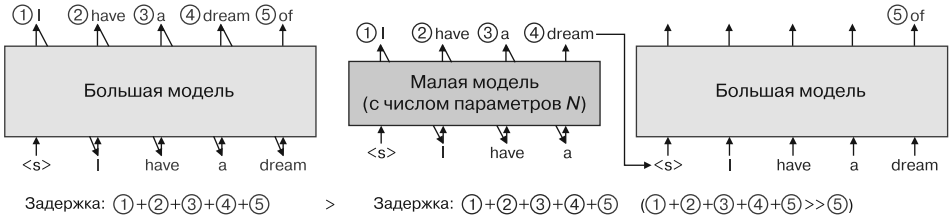


Рис. 8.3. Традиционное декодирование *(слева)* и спекулятивное декодирование *(справа)*

Рассмотрим процесс.

- Сначала применяется малая модель (например, дистиллированная или урезанная версия основной) для одновременного предсказания нескольких токенов. Обычно это 5–10 токенов за один шаг.
- Затем эти спекулятивные предсказания передаются в основную модель, которая проверяет, совпадают ли они с тем, что она могла бы сгенерировать сама.
- Наконец, наибольший совпадающий префикс сохраняется, а оставшиеся (ошибочные) токены отбрасываются.

Если вспомогательная модель достаточно точно приближает поведение основной, то можно генерировать несколько токенов за один шаг, избегая многократного запуска дорогостоящей большой модели. Степень ускорения напрямую зависит от качества предсказаний малой модели: при совпадении в 90 % возможен прирост в 3–4 раза.

Очень важно, чтобы обе модели использовали один и тот же токенизатор. В противном случае токены, сгенерированные вспомогательной моделью, не будут совпадать с токенами основной и их сопоставление окажется невозможным. Давайте реализуем эту технику с помощью библиотеки `Transformers`. В этом примере мы задействуем две модели Qwen1.5 от Alibaba Cloud: версию 1.8B в качестве основной и 0.5B — в роли вспомогательной. При наличии достаточного объема видеопамяти в качестве основной модели можно использовать более крупные версии — 14B, 32B, 72B и даже 110B.

В данном примере объем видеопамяти ограничен ресурсами графического процессора T4 в Google Colab. Однако для максимального ускорения важно, чтобы вспомогательная модель была существенно меньше основной.

Вот пошаговое описание процесса реализации спекулятивного декодирования.

1. Сначала загружаем токенизатор и обе модели:

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

model_id = "Qwen/Qwen1.5-1.8B-Chat"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id,
➥device_map="auto")
draft_model = AutoModelForCausalLM.from_pretrained("Qwen/Qwen1.5-0.5B-
➥Chat", device_map="auto")
```

2. Затем мы токенизируем один и тот же входной запрос и сохраняем его в памяти ускорителя (при его наличии):

```
device = "cuda" if torch.cuda.is_available() else "cpu"
inputs = tokenizer("What is 2 + 2?", return_tensors="pt").to(device)
```

3. После этого можно вызвать функцию `model.generate()`, передав ей аргумент `assistant_model`, чтобы активизировать режим спекулятивного декодирования:

```
outputs = model.generate(**inputs, do_sample=True,
➥assistant_model=draft_model, temperature=0.7, max_new_tokens=64)
print(tokenizer.batch_decode(outputs, skip_special_tokens=True))
['What is 2 + 2? 2 + 2 equals 4!']
```

Хотя в этом небольшом примере ускорение оказалось незначительным, при использовании более крупных моделей оно становится заметным.

Декодирование с поиском в промпте (prompt lookup decoding) — это вариант спекулятивного декодирования, специально адаптированный для задач, опирающихся на входной контекст, например резюмирования. Такие задачи часто предполагают значительное пересечение между промптом и ответом модели. В этом случае в качестве токенов-кандидатов используются общие *n*-граммы. Для активизации данного режима нужно передать в функцию `model.generate()` параметр `prompt_lookup_num_tokens`:

```
outputs = model.generate(**inputs, prompt_lookup_num_tokens=4)
```

Комбинируя статический KV-кэш, `torch.compile`, непрерывную пакетную обработку и спекулятивное декодирование, можно добиться ускорения инференса в 2–4 раза и более без потери качества.

Другой подход к построению вспомогательной модели состоит в совместной тонкой настройке малой и основной моделей для достижения максимального сходства в их предсказаниях. Типичным примером является метод Medusa, при котором в основную модель встраиваются специальные спекулятивные головы (speculation heads). В рамках подхода Medusa-1 тонко настраиваются только эти головы, в то время как основная модель остается замороженной. При подходе

Medusa-2 основная модель и спекулятивные головы тонко настраиваются совместно. Метод Medusa продемонстрировал весьма впечатляющие результаты: модель с 70 млн параметров смогла близко воспроизвести поведение модели с 7 млрд параметров при решении ряда задач. Спекулятивное декодирование реализовано в системе Text Generation Inference (TGI).

Оптимизированные механизмы внимания

Архитектура трансформеров основана на механизме внимания, вычислительная сложность которого растет квадратично по отношению к числу входных токенов (или длине последовательности). Она оказывается особенно неэффективной при работе с длинными последовательностями из-за резкого роста объема KV-кэша.

Метод PagedAttention, предложенный Квоном, Ли и соавторами (2023), решает проблему избыточного потребления памяти, заимствуя принципы использования виртуальной памяти и ее страничной организации из операционных систем. Вместо размещения KV-кэша в непрерывной области памяти он разбивает его на блоки фиксированного размера. Каждый блок содержит ключи и значения для определенного числа токенов. При вычислении функции внимания ядро PagedAttention эффективно извлекает нужные блоки вне зависимости от их физического расположения в памяти.

Такой подход позволяет добиться почти оптимального потребления памяти, обрабатывать больше последовательностей одновременно, повышая пропускную способность и загрузку GPU. Более того, блочная структура PagedAttention естественным образом поддерживает совместное использование памяти между несколькими вариантами вывода, сгенерированными на основе одного и того же промпта. Это особенно полезно при параллельной выборке (parallel sampling) и лучевом поиске (beam search). По утверждению авторов, использование общих блоков памяти позволяет сократить избыточные вычисления и снизить потребление ресурсов на 55 %, одновременно с этим повысив пропускную способность в 2,2 раза. Впервые метод PagedAttention был реализован в библиотеке vLLM, а впоследствии — в TGI и TensorRT-LLM.

Еще одним популярным подходом является разработанная Три Дао (2023) техника FlashAttention-2. Она предусматривает несколько ключевых инноваций, направленных на устранение ограничений по времени выполнения и памяти, присущих классическому механизму внимания. В рамках данного подхода входные и выходные матрицы разделяются на блоки, способные уместиться во встроенную SRAM-память графического процессора, которая работает гораздо быстрее, чем HBM (память с высокой пропускной способностью). Это существенно снижает количество передач данных между основной памятью GPU и его вычислительными блоками.

Кроме того, в рамках подхода FlashAttention-2 реализуется онлайн-вычисление softmax, при котором функция softmax вычисляется независимо для каждого блока матрицы весов внимания, а не для всей матрицы сразу. При этом сохраняются только текущий максимум и накопленная сумма экспонент, что позволяет вычислять вероятности внимания без хранения громоздких промежуточных матриц.

Онлайн-вычисление softmax также дает возможность выполнять поблочную обработку, сохраняя точность и значительно снижая требования к объему памяти. Это особенно важно на этапе обучения, так как повторное вычисление промежуточных значений при обратном проходе (вместо их хранения) позволяет уменьшить объем потребляемой памяти благодаря преобразованию его зависимости от длины последовательности с квадратичной на линейную.

В отличие от PagedAttention технику FlashAttention-2 можно легко использовать в связке с библиотекой `Transformers` — достаточно указать нужный тип реализации с помощью параметра `attn_implementation`.

1. Сначала установите библиотеку `flash-attn` с флагом `--no-build-isolation`, чтобы избежать установки зависимостей:

```
pip install flash-attn --no-build-isolation
```

2. Чтобы использовать FlashAttention-2 для инференса, укажите значение `flash_attention_2` в параметре `attn_implementation` при загрузке модели. Например, вот как можно загрузить модель Mistral-7B-Instruct-v0.3 с включенной поддержкой FlashAttention-2:

```
from transformers import AutoModelForCausalLM
model = AutoModelForCausalLM.from_pretrained(
    "mistralai/Mistral-7B-Instruct-v0.3",
    attn_implementation="flash_attention_2",
)
```

Методы, представленные в этом разделе, направлены на повышение эффективности обработки токенов. Далее мы обсудим способы распределения вычислений между несколькими GPU.

Модельный параллелизм

Модельный параллелизм (model parallelism) позволяет распределять нагрузку между несколькими графическими процессорами. Это дает возможность осуществлять инференс и обучать модели, которые не помещаются на одном устройстве, а также повышает их производительность в плане пропускной способности (число токенов в секунду).

Для реализации модельного параллелизма существует три основных подхода, каждый из которых предусматривает свои способы распределения весов и вычислений: *параллелизм данных*, *конвейерный параллелизм* и *тензорный параллелизм*.

Хотя изначально эти подходы разрабатывались для обучения моделей, их можно применять и для инференса, если сосредоточиться только на прямом проходе.

Параллелизм данных

Параллелизм данных (data parallelism, DP) представляет собой самый простой вариант модельного параллелизма. Его суть заключается в создании копий модели и их распределении между разными графическими процессорами (рис. 8.4), которые одновременно обрабатывают собственные подмножества данных. Градиенты, вычисленные на каждом GPU, в процессе обучения усредняются и используются для обновления параметров модели, что обеспечивает синхронизацию ее копий. Этот подход особенно полезен в ситуациях, когда размер пакета слишком велик для одного устройства и когда требуется ускорить процесс обучения.

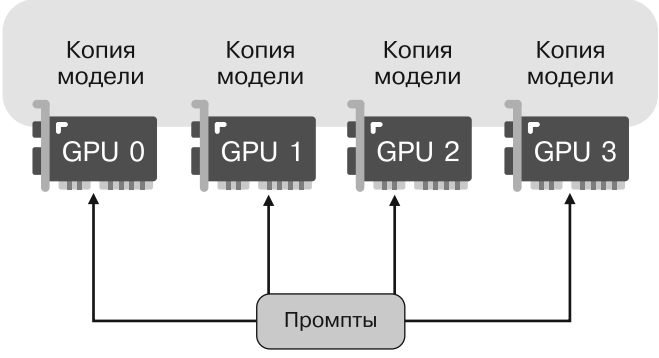


Рис. 8.4. Реализация параллелизма данных с использованием четырех GPU

На этапе инференса параллелизм данных может использоваться для обработки нескольких запросов в конкурентном режиме. Распределяя нагрузку между несколькими GPU, такой метод позволяет минимизировать задержки и повысить пропускную способность за счет обработки большего числа запросов за единицу времени.

Однако эффективность этого подхода ограничена размерами модели и затратами на межпроцессорное взаимодействие. Дублирование параметров модели на каждом графическом процессоре приводит к неэффективному расходованию

памяти, поэтому данный подход применим только в тех случаях, когда модель достаточно мала, чтобы поместиться в памяти одного устройства. Это, в свою очередь, сокращает доступный объем памяти для входных данных и ограничивает допустимый размер пакета. При работе с большими моделями или в условиях ограниченной памяти такой недостаток становится весьма существенным.

Обычно параллелизм данных реализуется на этапе обучения моделей, а на этапе инференса предпочтительными подходами являются конвейерный и тензорный параллелизм.

Конвейерный параллелизм

Конвейерный параллелизм (pipeline parallelism, PP), предложенный Хуаном и соавторами в статье о GPipe (2019), представляет собой стратегию распределения вычислительной нагрузки, связанной с обучением и использованием больших нейросетей, между несколькими GPU.

В отличие от параллелизма данных, при котором каждый GPU содержит полную копию модели, конвейерный параллелизм предполагает разделение слоев модели между различными GPU. Такой подход позволяет каждому GPU обрабатывать только определенную часть модели, что снижает требования, предъявляемые к объему памяти конкретного устройства.

Как показано на рис. 8.5, модель делится на четыре сегмента, каждый из которых назначается отдельному GPU. Например, первые 25 % слоев модели обрабатываются GPU 1, следующие 25 % — GPU 2 и т. д. Во время прямого прохода активации вычисляются и передаются следующему GPU. При обучении выполняется обратный проход с распространением градиентов по всем GPU. Число GPU часто называется степенью параллелизма.

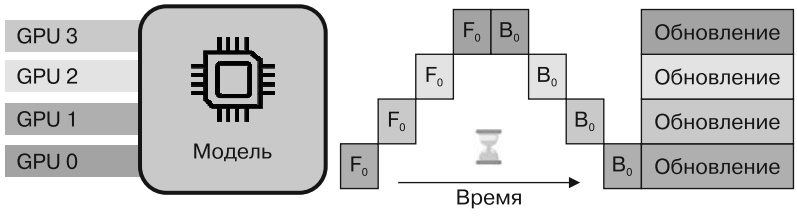


Рис. 8.5. Реализация конвейерного параллелизма с использованием четырех GPU

Основное преимущество конвейерного параллелизма — существенное снижение требований к объему памяти каждого GPU. Однако у такого подхода есть и свои сложности, связанные с последовательной природой пайплайна. Одной из главных проблем являются так называемые «конвейерные пузыри» (pipeline bubble) — периоды простоя, в течение которых отдельные GPU вынуждены

дожидаться окончания вычислений в предыдущих слоях, что снижает общую эффективность рабочего процесса.

Для борьбы с этими пузырями была разработана техника микропакетной обработки (micro-batching). Она предполагает разбиение входного пакета на более мелкие подпакеты, благодаря чему GPU остаются задействованными, так как следующий подпакет начинает обрабатываться до завершения обработки предыдущего.

На рис. 8.6 продемонстрирована реализация конвейерного параллелизма с микропакетной обработкой. В данном случае пайплайн предусматривает четыре этапа (F_0, F_1, F_2, F_3), а входной пакет разбит на четыре подпакета. GPU 0 последовательно обрабатывает подпакеты $F_{0.0}$, $F_{0.1}$, $F_{0.2}$ и $F_{0.3}$. Сразу после завершения обработки $F_{0.0}$ GPU 1 начинает обрабатывать $F_{0.1}$ и т. д. После завершения прямого прохода GPU 0 ожидает окончания соответствующих этапов работы остальных GPU, чтобы приступить к обратному проходу ($B_{0.3}$, $B_{0.2}$, $B_{0.1}$ и $B_{0.0}$).

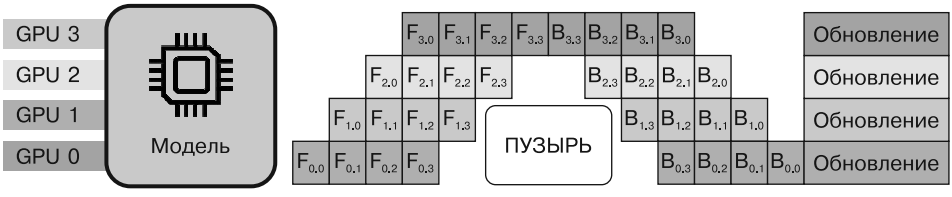


Рис. 8.6. Реализация конвейерного параллелизма с микропакетной обработкой

Конвейерный параллелизм реализован в таких фреймворках для распределенного обучения, как Megatron-LM, DeepSpeed (ZeRO) и PyTorch, который предусматривает специальную библиотеку *PiPPy* (Pipeline Parallelism for PyTorch). Что касается систем для инференса, то на момент написания книги данный вид параллелизма поддерживали лишь некоторые из них, в частности TensorRT-LLM.

Тензорный параллелизм

Тензорный параллелизм (tensor parallelism, TP), описанный Шуби, Патвари, Пури и соавторами в статье о Megatron-LM (2019), представляет собой еще одну популярную технику распределения вычислений, осуществляемых в слоях LLM, между несколькими устройствами. В отличие от конвейерного параллелизма данный подход предполагает разделение не самой модели, а матриц весов внутри ее слоев, что позволяет выполнять вычисления параллельно, существенно снижая нагрузку на память и повышая скорость обработки.

При реализации тензорного параллелизма крупные матрицы, такие как матрицы весов в многослойных перцептронах (MLP) или головы внимания в слоях

самовнимания, разделяются между несколькими GPU. Каждый GPU хранит свою часть матрицы и выполняет вычислительные операции только над ней.

Например, в слое MLP весовая матрица делится по столбцам так, что каждый GPU обрабатывает только свое подмножество весов (рис. 8.7). Входные данные передаются на все GPU, каждый из которых вычисляет свои выходы независимо от других. Затем эти частичные результаты объединяются в итоговый вывод с помощью операции all-reduce.

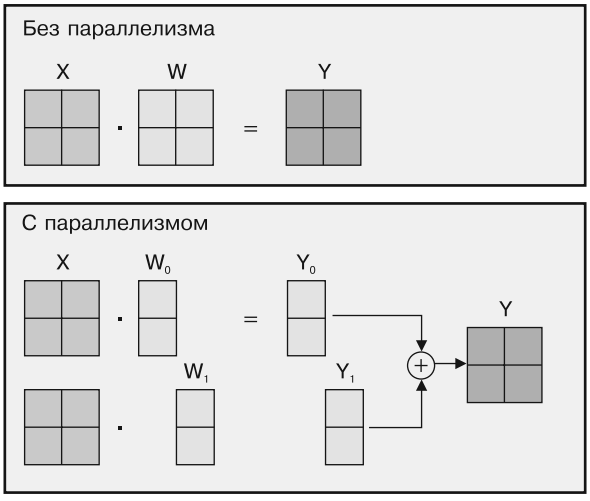


Рис. 8.7. Реализация тензорного параллелизма путем разделения матрицы весов (W) в слое MLP по столбцам

В слоях самовнимания метод TP особенно эффективен благодаря параллелизму, присущему головам внимания. Каждый GPU может независимо обрабатывать свое подмножество голов, ускоряя тем самым обработку длинных последовательностей. В этом отношении тензорный параллелизм оказывается более производительным, чем конвейерный, так как исключает ожидание завершения вычислений в предыдущих слоях.

Тем не менее метод TP можно применить не ко всем слоям нейросети. Например, слои LayerNorm и Dropout имеют зависимости, охватывающие всю входную последовательность, и не могут быть эффективно разделены между GPU, поэтому обычно их дублируют на всех устройствах. Однако в таких случаях можно реализовать параллелизм последовательностей (sequence parallelism), при котором разные графические процессоры обрабатывают разные фрагменты входной последовательности без дублирования весов. Хотя этот метод применим лишь к ограниченному числу слоев, он позволяет дополнительно снизить потребление памяти, особенно при работе с очень длинными входными последовательностями.

Применение метода TP требует высокоскоростных каналов связи между устройствами для минимизации издержек на передачу данных. Поэтому его трудно реализовать в сети с ограниченной пропускной способностью.

Тензорный параллелизм поддерживается такими фреймворками для распределенного обучения, как Megatron-LM, DeepSpeed (ZeRO) и PyTorch (FSDP), а также большинством систем для инференса, включая TGI, vLLM и TensorRT-LLM.

Комбинирование подходов

Параллелизм данных, тензорный и конвейерный параллелизм — это техники, которые можно сочетать. На рис. 8.8 показано, как одну и ту же модель можно разделить с использованием этих трех подходов.

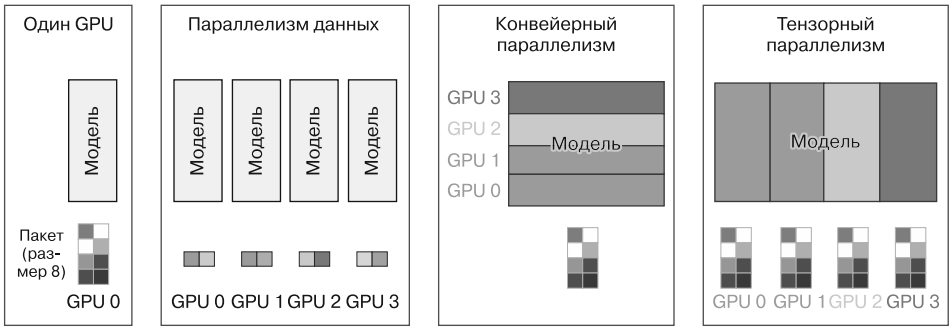


Рис. 8.8. Реализация различных видов модельного параллелизма

Комбинация этих подходов позволяет компенсировать ограничения, присущие каждому из них. Конвейерный параллелизм обеспечивает наибольшее снижение требований к памяти, но при этом уступает в эффективности из-за появления «пузырей». Данный метод особенно уместен, если основное ограничение связано с размещением модели в памяти GPU. Напротив, если приоритетом является минимизация задержек, имеет смысл отдать предпочтение тензорному параллелизму, смирившись с большим объемом потребляемой памяти. На практике модель часто разбивается на сегменты, обрабатываемые на разных GPU (конвейерный параллелизм), а внутри этих сегментов дополнительно реализуется тензорный параллелизм.

Поиск оптимального баланса между упомянутыми компромиссами и согласование архитектуры модели с возможностями доступных аппаратных ускорителей представляют собой одни из ключевых задач при развертывании больших языковых моделей.

Квантование модели

Квантование — это процесс представления весов и активаций нейронной сети с использованием типов данных меньшей точности. В контексте LLM под квантованием в первую очередь понимается понижение разрядности весов и активаций модели.

Обычно веса сохраняются в 16- или 32-битном формате с плавающей запятой (FP16 или FP32), который обеспечивает высокую точность, но предъявляет повышенные требования к объему памяти и вычислительным ресурсам. Квантование позволяет уменьшить потребление памяти и ускорить работу LLM на этапе инференса.

Кроме того, крупные модели с числом параметров более 30 млрд могут превосходить в производительности менее крупные модели (7–13 млрд), даже будучи квантованными до 2 или 3 бит. Это означает, что они способны демонстрировать более высокие результаты при сопоставимом объеме потребляемой памяти.

В данном разделе мы обсудим принцип квантования весов, формат GGUF и его использование с библиотекой `llama.cpp`, методы GPTQ и EXL2, а также ряд дополнительных техник. Помимо приведенного в этом разделе кода, для квантования моделей можно использовать блокнот Google Colab проекта AutoQuant (bit.ly/autoquant).

Принцип квантования весов

Существует два основных подхода к квантованию весов: *квантование после обучения (Post-Training Quantization, PTQ)* и *обучение с учетом квантования (Quantization-Aware Training, QAT)*. PTQ — это метод, в рамках которого веса уже обученной модели преобразуются в формат с пониженной разрядностью без дополнительной тонкой настройки. Несмотря на простоту реализации, такой подход может привести к снижению уровня производительности модели. В свою очередь, метод QAT предполагает выполнение квантования в процессе обучения или тонкой настройки, что позволяет модели адаптироваться к весам пониженной точности. Как правило, метод QAT помогает добиться более высокой производительности, чем PTQ, но требует дополнительных вычислительных ресурсов и репрезентативных обучающих данных.

Выбор типа данных играет ключевую роль в процессе квантования. В глубоких нейронных сетях обычно используются форматы чисел с плавающей запятой — *FP32*, *FP16* (число половинной точности) и *BF16* (brain floating-point, или bfloat, формат чисел с плавающей запятой, разработанный командой Google Brain). Эти форматы выделяют фиксированное количество битов для представления знака, экспоненты и мантиссы числа.

Если бит знака равен 0, то число положительное, а если 1 — отрицательное. Экспонента определяет диапазон представляемых значений — от самых больших до самых малых, а мантисса (значимая часть) определяет точность числа, то есть количество значащих цифр. Для преобразования этих представлений в действительные числа используется следующая формула:

$$(-1)^{\text{знак}} \times \text{основание}^{\text{экспонента}} \times \text{мантисса}.$$

Разные форматы данных, показанные на рис. 8.9, предполагают различные компромиссы, как видно на примере представления числа π (≈ 3,1415926535). Формат FP32 использует 32 бита и обеспечивает высокую точность, но требует больше памяти. Напротив, форматы FP16 и BF16 используют всего 16 бит, что позволяет уменьшить объем потребляемой памяти за счет снижения точности. В целом нейронные сети отдают предпочтение более широкому диапазону значений, а не высокой точности, поэтому при наличии аппаратной поддержки наиболее популярным форматом данных является BF16. Так, архитектура NVIDIA Ampere (A100, A30 и т. п.) поддерживает формат BF16 в отличие от предыдущих поколений, вроде Turing (T4, T40 и т. п.).

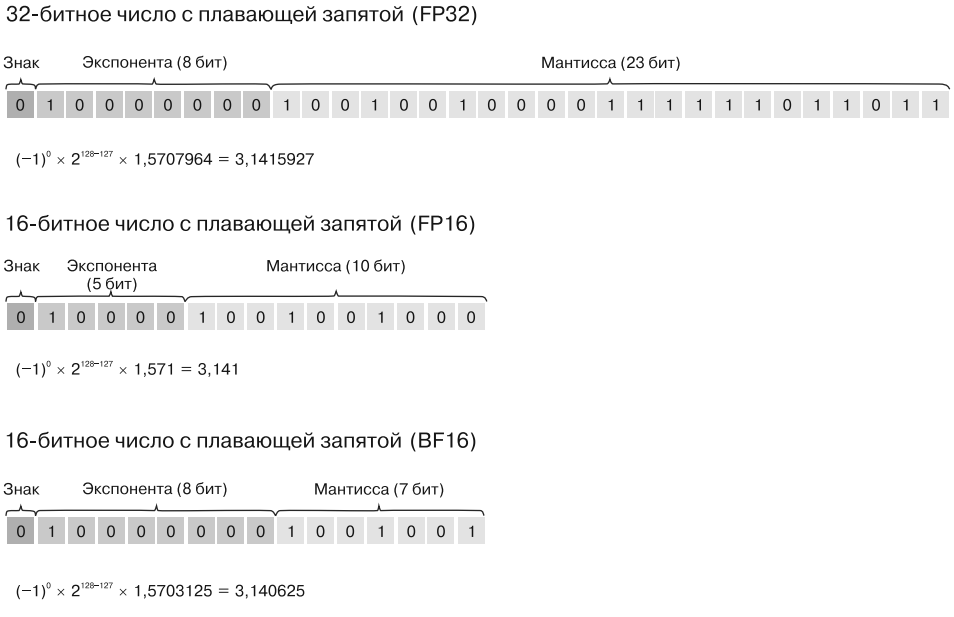


Рис. 8.9. Сравнение форматов FP32, FP16 и BF16

Однако мы не ограничены только этими тремя форматами. Для квантования можно задействовать и другие типы данных, например INT8 (8-битные целые числа), чтобы еще сильнее сократить объем потребляемой памяти. Для преобразования весов в формате FP32, FP16 или BF16 в формат INT8 допускается

использовать такие стандартные методы квантования, как absmax (по абсолютному максимуму) и zero-point (со смещением нуля), как показано на рис. 8.10.

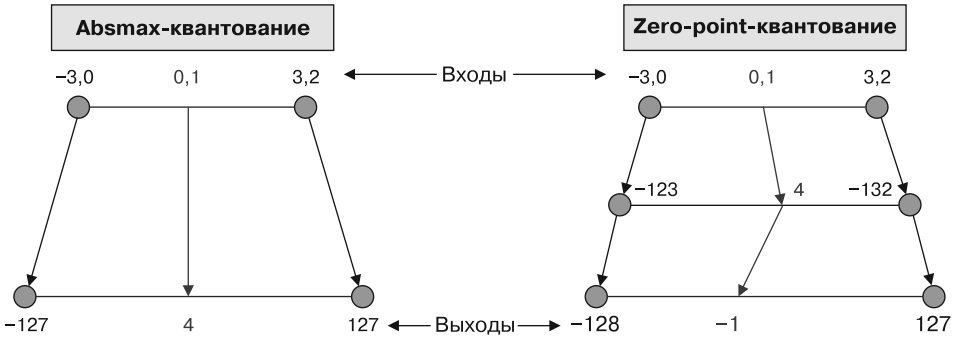


Рис. 8.10. Квантование значения 0,1 в диапазоне [–3,0, 3,2] с помощью методов absmax и zero-point

Absmax-квантование предполагает отображение исходных весов X в диапазон [–127, 127] путем их деления на абсолютное максимальное значение X и масштабирования:

$$X_{\text{квант}} = \text{округление}\left(\frac{127 \times X}{\max|X|}\right).$$

Например, если абсолютный максимум равен 3,2 (см. рис. 8.10), то вес 0,1 будет квантован так: округляем результат выражения $\left(\frac{127 \times 0{,}1}{3{,}2}\right)$ и получаем 4. Для его деквантования (восстановления) выполняется обратная операция:

$$X_{\text{деквант}} = \frac{\max|X| \times X_{\text{квант}}}{127}.$$

В нашем примере: $\frac{3{,}2 \times 4}{127} \approx 0{,}1008$. Таким образом, в данном случае при восстановлении значения веса возникает ошибка округления, равная 0,0008. Этот метод квантования можно реализовать на языке Python, например, с помощью библиотеки PyTorch:

```
import torch

def absmax_quantize(X):
    # Вычисление коэффициента масштабирования
    scale = 127 / torch.max(torch.abs(X))

    # Квантование
    X_quant = (scale * X).round()

    return X_quant.to(torch.int8)
```

В свою очередь, zero-point-квантование учитывает асимметричное распределение входных данных и отображает веса в диапазон [–128, 127], вводя смещение нуля:

$$X_{\text{квант}} = \text{округление}(\text{КМ} \times X + \text{смещение}),$$

где КМ (коэффициент масштабирования) = $\frac{255}{\max(X) - \min(X)}$, а смещение = –округление(КМ × min(X)) – 128.

В примере с весом 0,1 коэффициент масштабирования будет равен $\frac{255}{3{,}2 + 3{,}0} \approx$ ≈ 41,13, а смещение — –округление ($\frac{255}{3{,}2 + 3{,}0} \times -3{,}0$) – 128 = –5. Таким образом, вес 0,1 будет преобразован в округление (41,13 × 0,1 – 5) = –1, а не в 4, как при использовании метода absmax.

Обратное преобразование (деквантование) выполняется по формуле:

$$X_{\text{деквант}} = \frac{X_{\text{квант}} - \text{смещение}}{\text{КМ}}.$$

На языке Python zero-point-квантование можно реализовать следующим образом:

```
def zeropoint_quantize(X):
    # Вычисление диапазона значений (знаменателя)
    x_range = torch.max(X) - torch.min(X)
    x_range = 1 if x_range == 0 else x_range

    # Вычисление коэффициента масштабирования
    scale = 255 / x_range

    # Вычисление величины смещения
    zeropoint = (-scale * torch.min(X) - 128).round()

    # Масштабирование и округление входных значений
    X_quant = torch.clip((X * scale + zeropoint).round(), -128, 127)

    return X_quant.to(torch.int8)
```

Однако стандартные методы квантования имеют свои ограничения, особенно при работе с *выбросами*, то есть экстремальными значениями весов LLM (составляющими около 0,1 % от общего числа), которые могут существенно повлиять на процесс квантования, снизив точность представления остальных значений. Удаление выбросов нецелесообразно, поскольку может привести к ухудшению производительности модели. Пример таких выбросов показан на рис. 8.11.

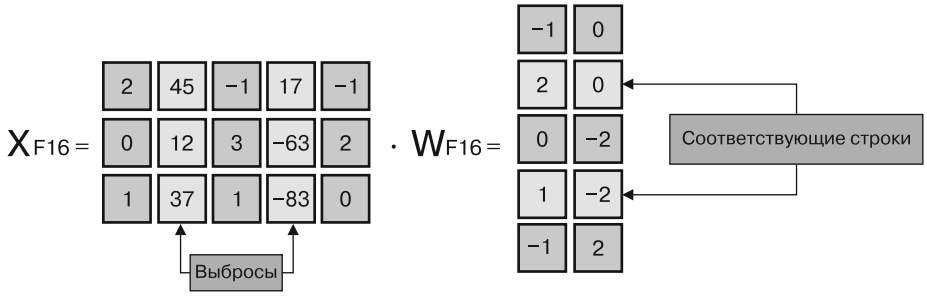


Рис. 8.11. Пример выбросов в матрице весов

Для решения проблемы выбросов были разработаны более продвинутые методы квантования. Одним из ярких примеров является метод `LLM.int8()`, предложенный Деттмерсом и соавторами в 2022 году. Этот метод использует схему квантования со смешанной точностью, в рамках которой выбросы обрабатываются в формате FP16, а оставшиеся значения преобразуются в формат INT8. Такой подход позволяет почти вдвое сократить объем потребляемой LLM памяти при минимальных потерях в ее производительности.

Метод `LLM.int8()` предполагает выполнение матричного умножения в три этапа. Сначала из скрытых состояний входных данных извлекаются содержащие выбросы столбцы с учетом заданного порогового значения. Затем выполняются операции матричного умножения отдельно над выбросами (в формате FP16) и над остальными значениями (в формате INT8) в режиме повекторного квантования. Наконец, результаты обработки обычных значений деквантуются и объединяются с результатами обработки выбросов для получения финального выхода в формате FP16.

Эффективность метода `LLM.int8()` была подтверждена экспериментально: в сравнении с исходными моделями в формате FP32 потери производительности при его использовании составляют менее 1 %. Однако при этом растут вычислительные затраты, из-за чего время инференса для больших моделей увеличивается примерно на 20 %. Мы можем загружать модели непосредственно в 8-битной точности с помощью библиотеки `Transformers`, используя метод `LLM.int8()` следующим образом:

```
from transformers import AutoModelForCausalLM

model_name = "meta-llama/Meta-Llama-3-8B-Instruct"
model = AutoModelForCausalLM.from_pretrained(model_name, device_map="auto",
➥load_in_8bit=True)
```

В 2023 году Деттмерс с соавторами также предложил формат с 4-битной точностью NF4, предназначенный для метода QLoRA (описанного в главе 5). Он тоже

интегрирован в библиотеку `transformers`, но требует пакет `bitsandbytes` в качестве зависимости. Для загрузки модели в формате NF4 (4-битная точность) можно использовать параметр `load_in_4bit`, как показано ниже:

```
from transformers import AutoModelForCausalLM

model_name = "meta-llama/Meta-Llama-3-8B-Instruct"
model = AutoModelForCausalLM.from_pretrained(model_name, device_map="auto",
➥load_in_4bit=True)
```

Квантование с использованием формата GGUF и библиотеки llama.cpp

Проект `llama.cpp` — это библиотека с открытым исходным кодом, созданная Георгием Гергановым на языке C++. Она предназначена для обеспечения работы различных LLM на этапе инференса и в настоящее время является самым популярным инструментом для квантования (множество квантованных моделей доступны на платформе Hugging Face Hub).

В отличие от других библиотек, зависящих от проприетарных и аппаратно-специфичных решений, таких как CUDA, `llama.cpp` может работать на более широком спектре устройств. Она получила особую популярность среди пользователей, не располагающих специализированным оборудованием, так как позволяет выполнять инференс даже на CPU и Android-устройствах. Кроме того, `llama.cpp` поддерживает перенос вычислений на GPU для ускорения инференса и совместима с такими техниками его оптимизации, как FlashAttention-2 и спекулятивное декодирование.

В рамках проекта `llama.cpp` задействуется собственный формат квантования GGUF, предназначенный для упрощения и ускорения загрузки моделей. Файлы GGUF хранят тензоры и метаданные, поддерживая различные форматы с точностью от 1 до 8 бит. При их именовании применяется соглашение, основанное на числе битов, например:

- `IQ1_S` и `IQ1_M` — 1 бит: очень низкое качество;
- `IQ2_XXS/XS/S/M` и `Q2_K` — 2 бита: как правило, низкое качество, однако IQ2 может использоваться с крупными моделями;
- `IQ3_XXS/XS/S/M` и `Q3_K_S/M/L` — 3 бита: низкое качество, но приемлемо для крупных моделей;
- `IQ4_XS/NL` и `Q4_K_S/M`, `Q4_0/1` — 4 бита: хорошее качество, подходит для большинства моделей;
- `Q5_K_S/M` и `Q5_0/1` — 5 бит: высокое качество;
- `Q6_K` — 6 бит: очень высокое качество;
- `Q8_0` — 8 бит: наивысшее качество.

В двух словах механизм GGUF-квантования можно описать так: библиотека `llama.cpp` группирует значения в блоки и понижает их разрядность путем округления. Например, устаревший формат Q4_0 обрабатывает по 32 значения в каждом блоке, масштабируя и квантуя их относительно наибольшего значения веса в блоке ($w = q \times$ КМ_блока, где КМ — коэффициент масштабирования). В формате Q4_1 к этому добавляется наименьшее значение в блоке ($w = q \times$ КМ_блока + мин_блока). В формате Q4_K веса разделяются на суперблоки, каждый из которых состоит из восьми блоков по 32 значения. При этом коэффициенты масштабирования и минимальные значения блоков тоже квантуются с повышенной точностью 6 бит ($w = q \times$ КМ_блока(6 бит) + мин_блока(6 бит)). Наконец, такие i-кванты, как IQ4_XS, вдохновлены другим методом квантования под названием QuIP#. Этот подход обеспечивает четное число положительных (или отрицательных) знаков квантов в группах по восемь значений и использует решетку E_8 для хранения их величин.

Ниже приведен практический пример квантования модели в формате GGUF. Эти шаги можно выполнить на бесплатном графическом процессоре T4 в среде Google Colab.

1. Первым делом нужно установить `llama.cpp` и другие сопутствующие библиотеки:

```
!git clone https://github.com/ggerganov/llama.cpp
!cd llama.cpp && git pull && make clean && LLAMA_CUBLAS=1 make
!pip install -r llama.cpp/requirements.txt
```

2. Далее необходимо загрузить модель, которую требуется преобразовать. Мы укажем идентификатор модели, используемый на Hugging Face Hub, например `mistralai/Mistral-7B-Instruct-v0.2`:

```
MODEL_ID = "mlabonne/EvolCodeLlama-7b"
MODEL_NAME = MODEL_ID.split('/')[-1]
!git lfs install
!git clone https://huggingface.co/{MODEL_ID}
```

3. Сначала преобразуем модель в формат FP16. Этот промежуточный артефакт будет использоваться для всех типов GGUF-квантования. Обратите внимание, что в `llama.cpp` предусмотрены разные сценарии конвертации, совместимые с разными типами моделей:

```
fp16 = f"{MODEL_NAME}/{MODEL_NAME.lower()}.fp16.bin"
!python llama.cpp/convert.py {MODEL_NAME} --outtype f16 –outfile {fp16}
```

4. Далее мы выбираем формат квантования (в данном случае Q4_K_M) и запускаем процесс, выполнение которого на T4 может занять около часа:

```
METHOD = "q4_k_m"
qtype = f"{MODEL_NAME}/{MODEL_NAME.lower()}.{method.upper()}.gguf"
!./llama.cpp/quantize {fp16} {qtype} {METHOD}
```

5. После завершения квантования модель можно сохранить локально или загрузить на платформу Hugging Face Hub с помощью следующего кода:

```
from huggingface_hub import create_repo, HfApi

hf_token = "" # Укажите свой токен
username = "" # Укажите свое имя пользователя

api = HfApi()

# Создайте пустой репозиторий
create_repo(
    repo_id = f"{username}/{MODEL_NAME}-GGUF",
    repo_type="model",
    exist_ok=True,
    token=hf_token
)

# Загрузите gguf-файлы
api.upload_folder(
    folder_path=MODEL_NAME,
    repo_id=f"{username}/{MODEL_NAME}-GGUF",
    allow_patterns=f"*.gguf",
    token=hf_token
)
```

GGUF-модели можно использовать с такими бэкендами, как `llama-cpp-python`, и фреймворками вроде LangChain. Это особенно удобно, если вы собираетесь интегрировать квантованную модель в более крупную систему. Кроме того, вы можете взаимодействовать с моделью напрямую с помощью легковесного сервера `llama.cpp`, LM Studio или Text Generation Web UI. Эти инструменты предоставляют простой интерфейс для взаимодействия с GGUF-моделями, напоминающий интерфейс ChatGPT.

Квантование с использованием форматов GPTQ и EXL2

Если GGUF и `llama.cpp` предназначены для выполнения инференса на CPU, но позволяют частично задействовать ресурсы GPU при его наличии, то форматы GPTQ (Generalized Post-Training Quantization) и EXL2 (ExLlamaV2) изначально ориентированы на GPU. Благодаря этому они обеспечивают более высокую скорость работы модели на этапе инференса по сравнению с `llama.cpp`. В частности, EXL2 характеризуется наибольшей пропускной способностью благодаря специализированной библиотеке ExLlamaV2.

Оба формата основаны на алгоритме GPTQ, представленном Франтаром и соавторами в 2023 году. Этот алгоритм оптимизирует квантование весов больших языковых моделей, уточняя метод *OBQ* (Optimal Brain Quantization) для более

эффективной обработки обширных матриц. Процесс начинается с разложения обратной матрицы Гессе методом Холецкого, что обеспечивает вычислительную устойчивость. Вместо квантования весов в жестко заданном порядке алгоритм GPTQ обрабатывает их пакетами, итеративно обновляя столбцы и соответствующие блоки. Благодаря использованию ленивых пакетных обновлений этот подход сокращает объем избыточных вычислений и уменьшает нагрузку на память.

Если формат GPTQ ограничен 4-битной точностью, то формат EXL2 отличается большей гибкостью. Он допускает настройку точности с использованием от 2 до 8 бит на вес, включая дробные значения, такие как 2,3, 3,5 или 6,0. Кроме того, EXL2 позволяет применять разные уровни квантования внутри одного линейного слоя, повышая точность представления наиболее значимых весов. Параметры выбираются автоматически. Для этого каждая матрица квантуется несколько раз, после чего выбирается такая комбинация, которая минимизирует ошибку квантования при заданном значении битрейта. Таким образом, мы можем запускать модели с 70 млрд параметров на одном GPU с 24 Гбайт памяти с точностью 2,55 бита.

При этом сам инференс выполняется с помощью библиотеки ExLlamaV2, поддерживающей модели как в формате GPTQ, так и в формате EXL2.

В следующем примере мы выполним квантование модели в формате EXL2 с помощью библиотеки ExLlamaV2. Эти шаги можно реализовать на бесплатном графическом процессоре T4 в среде Google Colab.

1. Сначала необходимо установить библиотеку ExLlamaV2:

```
!git clone https://github.com/turboderp/exllamav2
!pip install -e exllamav2
```

2. Затем мы загружаем модель, подлежащую квантованию, путем клонирования ее репозитория с платформы Hugging Face Hub:

```
MODEL_ID = "meta-llama/Llama-2-7b-chat-hf"
MODEL_NAME = MODEL_ID.split('/')[-1]
!git lfs install
!git clone https://huggingface.co/{MODEL_ID}
```

3. Далее нужно загрузить калибровочный набор данных для измерения ошибки квантования. В этом примере мы воспользуемся стандартным калибровочным набором WikiText-103, который содержит качественные статьи из Википедии:

```
!wget https://huggingface.co/datasets/wikitext/
resolve/9a9e482b5987f9d25b3a9b2883fc6cc9fd8071b3/wikitext-103-v1/wikitext-
test.parquet
```

4. Теперь выполняем квантование модели с заданной точностью (например, 4,5 бита на вес):

```
!mkdir quant
!python exllamav2/convert.py \
    -i {MODEL_NAME} \
    -o quant \
    -c wikitext-test.parquet \
    -b 4.5
```

Квантованную модель можно загрузить на платформу Hugging Face Hub, как было показано ранее.

Форматы GPTQ и EXL2 пока не так широко распространены, как GGUF. Например, интерфейсы вроде LM Studio в настоящее время с ними не работают. Однако вместо них можно использовать такие инструменты, как Text Generation Web UI от oobabooga. Кроме того, данные форматы поддерживаются в библиотеке `Transformers` и системе TGI, а GPTQ-модели — в TensorRT-LLM.

Хотя GPTQ и EXL2 менее популярны по сравнению с GGUF, на платформе Hugging Face Hub уже доступно множество моделей в этих форматах.

Прочие методы квантования

Помимо GGUF, GPTQ и EXL2, существует целый ряд других методов квантования. В этом разделе мы кратко обсудим алгоритм *AWQ* (Activate-aware Weight Quantization — «квантование весов с учетом активаций»), а также методы экстремального квантования, такие как QuIP# (Quantization with Incoherence Processing — «квантование с непоследовательной обработкой») и HQQ (Half-Quadratic Quantization — «полуквадратичное квантование»).

Алгоритм AWQ, предложенный Лином и соавторами в 2023 году, представляет собой еще один популярный метод квантования, который определяет и защищает наиболее значимые веса, основываясь не на их величине, а на величине соответствующих активаций. В рамках данного подхода выполняется оптимальное поканальное масштабирование наиболее значимых весов, при этом алгоритм не использует обратное распространение ошибки или реконструкцию, что позволяет избежать переобучения модели на калибровочном наборе данных. Несмотря на применение иного подхода, алгоритм AWQ во многом близок методам GPTQ и EXL2, хотя работает немного медленнее. Он поддерживается основными движками для инференса, включая TGI, vLLM и TensorRT-LLM.

В последние годы все бóльшую популярность приобретают методы экстремального квантования, при которых модели представляются в формате с 1- или 2-битной точностью. Хотя некоторые форматы, например EXL2, допускают столь радикальное уменьшение точности, это обычно приводит к серьезной

потере качества. Однако алгоритмы QuIP# и HQQ ориентированы на работу в таком режиме и гораздо лучше сохраняют уровень производительности исходных моделей. Это особенно актуально для моделей большого размера (свыше 30 млрд параметров), которые могут занимать меньше места, чем модели с 7 или 13 млрд параметров, и при этом генерировать более качественные ответы.

Скорее всего, данная тенденция сохранится, что будет способствовать дальнейшему развитию методов экстремального квантования.

В табл. 8.1 перечислены рассмотренные ранее методы, поддерживаемые тремя основными движками для инференса.

Таблица 8.1. Методы, поддерживаемые в системах TGI, vLLM и TensorRT-LLM

Метод	TGI	vLLM	TensorRT-LLM
Непрерывная пакетная обработка	✓	✓	✓
Спекулятивное декодирование	✓		
FlashAttention-2	✓	✓	✓
PagedAttention	✓	✓	✓
Конвейерный параллелизм			✓
Тензорный параллелизм	✓	✓	✓
GPTQ	✓		✓
EXL2	✓		
AWQ	✓	✓	✓

Резюме

Оптимизация инференса — важный этап эффективного развертывания LLM. В этой главе были рассмотрены такие техники, как оптимизация методов генерации, модельный параллелизм и квантование весов. Как было показано, существенного ускорения можно достичь за счет применения параллельного предсказания нескольких токенов с помощью спекулятивного декодирования или использования оптимизированного механизма внимания в рамках подхода FlashAttention-2. Мы также обсудили такие методы реализации модельного параллелизма, как параллелизм данных, конвейерный и тензорный параллелизм, которые позволяют распределить вычислительную нагрузку между несколькими GPU, увеличив пропускную способность и сократив время задержки. Методы квантования весов GGUF и EXL2 помогают дополнительно уменьшить

объем потребляемой памяти и ускорить процесс инференса без существенного ухудшения качества ответов модели.

Понимание и умение применять эти стратегии оптимизации являются ключевыми условиями для достижения высокой производительности при практическом использовании больших языковых моделей, например, в чат-ботах или системах автодополнения кода. Выбор конкретных техник и инструментов зависит от характеристик имеющегося оборудования, допустимой задержки и целевой пропускной способности. Комбинирование различных подходов вроде непрерывной пакетной обработки, спекулятивного декодирования, усовершенствованных механизмов внимания и модельного параллелизма позволяет адаптировать стратегии развертывания для достижения максимальной эффективности.

В главе 4 мы обсудили процесс реализации пайплайна загрузки данных — одного из компонентов стандартного RAG-приложения. В следующей главе мы завершим построение RAG-системы, создав пайплайны поиска и генерации и интегрировав их в пайплайн инференса.

Список используемых источников

- Hugging Face, Text Generation Inference. https://github.com/huggingface/text-generation-inference.
- *Kwon W., Li Z., Zhuang S., Sheng Y., Zheng L., Yu C. H., Gonzalez J. E., Zhang H., Stoica I.* Efficient Memory Management for Large Language Model Serving with PagedAttention.
- Nvidia, TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM.
- *Leviathan Y., Kalman M., Matias Y.* Fast Inference from Transformers via Speculative Decoding.
- *Cai T., Li Y., Geng Z., Peng H., Lee J. D., Chen D., Dao T.* Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads.
- *Aminabadi R. Y., Rajbhandari S., Zhang M., Awan A. A., Li C., Li D., Zheng E., Rasley J., Smith S., Ruwase O., He Y.* DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale.
- *Huang Y., Cheng Y., Bapna A., Firat O., Chen M. X., Chen D., Lee H., Ngiam J., Le Q. V., Wu Y., Chen Z.* GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism.
- *Reed K. J.* PiPPy: Pipeline Parallelism for PyTorch, https://github.com/pytorch/PiPPy.
- *Shoeybi M., Patwary M., Puri R., LeGresley P., Casper J., Catanzaro B.* Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism.

- *Verma, Vaidya*. Mastering LLM Techniques: Inference Optimization // NVIDIA Developer Technical Blog. https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/.
- *Dettmers T., Lewis M., Belkada Y., Zettlemoyer L.* LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale.
- *Gerganov G.* llama.cpp. https://github.com/ggerganov/llama.cpp.
- *Frantar E., Ashkboos S., Hoefler T., Alistarh D.* GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers.
- Tuboderp, exllamav2. https://github.com/turboderp/exllamav2.
- *Lin J., Tang J., Tang H., Yang S., Chen W.-M., Wang W.-C., Xiao G., Dang X., Gan C., Han S.* AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration.

9 RAG-пайплайн инференса

В главе 4 мы реализовали RAG-пайплайн признаков, с помощью которого извлекали данные из хранилища, очищали их, фрагментировали, векторизовали документы и загружали их в векторную *базу данных*. Таким образом, на текущем этапе векторная БД уже заполнена и готова к использованию для выполнения *генерации, дополненной поиском (RAG)*.

Методология RAG предполагает разделение архитектуры программы на три компонента: модуль поиска (извлечения) контекста, модуль дополнения промпта и модуль генерации ответа. Мы воспользуемся аналогичным подходом и реализуем модуль поиска, который позволяет выполнять запросы к векторной БД. В рамках данного модуля будут применяться продвинутые техники RAG для оптимизации процесса поиска. При этом, чтобы не усложнять архитектуру системы, мы не станем создавать отдельный модуль для дополнения промпта, а сделаем сервис для инференса, который будет принимать пользовательский запрос и релевантный контекст, формировать промпт и вызывать LLM для генерации ответа. Таким образом, мы реализуем два Python-модуля: один для поиска, другой — для вызова LLM и передачи ей пользовательского ввода и контекста. Объединив их, мы получим полноценный RAG-пайплайн инференса.

В главах 5 и 6 мы тонко настроили модель LLM-двойника, а в главе 8 обсудили способы оптимизации ее работы на этапе инференса. Следовательно, теперь наша модель готова к развертыванию в продакшене. Осталось лишь реализовать два указанных выше модуля.

В следующей главе мы сосредоточимся на развертывании тонко настроенной модели LLM-двойника на платформе AWS SageMaker, которая будет выступать в качестве конечной точки инференса. Таким образом, в текущей главе основной акцент делается на использовании продвинутых техник RAG для реализации модуля поиска. Мы посвятили целую главу обсуждению этапа поиска, поскольку основной объем кода RAG-системы, отвечающего за инференс, пишется именно для этого процесса, а не для вызова LLM. На этапе поиска необходимо грамотно обработать данные, чтобы извлечь из векторной БД наиболее

релевантные фрагменты. Поэтому большая часть продвинутой RAG-логики сосредоточена именно здесь.

К концу этой главы вы научитесь реализовывать модуль поиска для RAG-системы, дополнять промпт извлеченным контекстом и вызывать LLM для генерации финального ответа. В итоге вы сможете собрать полноценный RAG-пайплайн инференса, готовый к использованию в продакшене.

Устройство RAG-пайплайна инференса LLM-двойника

Прежде чем приступить к реализации RAG-пайплайна инференса, нам необходимо обсудить его архитектуру и используемые в нем продвинутые техники. На рис. 9.1 показана общая схема работы этого пайплайна. Сначала он получает на вход запрос пользователя, затем извлекает контекст с помощью поискового модуля и, наконец, вызывает LLM-сервис, развернутый на платформе AWS SageMaker, для генерации итогового ответа.

Пайплайн признаков и поисковый модуль, представленные на рис. 9.1, работают независимо друг от друга. Пайплайн признаков запускается на отдельной машине по расписанию и отвечает за наполнение векторной БД. Модуль поиска, напротив, вызывается по требованию при поступлении очередного пользовательского запроса в рамках работы пайплайна инференса.

Благодаря разделению ответственности между этими двумя компонентами векторная БД постоянно обновляется, а модуль поиска обращается к актуальным признакам при обработке каждого запроса. Входными данными для поискового модуля служит пользовательский запрос, на основе которого из векторной БД извлекаются наиболее релевантные элементы, используемые LLM при генерации финального ответа.

Чтобы лучше понять принцип работы RAG-пайплайна инференса, давайте последовательно рассмотрим этапы, представленные на рис. 9.1.

1. *Пользовательский запрос.* Все начинается с получения запроса от пользователя, например: «Напиши статью о...»
2. *Расширение запроса.* На этом этапе исходный запрос расширяется до нескольких вариантов, отражающих различные аспекты или интерпретации исходной формулировки. Использование *xN* запросов вместо одного повышает вероятность извлечения исчерпывающего набора релевантных данных. Этот шаг особенно важен в случае получения слишком узких или расплывчатых запросов.

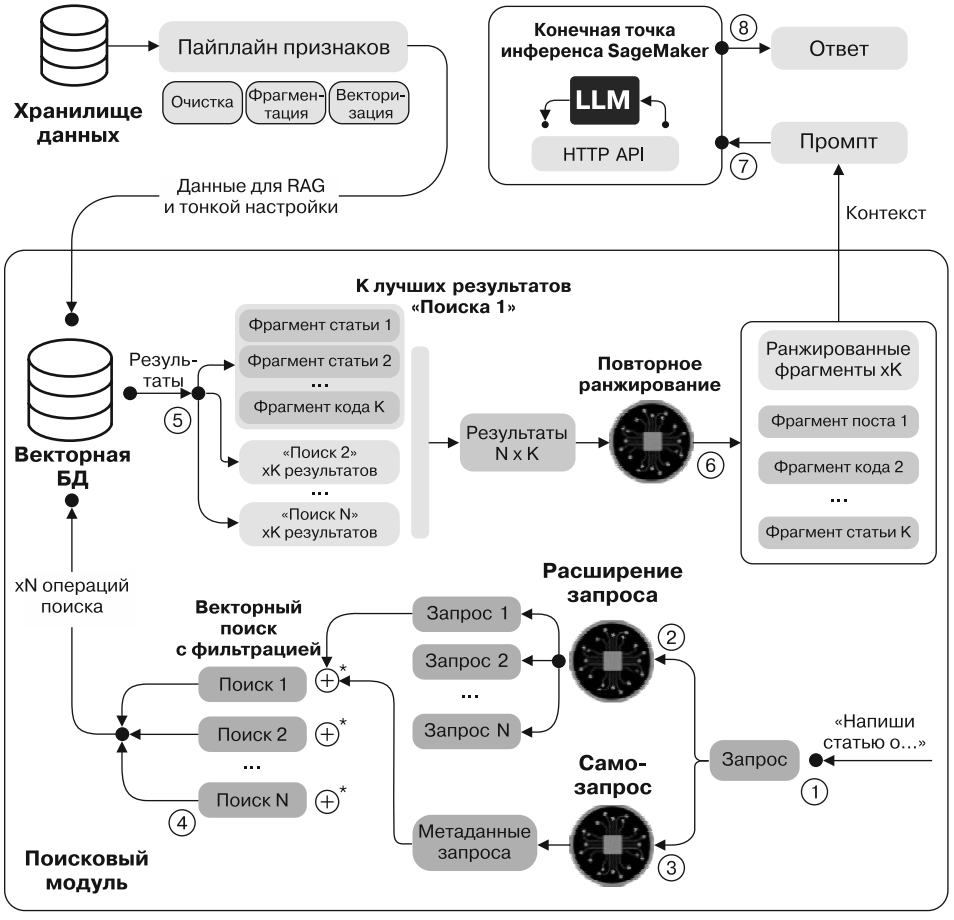


Рис. 9.1. Архитектура RAG-пайплайна инференса

3. *Реализация самозапроса (self-querying).* Из исходного запроса извлекаются полезные метаданные, например имя автора. Затем они используются в качестве фильтров при выполнении векторного поиска, что позволяет ускорить этот процесс и исключить из рассмотрения нерелевантные элементы.
4. *Векторный поиск с фильтрацией.* Здесь запросы векторизуются, и для каждого из них выполняется поиск ближайших K элементов. Число запросов равно N, поэтому выполняется xN отдельных операций поиска. Поскольку при этом применяются фильтры, основанные на метаданных, этот этап называется векторным поиском с фильтрацией.
5. *Сбор результатов.* Для каждого расширенного запроса извлекаются до xK ближайших фрагментов. Затем результаты xN операций поиска агрегируются в единый список, содержащий $N \times K$ фрагментов статей, постов и репозито-

риев, охватывающих широкий спектр возможных интерпретаций исходного запроса.

6. *Повторное ранжирование*. Чтобы сократить общий список из $N \times K$ фрагментов до xK наиболее релевантных, его необходимо еще раз отфильтровать. Для этого используется алгоритм повторного ранжирования, который оценивает каждый фрагмент по степени релевантности исходному запросу. Для вычисления оценки применяется кросс-энкодер, возвращающий значение от 0 до 1, где 1 соответствует полной релевантности. Затем результаты сортируются по убыванию этой оценки и формируется финальный список из K лучших фрагментов.
7. *Формирование промпта и вызов LLM*. Отобранное количество K фрагментов преобразуются в строку, которая используется при формировании финального промпта. Данный промпт создается с применением шаблона, извлеченного контекста и пользовательского запроса. Затем этот дополненный промпт передается LLM, развернутой на платформе AWS SageMaker и доступной через API-интерфейс.
8. *Предоставление ответа*. После обработки промпта LLM возвращает сгенерированный ответ. На этом этапе логика RAG завершает свою работу и пользователь получает результат.

На этом обзор архитектуры RAG-пайплайна инференса завершен.

Продвинутые техники RAG, используемые в системе LLM-двойника

Теперь, когда мы разобрались с принципом работы RAG-пайплайна инференса, давайте рассмотрим продвинутые техники RAG, применяемые в модуле поиска:

- *этап подготовки к поиску (pre-retrieval)* — расширение запроса и реализация самозапроса;
- *этап поиска (retrieval)* — векторный поиск с фильтрацией;
- *этап работы с найденными документами (post-retrieval)* — повторное ранжирование.

Перед обсуждением этих техник стоит перечислить интерфейсы Python, используемые в текущем разделе. Их исходный код доступен по адресу https://github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/llm_engineering/application/rag/base.py.

Первым интерфейсом является фабрика шаблонов промптов, стандартизирующая создание экземпляров этих шаблонов. Данный интерфейс наследует от класса `ABC` и предоставляет метод `create_template()`, возвращающий экземпляр `PromptTemplate` библиотеки LangChain. Хотя мы стремимся реализовать всю логику самостоятельно, чтобы лучше понять внутренние механизмы работы,

будем использовать некоторые классы вроде `PromptTemplate` — они помогают ускорить разработку, не скрывая при этом важную функциональность:

```
from abc import ABC, abstractmethod

from langchain.prompts import PromptTemplate
from pydantic import BaseModel

class PromptTemplateFactory(ABC, BaseModel):
    @abstractmethod
    def create_template(self) -> PromptTemplate:
        pass
```

Нам также следует определить интерфейс `RAGStep`, позволяющий стандартизировать реализацию таких продвинутых этапов RAG, как расширение запроса и самозапрос. Поскольку эти этапы часто зависят от других LLM, данный интерфейс предусматривает атрибут `mock`, предназначенный для снижения затрат и упрощения отладки в процессе разработки:

```
from typing import Any

from llm_engineering.domain.queries import Query

class RAGStep(ABC):
    def __init__(self, mock: bool = False) -> None:
        self._mock = mock

    @abstractmethod
    def generate(self, query: Query, *args, **kwargs) -> Any:
        pass
```

Наконец, нам нужно понять процесс моделирования доменной сущности `Query`, которая инкапсулирует пользовательский запрос с метаданными, необходимыми для применения продвинутых техник RAG. Ее реализация начинается с импорта классов:

```
from pydantic import UUID4, Field

from llm_engineering.domain.base import VectorBaseDocument
from llm_engineering.domain.types import DataCategory
```

Затем определяется класс `Query`, наследующий от `VectorBaseDocument` — *OVM-класса*, описанного в главе 4. Благодаря этому каждый запрос можно легко сохранять и извлекать из векторной БД:

```
class Query(VectorBaseDocument):
    content: str
    author_id: UUID4 | None = None
    author_full_name: str | None = None
    metadata: dict = Field(default_factory=dict)

class Config:
    category = DataCategory.QUERIES
```

Обратите внимание на атрибуты класса, объединяющие пользовательский запрос с метаданными:

- `content` — строка с исходным текстом запроса;
- `author_id` — необязательный идентификатор UUID4, извлекаемый из запроса и используемый в качестве фильтра при векторном поиске (для выборки фрагментов, написанных конкретным автором);
- `author_full_name` — необязательное строковое значение, связанное с атрибутом `author_id`;
- `metadata` — словарь для дополнительных метаданных (по умолчанию пустой).

Помимо доменного класса, мы также определяем метод класса `from_str()`, создающий экземпляр `Query` непосредственно из строки. Это позволяет стандартизировать процесс очистки запроса (например, удаление пробелов и символов перевода строки), выполняемой перед построением объекта `query`:

```
    @classmethod
    def from_str(cls, query: str) -> "Запрос":
        return Query(content=query.strip("\n "))
```

Дополнительно реализован метод экземпляра `replace_content()`, создающий версию объекта `query` с обновленным содержанием, но с сохранением всех остальных атрибутов: `id`, `author_id`, `author_full_name` и `metadata`.

```
    def replace_content(self, new_content: str) -> "Запрос":
        return Query(
            id=self.id,
            content=new_content,
            author_id=self.author_id,
            author_full_name=self.author_full_name,
            metadata=self.metadata,
        )
```

Этот метод особенно полезен в случаях, когда необходимо изменить текст запроса (например, при предварительной обработке или нормализации) без потери сопутствующих метаданных или идентификаторов. После класса `Query` определяется класс `EmbeddedQuery`:

```
class EmbeddedQuery(Query):
    embedding: list[float]

class Config:
    category = DataCategory.QUERIES
```

Он расширяет класс `Query`, добавляя поле `embedding`. Сущность `EmbeddedQuery` объединяет все данные и метаданные, необходимые для выполнения векторного поиска в системе Qdrant или другой векторной БД.

Теперь, когда мы познакомились с интерфейсами и новыми доменными сущностями, используемыми в RAG-пайплайне инференса, давайте перейдем к рассмотрению продвинутых техник оптимизации этапа подготовки к поиску.

Оптимизация этапа подготовки к поиску: расширение запроса и самозапрос

Для оптимизации данного этапа мы применили два метода: расширение запроса и самозапрос. Они тесно связаны с этапом векторного поиска с фильтрацией, о котором пойдет речь в следующем разделе. Давайте сначала изучим код для расширения запроса, а затем перейдем к реализации самозапроса.

В рамках этих двух методов мы задействовали API OpenAI для генерации вариантов исходного запроса на этапе его расширения и для извлечения метаданных при реализации самозапроса. Во всех примерах книги использовалась модель GPT-4o-mini, однако, учитывая быстрое обновление моделей OpenAI, ее поддержка может быть прекращена. Это не проблема, поскольку у вас всегда есть возможность указать нужную модель в файле `.env`, скорректировав значение переменной окружения `OPENAI_MODEL_ID`.

Расширение запроса

Типичная проблема этапа поиска состоит в том, что векторная БД опрашивается с помощью единственного векторного представления исходного запроса, что позволяет покрыть лишь небольшую область пространства эмбеддингов. Если в эмбеддинге не отражены все необходимые детали и нюансы запроса, извлеченный контекст может оказаться нерелевантным. В результате мы рискуем упустить важные документы, которые семантически связаны с запросом, но удалены от его вектора.

Расширение запроса позволяет преодолеть это ограничение. Используя LLM для генерации нескольких вариантов исходного запроса, вы получаете разные формулировки, охватывающие различные его аспекты. При векторизации эти расширенные запросы занимают другие векторы, что повышает вероятность извлечения релевантных документов из векторной БД.

Расширить запрос довольно просто — достаточно составить детализированный промпт, который направит работу LLM в процессе генерации альтернативных формулировок. В результате вместо одного запроса вы получаете xN запросов, позволяющих выполнить такое же количество операций поиска релевантного контекста.

Однако увеличение количества операций поиска влияет на длительность задержки. Поэтому вам следует поэкспериментировать с числом генерируемых запросов, чтобы соблюсти требования, предъявляемые к производительности

системы. Этот процесс также можно оптимизировать и существенно сократить время задержки путем распараллеливания операций поиска. Именно такой подход мы реализуем в классе `ContextRetriever` в конце главы.



Расширение запросов основано на тех же принципах, что и использование множественных запросов или мультизапросов (multi-query). Пример реализации данного подхода под названием MultiQueryRetriever можно найти в библиотеке LangChain: https://python.langchain.com/docs/how_to/MultiQueryRetriever.

Теперь перейдем к коду. Начнем с импорта модулей и классов, используемых для расширения запроса:

```
from langchain_openai import ChatOpenAI

from llm_engineering.domain.queries import Query
from llm_engineering.settings import settings

from .base import RAGStep
from .prompt_templates import QueryExpansionTemplate
```

Далее мы определяем класс `QueryExpansion`, отвечающий за генерацию расширенных версий запроса. Его реализацию можно найти по адресу https://github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/llm_engineering/application/rag/query_expanison.py:

```
class QueryExpansion(RAGStep):
    def generate(self, query: Query, expand_to_n: int) -> list[Query]:
        assert expand_to_n > 0, f"'expand_to_n' should be greater than
        ➥0.Got {expand_to_n}."

        if self._mock:
            return [query for _ in range(expand_to_n)]
```

В методе `generate` мы сначала проверяем, что количество запрашиваемых вариантов (параметр `expand_to_n`) превышает ноль. Если экземпляр класса находится в режиме имитации (`self._mock` установлен равным `True`), `generate` просто возвращает список, состоящий из копий исходного запроса, то есть симулирует расширение без фактического обращения к API. В обычном режиме выполняется формирование промпта и инициализация языковой модели:

```
    query_expansion_template = QueryExpansionTemplate()
    prompt = query_expansion_template.create_template(expand_to_n - 1)
    model = ChatOpenAI(model=settings.OPENAI_MODEL_ID, api_key=settings.
    ➥OPENAI_API_KEY, temperature=0)
```

На этом этапе создается объект `QueryExpansionTemplate` и промпт для генерации `expand_to_n - 1` новых запросов (исключая оригинал). Далее инициализируется модель `ChatOpenAI` с заданными параметрами; для получения детерминированного результата параметр `temperature` устанавливается равным `0`. Затем

мы создаем цепочку LangChain, объединяя промпт с моделью, и вызываем ее, передав пользовательский вопрос:

```
chain = prompt | model

response = chain.invoke({"question": query})
result = response.content
```

Используя оператор пайплайна (`prompt | model`), мы создаем цепочку, которая при вызове генерирует расширенные формулировки исходного запроса. Ответ модели сохраняется в объекте `result`. После получения ответа выполняется парсинг и очистка расширенных версий запроса:

```
queries_content = result.strip().split(query_expansion_template.separator)

queries = [query]
queries += [
    query.replace_content(stripped_content)
    for content in queries_content
    if (stripped_content := content.strip())
]

return queries
```

Полученный результат разбивается на отдельные строки с помощью разделителя, заданного в шаблоне, для извлечения индивидуальных запросов. Начав со списка, содержащего исходный запрос, мы поочередно добавляем к нему каждую расширенную версию, предварительно удалив лишние пробелы.

Наконец, мы определяем класс `QueryExpansionTemplate`, который отвечает за построение промпта, расширяющего запрос. Реализация этого класса, наряду с другими шаблонами промптов, находится здесь: https://github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/llm_engineering/application/rag/prompt_templates.py:

```
from langchain.prompts import PromptTemplate

from .base import PromptTemplateFactory

class QueryExpansionTemplate(PromptTemplateFactory):
    prompt: str = """Ты — интеллектуальный языковой помощник. Твоя задача —
    сгенерировать {expand_to_n} различных вариантов заданного пользователем
    вопроса, чтобы обеспечить извлечение релевантных документов из векторной
    базы данных. Создавая несколько формулировок одного и того же запроса, ты
    помогаешь пользователю преодолеть ограничения, присущие поиску на основе
    сходства по расстоянию. Представь альтернативные формулировки вопросов,
    разделяя их посредством '{separator}'. Исходный вопрос: {question}"""

    @property
    def separator(self) -> str:
        return "#next-question#"
```

```
    def create_template(self, expand_to_n: int) -> PromptTemplate:
        return PromptTemplate(
            template=self.prompt,
            input_variables=["question"],
            partial_variables={
                "separator": self.separator,
                "expand_to_n": expand_to_n,
            },
        )
```

Этот класс определяет промпт, который дает языковой модели инструкцию сгенерировать несколько формулировок исходного пользовательского запроса. Для гибкой настройки промпта используются такие заполнители, как `{expand_to_n}`, `{separator}` и `{question}`.

Параметр `expand_to_n` определяет, какое количество запросов необходимо сгенерировать при построении экземпляра `PromptTemplate`. Свойство `separator` задает уникальную строку, с помощью которой можно удобно разделить полученные варианты запросов. Переменные `expand_to_n` и `separator` передаются как `partial_variables`, что делает их неизменяемыми во время выполнения. В то же время заполнитель `{question}` будет обновляться при каждом вызове цепочки LLM.

Теперь, когда мы разобрались с расширением запроса, рассмотрим пример использования класса `QueryExpansion`. Выполним следующий код с помощью команды `python -m llm_engineering.application.rag.query_expansion`:

```
query = Query.from_str("Напиши статью о наиболее эффективных типах современных
➥методов RAG.")
    query_expander = QueryExpansion()
    expanded_queries = query_expander.generate(query, expand_to_n=3)
    for expanded_query in expanded_queries:
        logger.info(expanded_query.content)
```

В результате у нас появилось несколько вариантов исходного запроса. Как видите, метод расширения запроса успешно справился с задачей, добавив некоторые детали и предложив альтернативные формулировки, отражающие разные аспекты исходного запроса, в частности сделав акцент на самых эффективных техниках RAG и предоставив их общий обзор (напомним, что оригинальный запрос указан первым):

```
2024-09-18 17:51:33.529 | INFO - Напиши статью о наиболее эффективных
типах современных методов RAG.
2024-09-18 17:51:33.529 | INFO - Какие передовые методы RAG считаются
наиболее результативными и как их можно применять на практике?
2024-09-18 17:51:33.529 | INFO - Можешь ли ты привести обзор ведущих
техник RAG?
```

Теперь рассмотрим следующий метод оптимизации этапа подготовки к поиску на основе самозапроса.

Самозапрос

Проблема, возникающая при векторизации запроса, заключается в невозможности гарантировать, что все значимые для вашей задачи детали будут достаточно представлены в эмбеддинге. Например, вы хотите иметь абсолютную уверенность, что результаты поиска будут выполняться с учетом тегов[1], указанных пользователем во входных данных. Однако вы не можете контролировать степень выраженности того или иного тега в эмбеддинге. Векторизация одного лишь текста запроса не гарантирует того, что теги будут в достаточной мере представлены в эмбеддинге или обеспечат достаточно сильный сигнал при вычислении расстояния до других векторов.

Эта проблема касается и других метаданных, которые вы можете захотеть учесть при поиске, например идентификаторов, имен или категорий.

Решением служит метод самозапроса (self-querying) — техника, позволяющая извлечь из запроса теги или иные критически важные метаданные и применить их в качестве фильтров на этапе векторного поиска. Извлечение содержимого нужных полей метаданных, таких как теги, идентификаторы, число комментариев, лайков, репостов и т. д., осуществляется с помощью LLM, после чего вы получаете полный контроль над тем, как эти данные будут учитываться в процессе поиска. Например, в системе LLM-двойника из запроса извлекается имя автора и используется в качестве фильтра. Техника на основе самозапроса тесно связана с векторным поиском с фильтрацией, о котором мы поговорим в следующем подразделе.

Теперь перейдем к коду. Первым делом импортируем необходимые модули и классы:

```
from langchain_openai import ChatOpenAI

from llm_engineering.application import utils
from llm_engineering.domain.documents import UserDocument
from llm_engineering.domain.queries import Query
from llm_engineering.settings import settings

from .base import RAGStep
from .prompt_templates import SelfQueryTemplate
```

Далее мы определяем класс `SelfQuery`, наследующий от `RAGStep` и реализующий метод `generate()`. Исходный код этого класса доступен по адресу https://

[1] Теги — это метаданные, которые могут быть извлечены из запроса пользователя, например его имя. — *Примеч. науч ред.*

github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/llm_engineering/application/rag/self_query.py:

```
class SelfQuery(RAGStep):
    def generate(self, query: Query) -> Query:
        if self._mock:
            return query
```

В методе `generate()` мы проверяем, установлен ли атрибут `_mock` в значение `True`. Если да, то исходный объект запроса возвращается в неизмененном виде, что позволяет нам обойтись без вызова модели в процессе тестирования и отладки. В обычном режиме выполняется формирование промпта и инициализация языковой модели:

```
    prompt = SelfQueryTemplate().create_template()
    model = ChatOpenAI(model=settings.OPENAI_MODEL_ID, api_key=settings.
    ➥OPENAI_API_KEY, temperature=0)
```

Здесь мы создаем экземпляр промпта с помощью фабричного класса `SelfQueryTemplate`, а затем инициализируем экземпляр модели `ChatOpenAI` (как при расширении запроса). Далее мы объединяем промпт и модель в цепочку и вызываем ее, передав пользовательский запрос:

```
    chain = prompt | model

    response = chain.invoke({"question": query})
    user_full_name = response.content.strip("\n ")
```

Мы извлекаем содержимое из ответа LLM и удаляем пробелы в начале и в конце строки, чтобы получить значение `user_full_name`. Затем проверяем, удалось ли модели извлечь какую-либо информацию о пользователе.

```
    if user_full_name == "none":
        return query
```

Если значением `user_full_name` является `"none"`, значит, имя пользователя в запросе не обнаружено. В таком случае возвращается исходный объект запроса. Если имя обнаружено, оно разбивается на переменные `first_name` и `last_name` с помощью вспомогательной функции. Затем на основе этих данных происходит извлечение или создание экземпляра `UserDocument`:

```
    first_name, last_name = utils.split_user_full_name(user_full_name)
    user = UserDocument.get_or_create(first_name=first_name, last_name=last_name)
```

Наконец, объект запроса обновляется с учетом извлеченной информации об авторе и возвращается:

```
    query.author_id = user.id
    query.author_full_name = user.full_name

    return query
```

Теперь обновленный запрос содержит значения `author_id` и `author_full_name`, которые могут использоваться на следующих этапах работы пайплайна RAG.

Рассмотрим класс `SelfQueryTemplate`, определяющий промпт для извлечения информации о пользователе:

```
from langchain.prompts import PromptTemplate

from .base import PromptTemplateFactory

class SelfQueryTemplate(PromptTemplateFactory):
    prompt: str = """Ты — интеллектуальный языковой помощник. Твоя задача —
    извлечь информацию из пользовательского запроса. Необходимо определить имя
    пользователя или идентификатор. Твой ответ должен содержать только
    извлеченное имя пользователя (например, Иван Петров) или идентификатор
    (например, 1345256) — и больше ничего. Если в запросе нет ни имени
    пользователя, ни идентификатора, ты должен вернуть следующий токен: none.

    Примеры:
    ВОПРОС 1:
    Меня зовут Пол Юстин, и я хочу написать пост о...
    ОТВЕТ 1:
    Пол Юстин

    ВОПРОС 2:
    Я хочу написать пост о...
    ОТВЕТ 2:
    none

    ВОПРОС 3:
    Мой идентификатор пользователя 1345256, и я хочу написать пост о...
    ОТВЕТ 3:
    1345256

    Вопрос пользователя: {question}"""

    def create_template(self) -> PromptTemplate:
        return PromptTemplate(template=self.prompt,
        ➥input_variables=["question"])
```

В классе `SelfQueryTemplate` мы определяем промпт, который дает модели инструкцию извлечь из входного запроса имя пользователя или идентификатор. Для настройки поведения модели в разных ситуациях используется подход обучения на малом числе примеров (few-shot). При вызове шаблона заполнитель `{question}` заменяется на фактический пользовательский запрос.

Реализуя механизм самозапроса, мы гарантируем, что ключевые метаданные, необходимые для решения конкретной задачи, будут явно извлечены и использованы на этапе поиска. Такой подход позволяет преодолеть ограничения, связанные с тем, что семантика эмбеддинга не всегда отражает все аспекты запроса.

Теперь, когда класс `SelfQuery` реализован, давайте рассмотрим пример его использования. Для запуска следующего кода выполните команду `python -m llm_engineering.application.rag.self_query`:

```
query = Query.from_str("Я — Пол Юстин. Напиши статью о наиболее
➥эффективных типах современных методов RAG.")
self_query = SelfQuery()
query = self_query.generate(query)
logger.info(f"Extracted author_id: {query.author_id}")
logger.info(f"Extracted author_full_name: {query.author_full_name}")
```

В результате мы получим корректно извлеченные имя и идентификатор автора:

```
2024-09-18 18:02:10.362 | INFO - Extracted author_id: 900fec95-d621-4315-84c6-
52e5229e0b96
2024-09-18 18:02:10.362 | INFO - Extracted author_full_name: Пол Юстин
```

Теперь давайте объединим технику самозапроса и векторный поиск с фильтрацией для оптимизации этапа поиска.

Оптимизация этапа поиска: векторный поиск с фильтрацией

Векторный поиск играет ключевую роль при извлечении релевантной информации на основе семантического сходства. Однако применение его базовой версии может привести к серьезным проблемам, влияющим как на точность, так и на скорость извлечения данных. Это связано с тем, что такой поиск опирается исключительно на численную близость векторных представлений, не учитывая критически важный контекст или категории объектов поиска.

Одной из основных проблем обычного векторного поиска является извлечение семантически близких, но контекстуально нерелевантных документов. Поскольку эмбеддинги отражают общее семантическое значение, они могут присваивать высокие оценки сходства содержимому, которое совпадает по языковым паттернам или тематике, но не соответствует основной идее или ограничениям конкретного запроса. Например, в ответ на поисковый запрос Java алгоритм может выдать материалы как о языке программирования, так и об индонезийском острове, опираясь лишь на семантическое сходство, что приведет к неоднозначным или даже вводящим в заблуждение результатам.

Кроме того, по мере увеличения объема данных обычный алгоритм векторного поиска сталкивается с проблемами масштабируемости. При отсутствии фильтрации он вынужден вычислять степени сходства между запросом и всеми элементами векторного пространства, что существенно увеличивает время отклика.

Такая исчерпывающая проверка замедляет обработку запросов и требует больше вычислительных ресурсов, делая метод неэффективным при решении задач в реальном времени или работе с крупными наборами данных.

Векторный поиск с фильтрацией решает эту проблему, добавляя дополнительные критерии, вроде метаданных, тегов и категорий, перед сравнением векторов. Применяя фильтры, алгоритм сужает область поиска еще до вычисления результатов, соответствующих по контексту исходному запросу. Такой подход повышает точность поиска, исключая нерелевантные документы, которые в противном случае могли бы попасть в выборку на основании одного лишь семантического сходства.

Кроме того, векторный поиск с фильтрацией сокращает задержку за счет уменьшения записей в БД, среди которых надо проводить поиск. Работа с узким, но более релевантным подмножеством существенно снижает вычислительную нагрузку и ускоряет получение результата. Это особенно важно для приложений, требующих мгновенной реакции или обработки крупных запросов.

Поскольку метаданные, используемые для фильтрации, часто содержатся во входных данных пользователя, их необходимо извлечь до выполнения запроса к векторной БД. Именно это мы и сделали на этапе реализации самозапроса, когда извлекли имя автора для ограничения пространства поиска только его контентом. Таким образом, поскольку самозапрос выполняется до обращения к векторной базе, данная техника относится к оптимизации подготовительного этапа, в то время как векторный поиск с фильтрацией относится к оптимизации этапа поиска.

Например, при использовании системы Qdrant для добавления фильтра, проверяющего наличие заданного значения `author_id` в метаданных документа, необходимо реализовать следующий код:

```
from qdrant_client.models import FieldCondition, Filter, MatchValue

records = qdrant_connection.search(
            collection_name="articles",
            query_vector=query_embedding,
            limit=3,
            with_payload=True,
            query_filter= Filter(
                    must=[
                        FieldCondition(
                            key="author_id",
                            match=MatchValue(
                                value=str("1234"),
                            ),
                        )
                    ]
            ),
)
```

Таким образом, хотя обычный векторный поиск позволяет извлекать информацию на основе семантического сходства, он имеет ограничения, способные снижать производительность системы. Векторный поиск с фильтром по контексту решает данную проблему, сочетая преимущества векторных представлений с контекстной фильтрацией. Это помогает достичь большей точности и эффективности при извлечении информации в RAG-системах. Последней техникой оптимизации, которую мы рассмотрим, является повторное ранжирование.

Оптимизация этапа работы с найденными документами: повторное ранжирование

Одна из проблем, возникающих в RAG-системах, заключается в том, что извлеченный контекст может содержать нерелевантные фрагменты, которые:

- *добавляют шум* — полученный контекст может засорять информацию и сбивать языковую модель с толку;
- *увеличивают объем промпта* — включение лишних фрагментов приводит к увеличению длины промпта и росту затрат на его обработку. Кроме того, языковые модели, как правило, уделяют наибольшее внимание началу и концу контекста, поэтому при его избыточном объеме велик риск, что модель упустит суть;
- *не соответствуют запросу* — фрагменты информации извлекаются на основе сходства с векторами запроса. Однако модель, выполняющая векторизацию, может быть не адаптирована к конкретной задаче, в результате чего она присваивает высокие оценки сходства фрагментам, не имеющим прямого отношения к исходному запросу.

Решением этой проблемы является повторное ранжирование, то есть упорядочение всех извлеченных фрагментов (в количестве $N \times K$) по степени их релевантности исходному запросу: от наиболее подходящего к наименее. В данном случае N — это число операций поиска, выполненных после расширения запроса, а K — число фрагментов, извлекаемых в ходе каждой такой операции. В результате в общей сложности мы получаем $N \times K$ фрагментов.

В RAG-системах повторное ранжирование является ключевой техникой оптимизации этапа работы с найденными документами, позволяющей уточнить результаты, полученные от модели, отвечающей за извлечение фрагментов информации.

Для оценки релевантности каждого фрагмента исходному запросу применяется алгоритм повторного ранжирования, обычно использующий другую архитектуру. Эти модели обеспечивают более точную оценку семантической близости между запросом и фрагментом по сравнению с базовыми методами, использующими эмбеддинги и косинусное расстояние (см. раздел «Обзор продвинутой RAG-архитектуры» главы 4).

В итоге мы выбираем K наиболее релевантных фрагментов из отсортированного списка, содержащего $N \times K$ элементов, на основе оценки, присвоенной алгоритмом повторного ранжирования. Он особенно эффективен в сочетании с техникой *расширения запроса*. Однако сначала давайте рассмотрим, как данный алгоритм работает без применения расширения запроса.

1. *Извлечение более K фрагментов* для получения более обширного пула потенциально полезных данных.
2. *Повторное ранжирование* этого набора данных для оценки релевантности каждого из фрагментов исходному запросу.
3. *Выбор K лучших результатов* для их использования в качестве контекста в финальном промпте.

В сочетании с техникой расширения запроса повторное ранжирование позволяет собрать релевантный контекст из разных точек векторного пространства вместо того, чтобы сосредоточиваться только на одной его области. В таком случае процесс выглядит следующим образом.

1. *Извлечение $N \times K$ фрагментов* с помощью нескольких расширенных запросов.
2. *Повторное ранжирование* всех извлеченных фрагментов по степени релевантности.
3. *Выбор K лучших результатов* для применения в финальном промпте.

Интеграция этапа повторного ранжирования в пайплайн RAG повышает качество и релевантность извлеченного контекста, а также оптимизирует использование вычислительных ресурсов. Теперь реализуем этот этап в нашей системе LLM-двойника, чтобы лучше понять то, что описано выше. (Исходный код доступен по адресу https://github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/llm_engineering/application/rag/reranking.py.)

Мы начинаем с импорта необходимых модулей и классов:

```
from llm_engineering.application.networks import CrossEncoderModelSingleton
from llm_engineering.domain.embedded_chunks import EmbeddedChunk
from llm_engineering.domain.queries import Query

from .base import RAGStep
```

Затем мы определяем класс `Reranker`, отвечающий за упорядочение извлеченных документов по степени их релевантности запросу:

```
class Reranker(RAGStep):
    def __init__(self, mock: bool = False) -> None:
        super().__init__(mock=mock)

        self._model = CrossEncoderModelSingleton()
```

В инициализаторе класса `Reranker` создается экземпляр `CrossEncoderModelSingleton` — модель, которая будет использоваться для оценки степени релевантности каждого фрагмента документа исходному запросу.

Основная логика класса `Reranker` реализована в методе `generate()`:

```
def generate(self, query: Query, chunks: list[EmbeddedChunk],
➥keep_top_k: int) -> list[EmbeddedChunk]:
    if self._mock:
        return chunks

    query_doc_tuples = [(query.content, chunk.content) for chunk in chunks]
    scores = self._model(query_doc_tuples)

    scored_query_doc_tuples = list(zip(scores, chunks, strict=False))
    scored_query_doc_tuples.sort(key=lambda x: x[0], reverse=True)

    reranked_documents = scored_query_doc_tuples[:keep_top_k]
    reranked_documents = [doc for _, doc in reranked_documents]

    return reranked_documents
```

Метод `generate()` принимает запрос, список фрагментов (частей документов) и количество фрагментов, которые необходимо сохранить (`keep_top_k`). В режиме имитации этот метод просто возвращает исходные фрагменты. В обычном режиме он выполняет следующие шаги.

1. Формирует пары, состоящие из текста запроса и содержимого каждого фрагмента.
2. С помощью кросс-энкодера каждой паре присваивается оценка, отражающая степень соответствия фрагмента запросу.
3. Оценки объединяются с соответствующими фрагментами в список кортежей.
4. Этот список сортируется по убыванию оценок.
5. Из отсортированного списка выбираются `keep_top_k` лучших фрагментов.
6. Из полученных кортежей извлекаются фрагменты, которые возвращаются в качестве результата повторного ранжирования.

Перед определением класса `CrossEncoderModelSingleton` мы импортируем необходимые компоненты:

```
from sentence_transformers.cross_encoder import CrossEncoder

from .base import SingletonMeta
```

Из библиотеки `sentence_transformers` импортируется класс `CrossEncoder`, предоставляющий функциональность для оценки пар текстов. Кроме того, из базового модуля импортируется метакласс `SingletonMeta`, обеспечивающий реализацию паттерна Одиночка, то есть гарантирующий существование

только одного экземпляра модели в нашем приложении. Теперь определим класс `CrossEncoderModelSingleton`:

```
class CrossEncoderModelSingleton(metaclass=SingletonMeta):
    def __init__(
        self,
        model_id: str = settings.RERANKING_CROSS_ENCODER_MODEL_ID,
        device: str = settings.RAG_MODEL_DEVICE,
    ) -> None:
        """

        A singleton class that provides a pre-trained cross-encoder model
        ➥for scoring pairs of input text.
        """

        self._model_id = model_id
        self._device = device

        self._model = CrossEncoder(
            model_name=self._model_id,
            device=self._device,
        )
        self._model.model.eval()
```

Этот класс инициализирует модель, используя значения `model_id` и `device`, заданные в глобальных настройках, загружаемых из файла `.env`. После инициализации модель переводится в режим оценки (`evaluation mode`) с помощью `self._model.model.eval()`, что гарантирует ее готовность к инференсу.

Класс `CrossEncoderModelSingleton` реализует специальный метод `__call__`, предназначенный для оценки пар текстов:

```
    def __call__(self, pairs: list[tuple[str, str]], to_list: bool = True)
    -> NDArray[np.float32] | list[float]:
        scores = self._model.predict(pairs)

        if to_list:
            scores = scores.tolist()

        return scores
```

Метод `__call__` позволяет передавать модели списки пар (`pairs`), каждая из которых состоит из запроса и фрагмента документа, и получать оценки их релевантности. Для вычисления этих оценок используется функция модели `predict()`.

Таким образом, `CrossEncoderModelSingleton` представляет собой обертку над классом `CrossEncoder`, предназначенную для двух целей. Во-первых, она реализует паттерн Одиночка, позволяющий нам получить доступ к единственному экземпляру модели из любого места в приложении без повторной загрузки этой модели в память. Во-вторых, данная обертка определяет интерфейс для нашей

модели (или любых других моделей, отвечающих за повторное ранжирование). Такой подход делает код устойчивым к будущим изменениям. Если мы решим использовать другую реализацию или стратегию повторного ранжирования, например через API, достаточно будет написать обертку с той же структурой интерфейса и заменить старый класс новым, не затрагивая остальной код.

Итак, вы познакомились со всеми продвинутыми техниками RAG, применяемыми в нашей архитектуре. В следующем разделе мы рассмотрим класс `ContextRetriever`, объединяющий все описанные методы, и объясним, как использовать поисковый модуль совместно с LLM в рамках полноценного RAG-пайплайна инференса.

Реализация RAG-пайплайна инференса для LLM-двойника

Как было сказано в начале главы, RAG-пайплайн инференса можно условно разделить на три основных компонента: модуль поиска, модуль создания промпта и модуль генерации ответа. В этом разделе мы сосредоточимся на реализации поискового модуля, который будет содержать основную часть логики и кода. После этого мы рассмотрим процесс построения итогового промпта на основе извлеченного контекста и вопроса пользователя.

В заключение объединим поисковый модуль, логику построения промпта и языковую модель в полноценный RAG-пайплайн инференса. Однако протестировать работу модели мы сможем только после завершения главы 10, посвященной развертыванию нашего LLM-двойника на платформе AWS SageMaker.

Итак, приступим к реализации поискового модуля.

Реализация поискового модуля

Для начала рассмотрим реализацию класса `ContextRetriever`, который управляет этапом поиска в нашей RAG-системе, применяя все описанные ранее техники, такие как расширение запроса, самозапрос, повторное ранжирование и векторный поиск с фильтрацией. Исходный код этого класса доступен на GitHub по адресу https://github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/llm_engineering/application/rag/retriever.py.

Точкой входа в `ContextRetriever` является метод `search()`, который связывает воедино все рассмотренные в этой главе шаги, необходимые для нахождения результатов, схожих с пользовательским запросом. На рис. 9.2 показано, как извлеченные на этапе самозапроса сведения об авторе используются в рамках векторного поиска с фильтрацией. Кроме того, на схеме подробно представлена сама операция поиска, в ходе которой для каждого запроса выполняется три

обращения к векторной базе данных для извлечения статей, постов и репозиториев, релевантных исходному запросу. При выполнении каждой из N операций поиска мы стремимся извлечь не более K результатов. Поскольку у нас есть три категории данных, мы извлекаем не более $K/3$ фрагментов, относящихся к каждой из них. Таким образом, формируется список длиной $\leq K$ частей. Он также может содержать менее K элементов, если после фильтрации по имени автора в одной или нескольких категориях окажется недостаточно подходящих фрагментов.

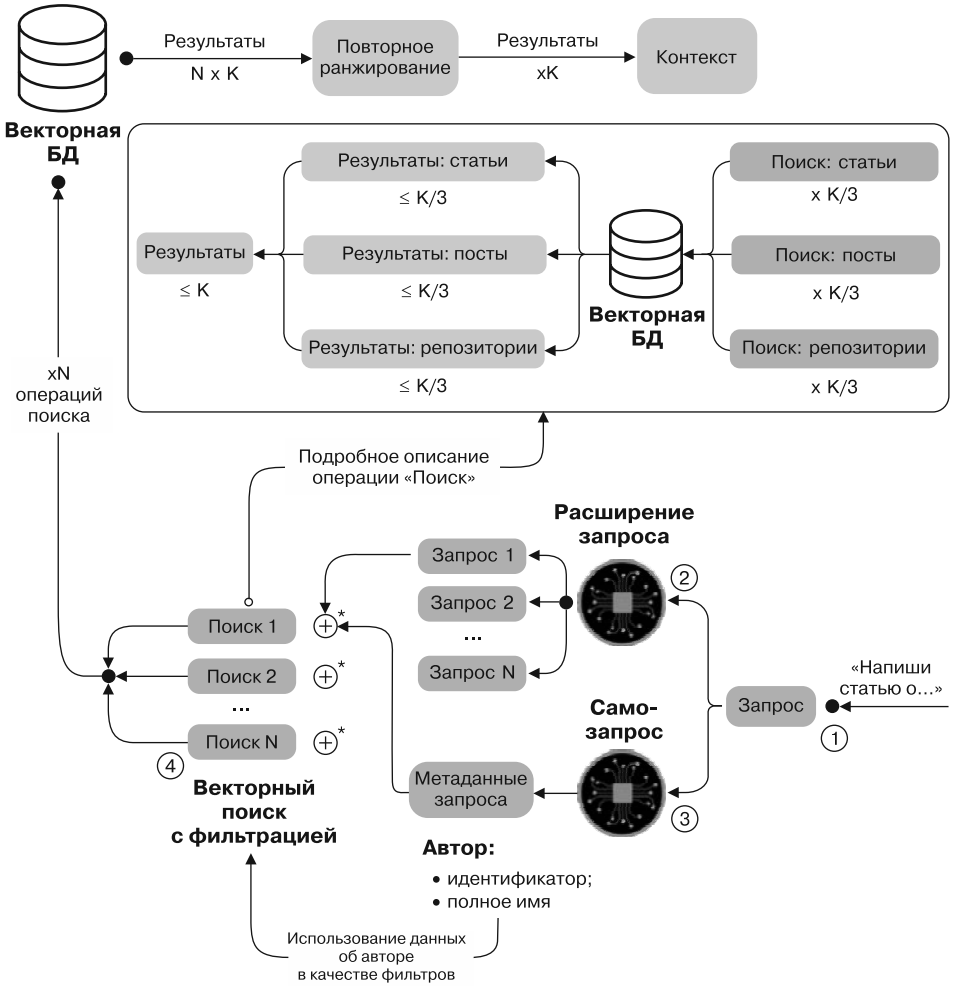


Рис. 9.2. Процесс работы поискового модуля RAG-системы

На рис. 9.3 показан процесс обработки результатов *xN* операций поиска. Поскольку каждая из них возвращает не более *K* элементов, общее число извлеченных фрагментов не будет превышать $N \times K$. Все они объединяются в общий список, после чего из него удаляются дубликаты, чтобы гарантировать уникальность каждого фрагмента. Наконец, результаты передаются в модель, отвечающую за повторное ранжирование, сортируются по убыванию оценки релевантности и *K наиболее релевантных* фрагментов отбираются для использования в качестве контекста в процессе генерации, дополненной поиском.

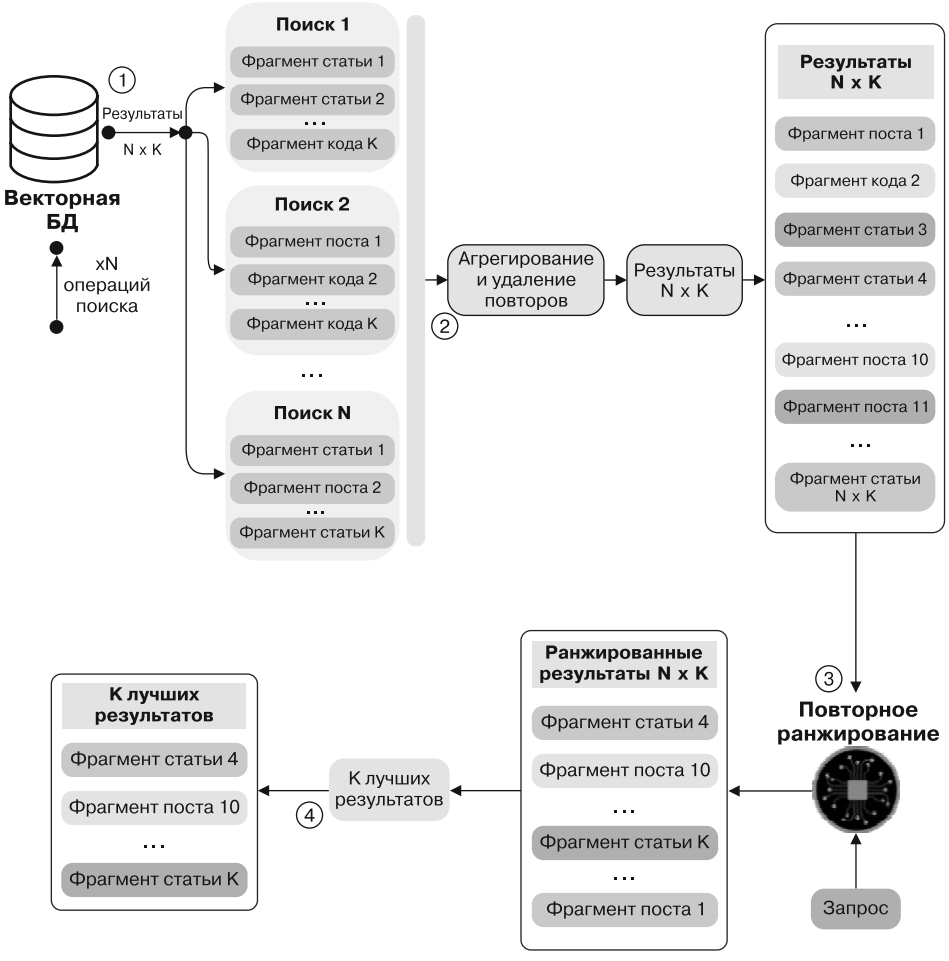


Рис. 9.3. Обработка результатов в поисковом модуле RAG-системы

Теперь рассмотрим, как этапы, показанные на рис. 9.2 и 9.3, реализованы внутри класса `ContextRetriever`. В первую очередь мы инициализируем этот класс, создавая экземпляры `QueryExpansion`, `SelfQuery` и `Reranker`:

```
class ContextRetriever:
    def __init__(self, mock: bool = False) -> None:
        self._query_expander = QueryExpansion(mock=mock)
        self._metadata_extractor = SelfQuery(mock=mock)
        self._reranker = Reranker(mock=mock)
```

В методе `search()` строка пользовательского запроса преобразуется в объект `query`. Затем с помощью экземпляра `SelfQuery` из него извлекаются параметры `author_id` и `author_full_name`:

```
    def search(
        self,
        query: str,
        k: int = 3,
        expand_to_n_queries: int = 3,
    ) -> list:
        query_model = Query.from_str(query)

        query_model = self._metadata_extractor.generate(query_model)
        logger.info(
            "Successfully extracted the author_id from the query.",
            author_id=query_model.author_id,
        )
```

Далее при помощи экземпляра `QueryExpansion` запрос расширяется до набора семантически близких вариантов:

```
        n_generated_queries = self._query_expander.generate(query_model,
        ➥expand_to_n=expand_to_n_queries)
        logger.info(
            "Successfully generated queries for search.",
            num_queries=len(n_generated_queries),
        )
```

После этого выполняется конкурентный поиск по всем расширенным запросам с использованием пула потоков. Каждый запрос обрабатывается методом `_search()`, который мы рассмотрим чуть позже. Полученные результаты проходят несколько этапов обработки: сглаживание, удаление повторов и сбор в единый список.

```
        with concurrent.futures.ThreadPoolExecutor() as executor:
            search_tasks = [executor.submit(self._search, _query_model, k)
            ➥for _query_model in n_generated_queries]

            n_k_documents = [task.result() for task in
            ➥concurrent.futures.as_completed (search_tasks)]
```

```
        n_k_documents = utils.misc.flatten(n_k_documents)
        n_k_documents = list(set(n_k_documents))

        logger.info("All documents retrieved successfully.",
        ➥num_documents=len(n_k_documents))
```

На следующем этапе извлеченные документы повторно ранжируются по степени релевантности исходному запросу и сохраняются только k лучших фрагментов:

```
    if len(n_k_documents) > 0:
        k_documents = self.rerank(query, chunks=n_k_documents,
        ➥keep_top_k=k)
    else:
        k_documents = []

    return k_documents
```

Метод `_search()` выполняет векторный поиск с фильтрацией по различным категориям данных, таким как посты, статьи и репозитории. Для этого используется `EmbeddingDispatcher`, который преобразует запрос в объект `EmbeddedQuery`, содержащий эмбеддинг запроса и извлеченные метаданные:

```
def _search(self, query: Query, k: int = 3) -> list[EmbeddedChunk]:
    assert k >= 3, "k should be >= 3"

    def _search_data_category(
        data_category_odm: type[EmbeddedChunk], embedded_query:
        ➥EmbeddedQuery
    ) -> list[EmbeddedChunk]:
        if embedded_query.author_id:
            query_filter = Filter(
                must=[
                    FieldCondition(
                        key="author_id",
                        match=MatchValue(
                            value=str(embedded_query.author_id),
                        ),
                    )
                ]
            )
        else:
            query_filter = None

        return data_category_odm.search(
            query_vector=embedded_query.embedding,
            limit=k // 3,
            query_filter=query_filter,
        )

    embedded_query: EmbeddedQuery = EmbeddingDispatcher.dispatch(query)
```

Для векторизации запроса мы используем тот же класс `EmbeddingDispatcher`, который применялся в RAG-пайплайне признаков для векторизации фрагментов документов, хранящихся в векторной БД. Использование одного и того же класса гарантирует, что при загрузке данных и при обработке запроса задействуется одна и та же модель векторизации, что критически важно для корректной работы этапа поиска.

Операция поиска выполняется отдельно для каждой категории данных с помощью вспомогательной функции `_search_data_category()`. Внутри нее применяются фильтры, извлеченные из объекта `embedded_query`. Например, если в этом объекте присутствует значение `author_id`, то поиск выполняется только по документам соответствующего автора. Результаты из всех категорий затем объединяются в общий список:

```
        post_chunks = _search_data_category(EmbeddedPostChunk, embedded_query)
        articles_chunks = _search_data_category(EmbeddedArticleChunk,
        ➥embedded_query)
        repositories_chunks = _search_data_category(EmbeddedRepositoryChunk,
        ➥embedded_query)
        retrieved_chunks = post_chunks + articles_chunks +
        ➥repositories_chunks

        return retrieved_chunks
```

Далее вызывается метод `rerank()`, который упорядочивает полученные документы на основе их релевантности исходному запросу:

```
    def rerank(self, query: str | Query, chunks: list[EmbeddedChunk],
    ➥keep_top_k: int) -> list[EmbeddedChunk]:
        if isinstance(query, str):
            query = Query.from_str(query)

        reranked_documents = self._reranker.generate(query=query,
        ➥chunks=chunks, keep_top_k=keep_top_k)
        logger.info("Documents reranked successfully.",
        ➥num_documents=len(reranked_documents))

        return reranked_documents
```

Класс `ContextRetriever` позволяет извлекать релевантный контекст для любого запроса с помощью всего нескольких строк кода. Например, рассмотрим следующий фрагмент кода, в котором простой вызов метода `search()` вызывает всю продвинутую RAG-архитектуру:

```
from loguru import logger

from llm_engineering.application.rag.retriever import ContextRetriever

query = """
        Меня зовут Пол Юстин.
```

```
        Не мог бы ты написать пост в LinkedIn о системах RAG?
        Меня особенно интересует :
            - как работает RAG
            - как он интегрируется с векторными базами данных и большими
              ➥языковыми моделями.
        """

retriever = ContextRetriever(mock=False)
documents = retriever.search(query, k=3)

logger.info("Retrieved documents:")
for rank, document in enumerate(documents):
    logger.info(f"{rank + 1}: {document}")
```

Вызов приведенного выше кода осуществляется с помощью консольной команды `poetry poe call-rag-retrieval module`. Результат ее выполнения выглядит так:

```
2024-09-18 19:01:50.588 | INFO – Retrieved documents:
2024-09-18 19:01:50.588 | INFO – 1: id=UUID('541d6c22-d15a-4e6a-924a-
68b7b1e0a330') content='4 продвинутых алгоритма RAG, которые вы должны знать,
автор Пол Юстин. Внедрите 4 продвинутых метода поиска RAG для оптимизации
запросов в векторной базе данных. Интегрируйте модуль поиска RAG в промышленную
LLM-систему ..." platform='decodingml.substack.com' document_id=UUID('32648f33-
87e6-435c-b2d7-861a03e72392') author_id=UUID('900fec95-d621-4315-84c6-
52e5229e0b96') author_full_name='Пол Юстин' metadata={'embedding_model_id':
'sentence-transformers/all-MiniLM-L6-v2', 'embedding_size': 384, 'max_input_
length': 256} link='https://decodingml.substack.com/p/the-4-advanced-rag-
algorithms-you?r=1ttoeh'
2024-09-18 19:01:50.588 | INFO – 2: id=UUID('5ce78438-1314-4874-8a5a-
04f5fcf0cb21') content='Обзор продвинутых методов оптимизации RAGПромышленная
RAG-система делится на 3 основных компонента: ingestion (очистка, чанкинг,
эмбеддинг и загрузка данных в векторную БД), retrieval (запрос к векторной
БД для ..." platform='medium' document_id=UUID('bd9021c9-a693-46da-97e7-
0d06760ee6bf') author_id=UUID('900fec95-d621-4315-84c6-52e5229e0b96')
author_full_name='Пол Юстин' metadata={'embedding_model_id': 'sentence-
transformers/all-MiniLM-L6-v2', 'embedding_size': 384, 'max_input_length': 256}
link='https://medium.com/decodingml/the-4-advanced-rag-algorithms-you-must-know-
to-implement-5d0c7f1199d2'
2024-09-18 19:02:45.729 | INFO – 3: id=UUID('0405a5da-4686-428a-91ca-
446b8e0446ff') content='Каждая статья на Medium будет отдельным уроком. Сквозной
фреймворк для промышленных LLM-систем через создание вашего LLM-двойника.
Важность пайплайнов данных в эпоху генеративного ИИ. Change Data Capture (CDC),
обеспечивающий событийно-ориентированную ..." platform='medium' document_
id=UUID('bd9021c9-a693-46da-97e7-0d06760ee6bf') author_id=UUID('900fec95-d621-
4315-84c6-52e5229e0b96') author_full_name='Пол Юстин' metadata={'embedding_
model_id': 'sentence-transformers/all-MiniLM-L6-v2', 'embedding_size': 384,
'max_input_length': 256} link='https://medium.com/decodingml/the-4-advanced-rag-
algorithms-you-must-know-to-implement-5d0c7f1199d2'
```

Как видите, результат содержит не только извлеченный контент, но и разнообразные метаданные, такие как имя использованной модели векторизации или ссылка на исходный документ. Эти данные можно легко добавить в список

источников при формировании окончательного ответа, что повышает доверие пользователя к полученному результату.

Теперь, когда мы разобрались с реализацией поискового модуля, давайте рассмотрим устройство полноценного RAG-пайплайна инференса.

Реализация RAG-пайплайна инференса

На последнем этапе процесса реализации RAG-системы нам нужно сформировать промпт, используя контекст, полученный от поискового модуля, и вызвать языковую модель для генерации ответа. В данном разделе мы рассмотрим эти шаги и объединим их в функцию `rag()`. Исходный код всех описанных здесь функций можно найти на GitHub по адресу https://github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/llm_engineering/infrastructure/inference_pipeline_api.py.

Начнем с функции `call_llm_service()`, которая отвечает за взаимодействие с LLM. Она принимает пользовательский запрос и контекст, настраивает путь вызова языковой модели, выполняет инференс и возвращает сгенерированный ответ. При этом контекст предоставлять не обязательно, к модели можно обратиться и без него:

```
def call_llm_service(query: str, context: str | None) -> str:
    llm = LLMInferenceSagemakerEndpoint(
        endpoint_name=settings.SAGEMAKER_ENDPOINT_INFERENCE,
        ➥inference_component_name=None
    )
    answer = InferenceExecutor(llm, query, context).execute()

    return answer
```

Эта функция отправляет HTTP-запрос к тонко настроенной модели LLM-двойника, размещенной на платформе AWS SageMaker. Подробности, связанные с SageMaker, мы разберем в следующей главе, где будет описана работа классов `LLMInferenceSagemakerEndpoint` и `InferenceExecutor`. Сейчас важно понять, что именно эта функция используется для вызова нашей тонко настроенной модели. Особое внимание следует уделить тому, как запрос и контекст, передаваемые в класс `InferenceExecutor`, преобразуются в итоговый промпт. Это осуществляется с помощью простого шаблона, в который подставляются пользовательский запрос и извлеченный контекст:

```
prompt = f"""
Ты — создатель контента. Напиши то, о чем тебя попросил пользователь,
используя предоставленный контекст в качестве основного источника информации
для контента.
Пользовательский запрос: {query}
Контекст: {context}
          """
```

Теперь перейдем к функции `rag()` — здесь объединяются все элементы логики RAG. Она отвечает за извлечение релевантных запросу документов, преобразование этих документов в контекст, включаемый в промпт, и получение финального ответа от LLM:

```
def rag(query: str) -> str:
    retriever = ContextRetriever(mock=False)
    documents = retriever.search(query, k=3)
    context = EmbeddedChunk.to_context(documents)

    answer = call_llm_service(query, context)

    return answer
```

Поскольку все этапы работы RAG-системы были реализованы в виде отдельных классов, итоговая функция `rag()` занимает всего пять строк кода, инкапсулируя при этом всю сложность системы, подобно тому как это сделано в библиотеках вроде LangChain, LlamaIndex или Haystack. Однако в отличие от готовых решений мы разобрали, как построить продвинутый RAG-сервис с нуля. Благодаря четко разделенным обязанностям каждого класса их можно использовать как блоки LEGO. Например, мы можем вызывать LLM без всякого контекста или задействовать поисковый модуль в качестве механизма обработки запросов поверх векторной БД. В следующей главе вы увидите работу функции `rag()` в действии после развертывания тонко настроенной модели на платформе AWS SageMaker.

Прежде чем завершить главу, обсудим улучшения, которые можно внести в RAG-пайплайн инференса. Поскольку мы создаем чат-бота, имеет смысл реализовать возможность для хранения в памяти всех предыдущих пользовательских запросов и сгенерированных ответов. Это позволит модели опираться не только на текущий промпт, но и на всю историю взаимодействий. Поскольку история сообщений со временем увеличивается, существует риск превышения допустимой длины контекстного окна или увеличения затрат на обработку запроса. В связи с этим необходимо реализовать механизм сокращения размера памяти. Как показано на рис. 9.4, самым простым решением является сохранение только последних *K* сообщений из диалога. К сожалению, при таком подходе модель никогда не будет обладать полной информацией о всей беседе.

Альтернативный подход предполагает сохранение краткого содержания всей предыдущей беседы, помимо *K* последних реплик. Для составления такого резюме существует множество способов, обсуждение которых выходит за рамки книги. Однако самый простой подход заключается в том, чтобы при каждом новом пользовательском запросе обновлять краткое содержание диалога и на его основе генерировать ответ.

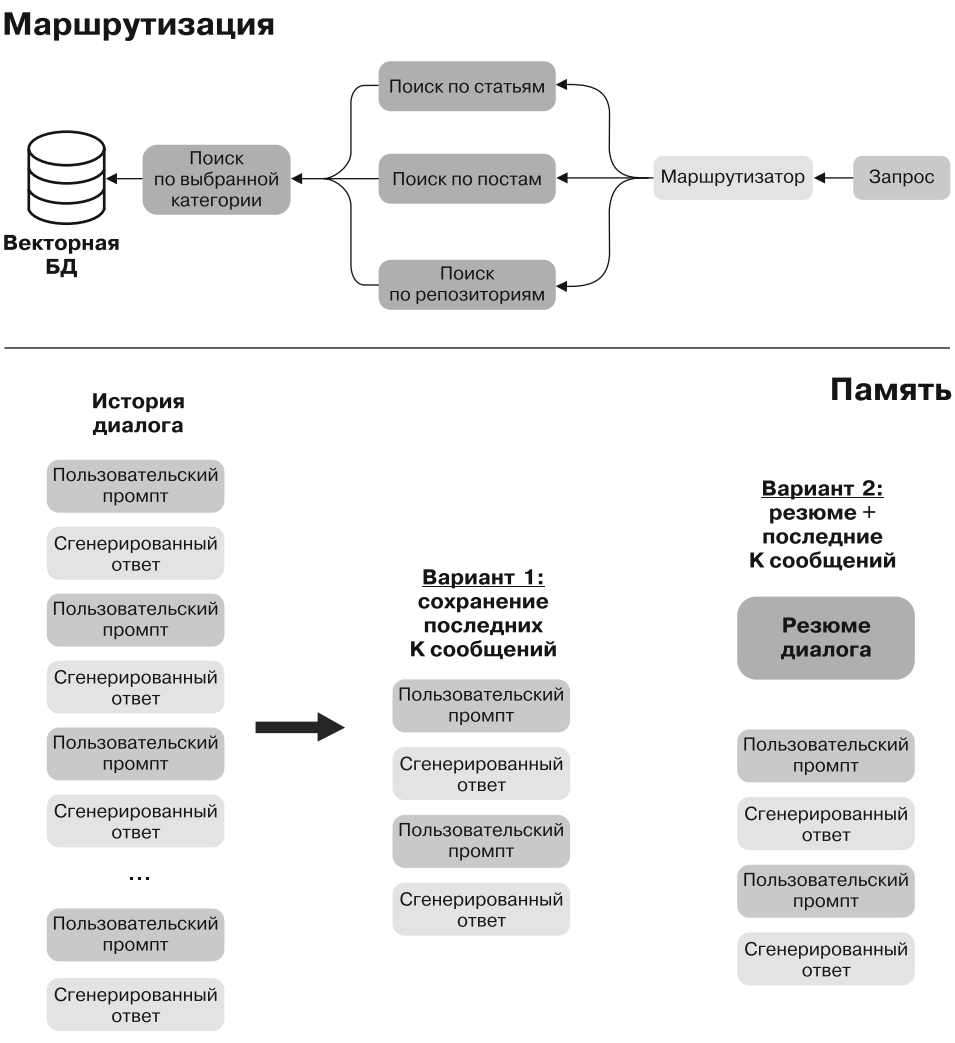


Рис. 9.4. Маршрутизация и использование памяти

Как уже было сказано, в ходе каждой операции поиска мы отправляем в векторную базу данных три запроса — по одному для каждой категории данных. Поэтому второе возможное улучшение заключается в добавлении маршрутизатора между запросом и поиском. В этой роли может выступать классификатор, определяющий, какие категории данных необходимо задействовать при обработке конкретного запроса. Таким образом, вместо выполнения трех запросов в ходе каждой операции поиска мы часто сможем ограничиться одним или двумя. Например, если пользователь хочет написать теоретическую статью о технологии RAG, то имеет смысл выполнить поиск только по коллекции статей.

В этом случае маршрутизатор предскажет категорию `articles` и при обработке запроса будет задействована соответствующая коллекция.

Если же пользователю требуется пример кода, иллюстрирующий процесс построения пайплайна RAG, то маршрутизатор должен будет выбрать категории `articles` и `repositories`, поскольку для формирования исчерпывающего контекста необходимо извлечь примеры из обеих коллекций.

Обычно маршрутизатор определяет, какую модель следует использовать для обработки конкретного запроса (например, GPT-4 или локально размещенную Llama 3.1). Однако в нашем случае алгоритм маршрутизации можно адаптировать для оптимизации этапа поиска.

Мы можем еще больше повысить эффективность процесса поиска, применив гибридный алгоритм, сочетающий векторный поиск (на основе эмбеддингов) с поиском по ключевым словам, например BM25[1]. До появления векторных методов такие алгоритмы, как BM25, активно использовались для поиска схожих документов в базе данных. Комбинируя вышеуказанные подходы, гибридный алгоритм поиска позволяет находить как документы с точными вхождениями терминов вроде *RAG*, *LLM* или *SageMaker*, так и семантически близкие тексты, что повышает точность и релевантность результатов. В основе алгоритма гибридного поиска лежат следующие механизмы.

1. *Параллельная обработка*. Поисковый запрос одновременно обрабатывается алгоритмами векторного поиска и поиска по ключевым словам (например, BM25). Каждый из них возвращает набор релевантных документов, отобранных по их собственным критериям.
2. *Нормализация оценок*. Результаты обоих поисков получают числовые оценки релевантности, которые затем нормализуются, чтобы их можно было корректно сравнивать. Этот шаг необходим, поскольку алгоритм векторного поиска и BM25 используют разные шкалы, поэтому выставляемые ими оценки нельзя сравнить или объединить без приведения к одному масштабу.
3. *Объединение результатов*. Нормализованные оценки совмещаются (как правило, методом взвешенной суммы) для формирования итогового рейтинга документов. Настройка весов позволяет задать степень влияния результатов семантического поиска и поиска по ключевым словам на итоговую оценку.

Таким образом, сочетая семантический поиск с поиском по ключевым словам, мы можем повысить точность работы поискового модуля RAG-системы. Алгоритм векторного поиска распознает синонимы и родственные понятия, благодаря чему релевантная информация не теряется из-за различий в формулировках.

---

[1] Алгоритм Best Match 25 оценивает, насколько часто слова из запроса встречаются в документе и насколько эти слова редки в коллекции (редкие слова дают больший вес). — *Примеч. науч ред.*

В свою очередь, алгоритм поиска по ключевым словам позволяет выделить документы, содержащие важные термины, что особенно важно в технических областях со строго определенной терминологией.

Наконец, еще одно улучшение, которое можно внести в нашу RAG-систему, — использование многоиндексных векторных структур вместо индексации, основанной только на эмбеддингах содержимого. Многоиндексный подход (multi-indexing — «мультииндексация») позволяет комбинировать эмбеддинги нескольких полей, вместо того чтобы выполнять векторный поиск по эмбеддингам только одного поля.

Например, в случае с LLM-двойником мы использовали только поле `content` статей, постов и репозиториев для выполнения запросов к векторной БД. Применяя стратегию мультииндексации, можно дополнительно индексировать эмбеддинги других полей, например содержащих название платформы, на которой размещен контент, или дату его публикации. Это может повлиять на точность поиска, поскольку разные платформы содержат разные типы контента, а более актуальный материал обычно оказывается более релевантным. Фреймворки вроде Superlinked существенно упрощают реализацию многоиндексного подхода. Ниже показан пример кода, в котором с помощью Superlinked мы всего в несколько строк реализовали индексацию по содержимому и названию платформы для нашей коллекции статей:

```
from superlinked.framework.common.schema.id_schema_object import IdField
from superlinked.framework.common.schema.schema import schema
from superlinked.framework.common.schema.schema_object import String
... # Другие инструкции импорта из Superlinked.

@schema
class ArticleSchema:
    id: IdField
    platform: String
    content: String

article = ArticleSchema()

articles_space_content = TextSimilaritySpace(
    text=chunk(article.content, chunk_size=500, chunk_overlap=50),
    model=settings.EMBEDDING_MODEL_ID,
)

articles_space_plaform = CategoricalSimilaritySpace(
    category_input=article.platform,
    categories=["medium", "substack", "wordpress"],
    negative_filter=-5.0,
)

article_index = Index(
    [articles_space_content, articles_space_plaform],
    fields=[article.author_id],
)
```

Superlinked — это мощный инструмент Python для решения задач, связанных с векторными вычислениями, включая RAG, рекомендательные системы и семантический поиск. Он предоставляет целую экосистему, позволяющую быстро загружать данные в векторную БД, формулировать сложные запросы и разворачивать готовый сервис в качестве RESTful API.

Мир языковых моделей и RAG — экспериментальная среда, как и другие области ИИ. Поэтому при создании реальных продуктов важно быстро построить работающее решение, даже если оно будет далеко от идеала. Затем его можно постепенно улучшать на основе результатов экспериментов и пользовательской обратной связи. Такой подход является стандартом в индустрии: он позволяет итеративно развивать продукт, предоставляя бизнес-ценность на самых ранних этапах его жизненного цикла.

Резюме

В этой главе вы узнали, как построить продвинутый RAG-пайплайн инференса. Сначала мы обсудили архитектуру программных компонентов RAG-системы. Затем мы подробно разобрали используемые в поисковом модуле техники, включая расширение запроса, самозапрос, векторный поиск с фильтрацией и повторное ранжирование. Далее мы рассмотрели реализацию модульного класса `ContextRetriever`, объединяющего все эти компоненты под единым интерфейсом, что значительно упрощает процесс поиска релевантных документов. Наконец, мы связали воедино этапы поиска, дополнения промпта и вызова LLM внутри функции `rag()`, которая и станет нашим RAG-пайплайном инференса.

Как уже было сказано, мы не можем протестировать тонко настроенную модель до ее развертывания на платформе AWS SageMaker. В следующей главе вы узнаете, как это сделать, а также мы создадим интерфейс для вызова конечной точки инференса и реализуем веб-сервер FastAPI.

Список используемых источников

- A real-time retrieval system for social media data | VectorHub by SuperLinked. https://superlinked.com/vectorhub/articles/real-time-retrieval-system-social-media-data.
- Building a Router from Scratch — LlamaIndex. https://docs.llamaindex.ai/en/stable/examples/low_level/router/.
- How to add memory to chatbots | LangChain. https://python.langchain.com/docs/how_to/chatbots_memory/#summary-memory.
- How to do “self-querying” retrieval | LangChain. https://python.langchain.com/docs/how_to/self_query/.

- How to route between sub-chains | LangChain. https://python.langchain.com/docs/how_to/routing/#routing-by-semantic-similarity.
- How to use the MultiQueryRetriever | LangChain. https://python.langchain.com/docs/how_to/MultiQueryRetriever/.
- Hybrid Search explained. https://weaviate.io/blog/hybrid-search-explained.
- *Iusztin P.* 4 Advanced RAG Algorithms You Must Know | Decoding ML. Medium. https://medium.com/decodingml/the-4-advanced-rag-algorithms-you-must-know-to-implement-5d0c7f1199d2.
- *Monigatti L.* Advanced Retrieval-Augmented Generation: From Theory to LlamaIndex Implementation. Medium. https://towardsdatascience.com/advanced-retrieval-augmented-generation-from-theory-to-llamaindex-implementation-4de1464a9930.
- Multi-attribute search with vector embeddings | VectorHub by Superlinked. https://superlinked.com/vectorhub/articles/multi-attribute-semantic-search.
- Optimizing RAG with Hybrid Search & Reranking | VectorHub by Superlinked. https://superlinked.com/vectorhub/articles/optimizing-rag-with-hybrid-search-reranking.
- Refactoring.Guru. Singleton. https://refactoring.guru/design-patterns/singleton.
- *Stoll M.* Visualize your RAG Data — Evaluate your Retrieval-Augmented Generation System with Ragas. Medium. https://towardsdatascience.com/visualize-your-rag-data-evaluate-your-retrieval-augmented-generation-system-with-ragas-fc2486308557.
- Using LLM's for retrieval and reranking — LlamaIndex, data framework for LLM applications. https://www.llamaindex.ai/blog/using-llms-for-retrieval-and-reranking-23cf2d3a14b6.

10 Развертывание пайплайна инференса

Развертывание пайплайна инференса для приложения на основе LLM — это критически важный этап жизненного цикла ML-системы. Именно на этом этапе модели начинают приносить практическую пользу, поскольку становятся доступными конечным пользователям. Однако успешное развертывание ИИ-моделей сопряжено с определенными трудностями, так как для выполнения инференса требуются большие вычислительные мощности и доступ к актуальным признакам. Чтобы преодолеть эти ограничения и добиться целевых показателей длительности задержки, пропускной способности и стоимости, крайне важно тщательно продумать стратегию развертывания. При работе с LLM необходимо учитывать описанные в главе 8 методы оптимизации инференса, такие как квантование модели. Кроме того, для автоматизации процессов развертывания следует опираться на лучшие практики MLOps, включая использование реестров моделей, позволяющих версионировать и распространять модели по всей инфраструктуре.

Чтобы понять, как спроектировать архитектуру развертывания LLM-двойника, мы рассмотрим такие стратегии, как онлайн-инференс в реальном времени, асинхронный инференс и пакетное преобразование в офлайн-режиме. Затем обозначим основные критерии, которые необходимо учитывать перед принятием архитектурного решения, в частности задержку, пропускную способность, характеристики данных и инфраструктуры. Мы также сравним преимущества и недостатки монолитной и микросервисной архитектуры в контексте обслуживания модели, поскольку этот выбор существенно влияет на масштабируемость и удобство сопровождения всей системы. Разобравшись с архитектурными решениями, мы сосредоточимся на стратегии развертывания пайплайна инференса для LLM-двойника. Затем пошагово разберем весь процесс организации работы данного сервиса, включая развертывание тонко настроенной модели на конечной точке AWS SageMaker и реализацию сервера FastAPI, выступающего в роли основного интерфейса для взаимодействия с пользователями. В конце главы мы кратко обсудим стратегии автоматического масштабирования и способы их применения на платформе SageMaker.

Критерии выбора стратегии развертывания пайплайна инференса

При развертывании ML-моделей важно знать, что к пропускной способности, длительности задержки, данным и инфраструктуре любого ML-приложения предъявляются определенные требования.

Понимание этих аспектов и их взаимосвязей имеет решающее значение. При проектировании архитектуры развертывания всегда приходится искать баланс между ними, поскольку они напрямую влияют на пользовательский опыт. Например, что важнее для вашей модели — минимальная задержка или высокая пропускная способность?

Пропускная способность и задержка

Пропускная способность определяет количество запросов, которые система может обработать за единицу времени. Обычно она измеряется в *запросах в секунду (requests per second, RPS)*. Этот показатель особенно важен, если ML-модель должна обрабатывать большое количество запросов. Он показывает, насколько эффективно система справляется с нагрузкой. Для обеспечения высокой пропускной способности требуется масштабируемая и надежная инфраструктура, например машины или кластеры с несколькими мощными GPU.

Задержка — это время, необходимое системе для обработки одного запроса на инференс: от момента его получения до возвращения результата. Длительность задержки имеет решающее значение в приложениях, работающих в режиме реального времени, от которых требуется мгновенный отклик. В частности, это касается приложений, взаимодействующих с пользователем в интерактивном режиме, систем обнаружения мошенничества и любых других сервисов, которые должны давать немедленную обратную связь. Например, средняя задержка API OpenAI определяется как среднее время между отправкой запроса пользователем и моментом, когда результат его обработки становится доступным в приложении.

Длительность задержки складывается из временных затрат на выполнение операций сетевого ввода/вывода, сериализацию и десериализацию, а также на выполнение инференса самой моделью. Пропускная способность, в свою очередь, отражает среднее количество запросов, которое API способен обработать и обслужить за одну секунду.

Для создания системы с низкой задержкой требуется высокоэффективная и дорогостоящая инфраструктура: быстрые процессоры, скоростные сети, а иногда и организация вычислений на клиенте, уменьшающих потребность в передаче данных.

При эффективной параллельной обработке нескольких запросов сокращение длительности задержки может привести к увеличению пропускной способности. Например, если система обрабатывает запрос за 100 мс, значит, ее пропускная способность составляет 10 запросов в секунду. Если задержка уменьшается до 10 мс, пропускная способность возрастает до 100 запросов в секунду.

Однако ситуация усложняется тем, что большинство ML-приложений использует стратегию пакетной обработки (batching), при которой модели передается сразу несколько запросов. В таких случаях сокращение задержки может привести к снижению пропускной способности и, наоборот, увеличение задержки может повысить пропускную способность. Например, если пакет из 20 запросов обрабатывается за 100 мс, задержка составляет 100 мс, а пропускная способность — 200 запросов в секунду. Если 60 запросов обрабатываются за 200 мс, то задержка составляет 200 мс, а пропускная способность возрастает до 300 запросов в секунду. Следовательно, даже при пакетной обработке запросов на этапе инференса необходимо учитывать минимально допустимую задержку, обеспечивающую комфортное взаимодействие пользователя с системой.

Данные

Как известно, данные — это фундамент любой ML-системы. Однако когда речь идет о выдаче модели, нас прежде всего интересуют входные и выходные данные, а именно их формат, объем и сложность. Данные лежат в основе процесса инференса, а их характеристики, в частности размер и тип, определяют требования к конфигурации системы и способу ее оптимизации для эффективной обработки запросов.

Тип и размер данных напрямую влияют на задержку и пропускную способность: чем объемнее или сложнее данные, тем больше времени требуется на их обработку. Например, проектирование модели, которая принимает структурированные данные и возвращает значение вероятности, принципиально отличается от проектирования LLM, получающей на вход текст (или даже изображения) и выдающей массив символов.

Инфраструктура

Под инфраструктурой понимается совокупность аппаратного и программного обеспечения, сетевых компонентов и системной архитектуры, поддерживающих развертывание и работу ML-моделей. Инфраструктура предоставляет ресурсы, необходимые для размещения, масштабирования и сопровождения моделей, к которым относятся вычислительные ресурсы, память, хранилища, сетевые компоненты и программная среда.

- Для обеспечения *высокой пропускной способности* система должна обладать масштабируемой инфраструктурой, способной обрабатывать большие

объемы данных и справляться с высокими нагрузками, например, путем организации параллельной обработки, использования распределенных систем и мощных GPU.

- Чтобы добиться *низкой задержки*, инфраструктуру необходимо оптимизировать для ускорения обработки запросов, например задействуя более быстрые CPU, GPU или специализированное оборудование. Однако при такой оптимизации в условиях пакетной обработки запросов приходится жертвовать пропускной способностью. Это приводит к тому, что оборудование используется не на полную мощность: система обрабатывает меньше запросов в секунду, простаивая часть времени. В итоге растут затраты на обработку одного запроса. Таким образом, выбор подходящей конфигурации оборудования критически важен для оптимизации затрат.

Инфраструктуру необходимо проектировать с учетом специфики данных. Это включает выбор хранилищ, способных обслуживать большие наборы данных, и реализацию механизмов быстрого извлечения, обеспечивающих эффективный доступ к данным. Например, при офлайн-обучении модели нас, как правило, интересует высокая пропускная способность, а при онлайн-инференсе — низкая задержка.

Учитывая вышесказанное, перед выбором конкретной стратегии развертывания необходимо задать себе следующие вопросы.

- Каковы требования к пропускной способности (важно учесть минимальные, средние и максимальные значения)?
- Сколько запросов система должна обрабатывать одновременно (1, 10, 1000, 1 000 000)?
- Какая задержка считается допустимой (1 мс, 10 мс, 1 с)?
- Как должна масштабироваться система (по уровню загрузки CPU, числу запросов, длине очереди, размеру данных или их комбинации)?
- Каковы допустимые затраты? С какими типами данных мы работаем (изображения, тексты, таблицы)?
- Каков размер обрабатываемых данных (100 Мбайт, 1 Гбайт, 10 Гбайт)?

Ответы на эти вопросы напрямую влияют на качество пользовательского опыта, а значит, и на успешность продукта. Даже если ваша модель показывает выдающиеся результаты, слишком долгое ожидание ответа или частые сбои могут привести к тому, что пользователь выберет более надежную, пусть и менее точную альтернативу. Например, исследование, проведенное компанией Google в 2016 году, показало, что 53 % пользователей мобильных устройств покидают сайт, если его загрузка занимает более трех секунд: https://www.thinkwithgoogle.com/consumer-insights/consumer-trends/mobile-site-load-time-statistics.

Теперь давайте рассмотрим три стратегии развертывания пайплайна инференса.

Стратегии развертывания пайплайна инференса

На рис. 10.1 представлены три стратегии развертывания пайплайна инференса:

- онлайн-инференс в режиме реального времени;
- асинхронный инференс;
- пакетная офлайн-обработка.

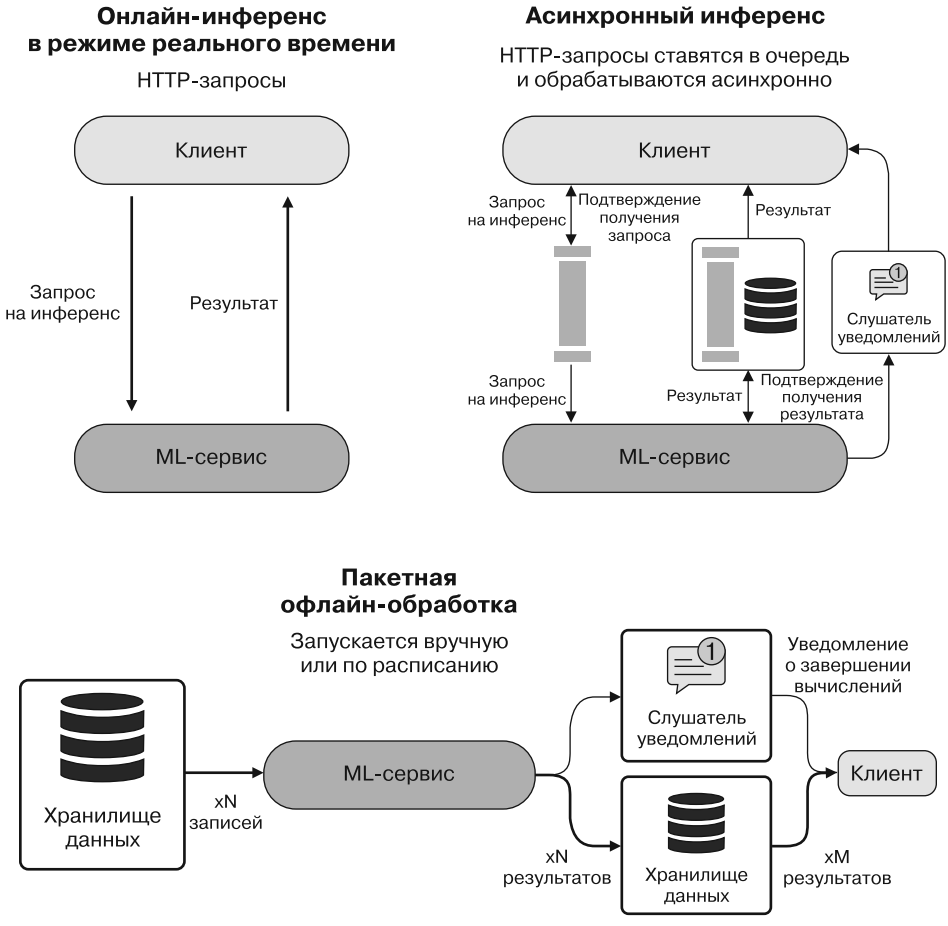


Рис. 10.1. Три базовые стратегии развертывания пайплайна инференса

Выбор того или иного варианта всегда предполагает компромисс между задержкой, пропускной способностью и стоимостью. При этом необходимо учитывать, каким образом осуществляется доступ к данным и какая инфраструктура

применяется. Еще один важный фактор — характер взаимодействия пользователя с моделью. Например, будет ли модель использоваться напрямую, как в случае с чат-ботом, или интегрироваться в систему в качестве скрытого компонента, например классификатора, проверяющего безопасность входных (или выходных) данных?

Необходимо также учитывать актуальность предсказаний. Если в вашем сценарии допустимы предсказания с некоторой задержкой, то вам может подойти более простой в реализации пакетный офлайн-режим. Если же предполагается работать с моделью в режиме реального времени, то вам потребуется более мощная инфраструктура. Кроме того, важно оценить характер трафика, то есть будет ли ваше приложение использоваться постоянно или вы ожидаете чередования пиков и спадов в уровне нагрузки?

Учитывая все вышесказанное, рассмотрим три основные стратегии развертывания пайплайна инференса.

Онлайн-инференс в режиме реального времени

Для инференса в режиме реального времени используется простая архитектура на основе сервера, доступ к которому осуществляется с помощью HTTP-запросов. Наиболее распространенные варианты — REST API или gRPC-сервер. REST API проще в реализации и более универсален, но работает медленнее, так как задействует формат JSON для обмена данными между клиентом и сервером. Такой подход обычно применяется в тех случаях, когда модель обслуживает пользователей, находящихся за пределами внутренней сети. Примером может служить API OpenAI.

В свою очередь, протокол gRPC обеспечивает более высокую скорость работы ML-сервера, но делает его менее гибким. В рамках этого подхода в клиентском приложении необходимо реализовать protobuf-схемы, работать с которыми сложнее, чем с JSON-структурами. Однако преимущество заключается в том, что protobuf-объекты можно скомпилировать в байтовое представление, что существенно ускоряет передачу данных по сети. В связи с этим данный протокол часто используется для организации работы внутренних сервисов в рамках одной ML-системы.

При реализации инференса в режиме реального времени клиент отправляет HTTP-запрос ML-сервису, который незамедлительно обрабатывает его и возвращает результат. Такое синхронное взаимодействие заставляет клиента дожидаться ответа перед продолжением работы.

Для обеспечения эффективной работы в таком режиме инфраструктура должна поддерживать отзывчивые сервисы с низкой задержкой, развернутые на быстрых масштабируемых серверах. При этом важно распределить нагрузку

для равномерного поступления входящего трафика, а также реализовать автоматическое масштабирование для адаптации к изменениям в уровне нагрузки. Высокая доступность тоже крайне важна для бесперебойной работы сервиса.

Такую архитектуру часто применяют для организации взаимодействия с LLM, например, в чат-ботах или API, пользователи которых ожидают мгновенного ответа. Сервисы вроде ChatGPT или Claude нередко задействуют протокол WebSocket, чтобы сделать интерфейс как можно более отзывчивым за счет передачи токенов по одному за раз. Другими примерами являются сервисы векторизации и повторного ранжирования, используемые в RAG-системах, а также рекомендательные механизмы таких платформ, как TikTok.

Простота взаимодействия «клиент — сервер» делает онлайн-инференс в режиме реального времени привлекательным решением для приложений, от которых ожидается мгновенный отклик, например, чат-ботов и систем рекомендаций. Однако данный подход плохо масштабируется и в периоды низкой активности приводит к неполной загрузке ресурсов.

Асинхронный инференс

При асинхронном инференсе клиент отправляет запрос ML-сервису, который подтверждает его получение и помещает запрос в очередь для последующей обработки. В отличие от предыдущей стратегии в данном случае клиент не ожидает немедленного ответа, так как запрос обрабатывается асинхронно. Реализация этого подхода требует надежной инфраструктуры, поддерживающей очередь сообщений, подлежащих дальнейшей обработке.

Вернуть результат клиенту можно разными способами. Например, в зависимости от объема данных результат можно сохранить в отдельной очереди или в специальном хранилище объектов.

Клиент может использовать механизм опроса (polling), периодически проверяя наличие новых результатов, или же можно реализовать систему push-уведомлений, сигнализирующую об их готовности.

Асинхронный инференс обеспечивает более эффективное использование ресурсов. При этом системе не требуется обрабатывать все запросы одновременно, достаточно задать ограничение на число работающих параллельно машин. Это возможно, поскольку запросы хранятся в очереди и обрабатываются по мере освобождения ресурсов. Это особенно полезно при резких всплесках трафика: например, если интернет-магазин обычно обслуживает 10 запросов в секунду с помощью двух машин, а во время акции трафик возрастает до 100 запросов в секунду, нет нужды увеличивать число машин в десять раз. Запросы просто накапливаются в очереди и обрабатываются с прежней скоростью без каких-либо ошибок.

Преимущества асинхронной архитектуры также проявляются в тех случаях, когда выполнение задачи занимает значительное время. Если на обработку запроса уходит более пяти минут, заставлять клиента дожидаться завершения этого процесса нецелесообразно.

Тем не менее у асинхронного инференса есть и недостатки. В частности, он увеличивает длительность задержки, что делает его менее подходящим для приложений, чувствительных к времени отклика. Кроме того, для него характерны более сложная реализация и повышенные требования к инфраструктуре. Можно сказать, что асинхронная архитектура предлагает компромисс между онлайн- и офлайн-стратегиями, балансируя их плюсы и минусы.

Это надежное решение, если задержка не играет критической роли, а приоритетом выступает оптимизация затрат. Такой подход часто применяется при решении задач вроде извлечения ключевых слов из документов, резюмирования текстов с помощью LLM или создания дипфейк-видео. Однако если настроить систему автоматического масштабирования так, чтобы она эффективно обрабатывала очередь запросов, то эту архитектуру можно использовать и в онлайн-системах рекомендаций — например, в сфере электронной коммерции. В конечном счете все зависит от того, какой объем вычислительных ресурсов вы готовы оплачивать для обеспечения заявленных характеристик приложения.

Пакетная офлайн-обработка

Пакетная обработка (batch transform) предполагает одновременную обработку больших объемов данных (по расписанию или вручную). В рамках этой архитектуры ML-сервис извлекает данные из хранилища, обрабатывает их за один шаг и возвращает результаты обратно. В качестве хранилища могут выступать как объектные хранилища вроде AWS S3, так и хранилища данных, например GCP BigQuery.

В отличие от асинхронного инференса пакетная обработка оптимизирована под высокую пропускную способность при слабо выраженных требованиях к задержке. Если вам не нужны предсказания в режиме реального времени, такой подход может значительно сократить издержки, поскольку обработка данных крупными пакетами является наиболее экономичным методом. Кроме того, стратегия пакетного преобразования максимально упрощает обслуживание модели, ускоряя процесс разработки.

В рамках этой стратегии клиент извлекает результаты непосредственно из хранилища данных, не взаимодействуя напрямую с ML-сервисом. Это означает, что клиенту не нужно ждать завершения обработки, однако он лишается возможности запрашивать новые предсказания в любой момент. Хранилище

здесь выступает как промежуточный буфер, где сохраняются готовые предсказания, которые клиент при необходимости может извлечь. Для повышения отзывчивости приложения допускается реализовать механизм уведомлений, посылающий клиенту сигнал о том, что обработка его запроса завершена и он может извлечь результаты.

К сожалению, такой подход всегда предполагает некоторую задержку между моментом генерации предсказаний и их потреблением, что не позволяет использовать его во всех типах приложений. Например, если вы создаете рекомендательную систему для видеосервиса, то задержка в один день для фильмов и сериалов может быть вполне допустимой, поскольку пользователи не обращаются к этой категории контента слишком часто. Однако в случае рекомендательной системы для соцсети задержка даже в один час становится неприемлемой, так как здесь важна оперативность.

Пакетная обработка особенно эффективна в сценариях, когда необходима высокая пропускная способность, например при анализе данных или периодической генерации отчетов. Однако эта стратегия не подходит для приложений, работающих в режиме реального времени, поскольку отличается высокой задержкой и требует тщательного планирования и составления расписаний для эффективной обработки больших наборов данных. Именно поэтому она относится к офлайн-методам обслуживания.

Итак, мы рассмотрели три наиболее распространенные архитектуры для обслуживания ML-моделей. Сначала мы обсудили онлайн-инференс в режиме реального времени, при котором предсказания модели выдаются по запросу клиента. Затем мы перешли к асинхронному инференсу — промежуточному варианту между онлайн- и офлайн-режимами. Наконец, обсудили пакетную офлайн-обработку, при которой модель обрабатывает большие объемы данных и сохраняет результаты в хранилище, откуда клиент впоследствии их извлекает.

Монолитная и микросервисная архитектура в контексте обслуживания модели

В предыдущем разделе мы рассмотрели три способа развертывания ML-сервиса. Они различались прежде всего взаимодействием клиента и модели и такими характеристиками, как используемый протокол, отзывчивость сервиса и актуальность предсказаний.

Однако важно учитывать и архитектуру самого ML-сервиса, которая может быть реализована в виде монолитного сервера или набора микросервисов. Этот выбор влияет на дальнейшее внедрение, сопровождение и масштабирование ML-сервиса. Рассмотрим оба варианта подробнее (рис. 10.2).

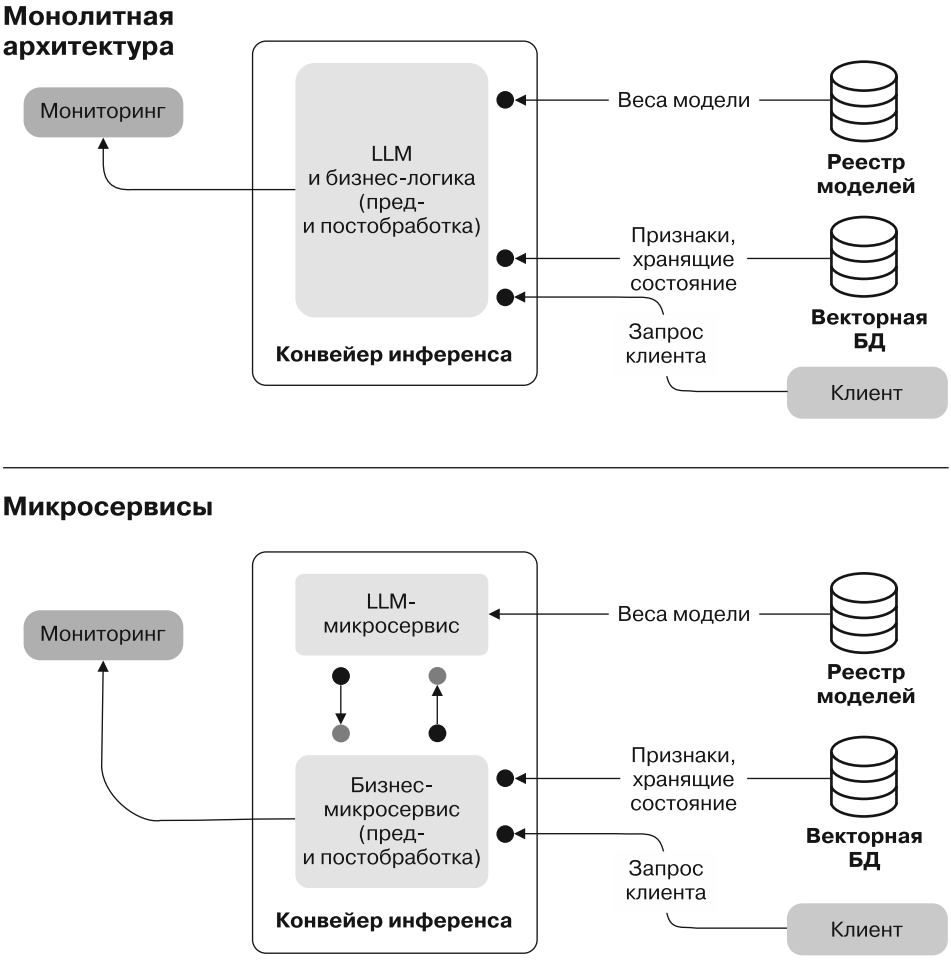


Рис. 10.2. Сравнение монолитной и микросервисной архитектуры в контексте обслуживания модели

Монолитная архитектура

В монолитной архитектуре LLM (или любая другая ML-модель) и связанная с ней бизнес-логика (этапы предварительной обработки и постобработки) объединяются в один сервис. Такой подход легко реализуется на начальном этапе разработки проекта, так как все компоненты сосредоточены в одной кодовой базе. Это упрощает сопровождение небольших и средних приложений, поскольку все изменения происходят внутри единой системы.

Однако основное ограничение монолитной архитектуры заключается в том, что отдельные ее компоненты невозможно масштабировать независимо друг от друга. LLM, как правило, требует вычислений на GPU, в то время как остальная бизнес-логика опирается на CPU и операции ввода/вывода. Одновременная оптимизация инфраструктуры под GPU и CPU приводит к неэффективному использованию ресурсов, например к простоям GPU во время выполнения бизнес-логики. Это сопровождается дополнительными затратами, которых можно было бы избежать.

Кроме того, такая архитектура снижает гибкость, поскольку все компоненты привязаны к одному технологическому стеку и среде исполнения. Например, вы можете захотеть запускать LLM на Rust или C++ либо компилировать ее с помощью ONNX или TensorRT, а бизнес-логику реализовать на Python. Сосредоточение всего кода в одной системе существенно затрудняет подобное разграничение. Наконец, это усложняет распределение работы между разными командами, что часто приводит к возникновению узких мест и снижению оперативности разработки.

Микросервисная архитектура

Микросервисная архитектура разбивает пайплайн инференса на отдельные, независимые сервисы, как правило разделяя LLM и бизнес-логику на самостоятельные компоненты. Эти сервисы обмениваются данными по сети, используя протоколы вроде REST или gRPC.

Как показано на рис. 10.3, основным преимуществом такого подхода является возможность независимого масштабирования компонентов. Например, если службе с LLM требуется больше ресурсов GPU, ее можно масштабировать горизонтально, не затрагивая остальные части системы. Это обеспечивает более рациональное использование ресурсов и снижает издержки, поскольку мы можем применять разные типы машин (например, GPU или CPU) в зависимости от потребностей конкретного сервиса.

Предположим, что инференс LLM стал занимать больше времени и вы хотите увеличить число реплик ML-сервиса, чтобы справиться с возросшей нагрузкой. Однако использование машин с GPU обходится дорого. Разделив компоненты, вы можете выполнять на машине с GPU только инференс модели, не занимая ее задачами, которые можно поручить более дешевым машинам с CPU.

Таким образом, разбивая систему на компоненты, вы получаете возможность масштабировать ее горизонтально по мере необходимости, минимизируя издержки и предлагая экономически эффективное решение, соответствующее вашим требованиям.

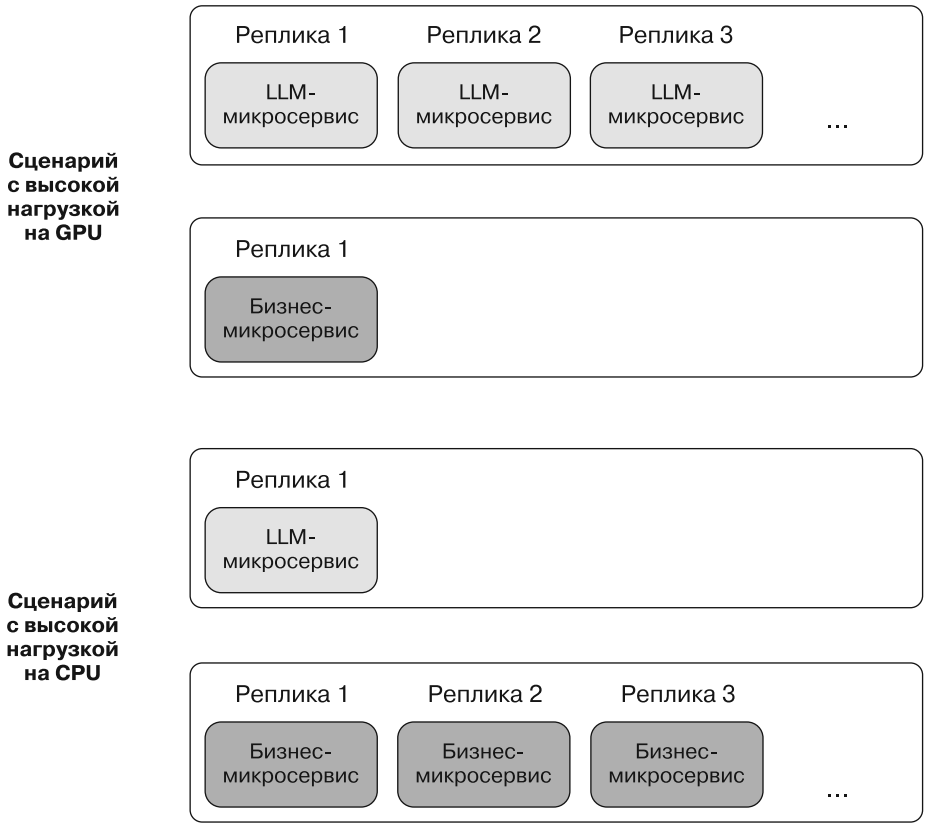


Рис. 10.3. Независимое масштабирование микросервисов на основе их потребностей в вычислительных мощностях

Кроме того, каждый микросервис может использовать наиболее подходящий технологический стек, что позволяет командам проводить независимую оптимизацию и экспериментировать с новыми решениями.

Однако по сравнению с монолитной системой микросервисная архитектура является более сложной в плане развертывания и сопровождения, так как сервисы приходится развертывать, мониторить и сопровождать независимо друг от друга.

В то же время возросшее сетевое взаимодействие между сервисами может привести к увеличению задержек и появлению потенциальных точек отказа, что требует внедрения надежных механизмов мониторинга и обеспечения устойчивости.

Предложенную схему разделения модели и бизнес-логики на два сервиса при необходимости можно расширить, например создав отдельный сервис для предварительной обработки данных, один — для самой модели и еще один — для постобработки. Опираясь на четыре основных фактора (задержка, пропускная способность, данные и инфраструктура), вы можете спроектировать оптимальную архитектуру, отвечающую задачам вашего приложения.

Выбор между монолитной и микросервисной архитектурой

Выбор между монолитной и микросервисной архитектурой для обслуживания ML-моделей в значительной степени зависит от специфики приложения. Монолитный подход может быть оптимален для небольших команд или несложных приложений, где на первом месте стоят простота разработки и сопровождения. Он также хорошо подходит для проектов, не требующих частого масштабирования. Кроме того, если модели невелики, не нуждаются в графическом процессоре или могут работать на более дешевых и маломощных GPU, то компромисс между снижением затрат и усложнением инфраструктуры также стоит учесть при выборе архитектуры.

В свою очередь, микросервисы с их гибкостью и масштабируемостью особенно хорошо подходят для больших комплексных систем, компоненты которых предъявляют разные требования к масштабированию или используют различные технологические стеки. Эта архитектура особенно уместна в случае LLM, работающих на мощных машинах с графическими процессорами, такими как Nvidia A100, V100 или A10g. Эти ресурсы весьма дороги, и микросервисный подход позволяет гибко управлять их загрузкой, поддерживая занятость данных машин и оперативно освобождая ресурсы, когда они не используются. Однако подобная гибкость достигается ценой увеличения сложности разработки и сопровождения.

Один из распространенных подходов заключается в том, чтобы начать с монолитной архитектуры и выделять микросервисы постепенно по мере роста проекта. Однако, чтобы сделать такой переход безболезненным и малозатратным, важно с самого начала проектировать монолит как модульную систему. Например, даже если все приложение работает на одной машине, можно разделить его на модули на уровне кода, чтобы упростить будущий перенос этих компонентов в отдельные микросервисы. В частности, при работе на Python можно реализовать ML-модель и бизнес-логику в виде двух независимых модулей, которые не взаимодействуют друг с другом напрямую, а затем объединить эти два модуля на более высоком уровне, например через сервисный класс.

Еще один вариант заключается в том, чтобы оформить модель и бизнес-логику в виде двух отдельных Python-пакетов, объединяемых аналогичным образом. Данный подход строго разграничивает два компонента, но при этом увеличивает сложность разработки. Таким образом, главная идея заключается в том, что, если вы начинаете с монолитной архитектуры, но в перспективе планируете перейти к микросервисной, необходимо изначально проектировать программное обеспечение с учетом модульности. В противном случае при тесном переплетении логики вам, скорее всего, придется переписать всю систему с нуля, что приведет к значительным затратам времени и ресурсов.

В итоге можно сказать, что монолитные архитектуры отличаются простотой и легкостью сопровождения, но уступают в гибкости и масштабируемости. Микросервисные архитектуры, напротив, дают простор для развития и быстрой адаптации системы, но более сложны в плане разработки и эксплуатации.

Стратегия развертывания пайплайна инференса для LLM-двойника

Теперь, когда мы обсудили все возможные стратегии развертывания пайплайна инференса, посмотрим, как он может быть реализован в нашей системе LLM-двойника.

Поскольку наша цель заключается в создании чат-бота для генерации контента, мы будем обрабатывать запросы последовательно, делая акцент на минимальной задержке, а значит, нам стоит выбрать стратегию онлайн-инференса в режиме реального времени.

Что касается вопроса архитектуры, то мы разделим ML-сервис на два компонента: сервер REST API, содержащий бизнес-логику, и LLM-микросервис, оптимизированный для запуска выбранной языковой модели. Поскольку для инференса LLM требуется мощная машина, а в будущем мы можем захотеть оптимизировать модель с помощью различных движков для сокращения задержек и экономии памяти, наиболее разумным решением является микросервисная архитектура. Такой подход позволяет гибко адаптировать инфраструктуру под модели разного размера. Например, модель с 8 млрд параметров после квантования можно запустить на одной машине с GPU Nvidia A10G. Однако если мы захотим использовать модель с 30 млрд параметров, нам придется перейти на Nvidia A100. При этом модернизация затронет только LLM-микросервис, а сервер REST API останется неизменным.

Как показано на рис. 10.4, в нашем случае основная часть бизнес-логики сосредоточена вокруг механизма RAG. Поэтому этапы поиска и дополнения контекста будут реализованы в рамках бизнес-микросервиса, в котором также

будут применяться описанные в предыдущей главе техники оптимизации этапов подготовки к поиску, самого поиска и работы с найденными документами.

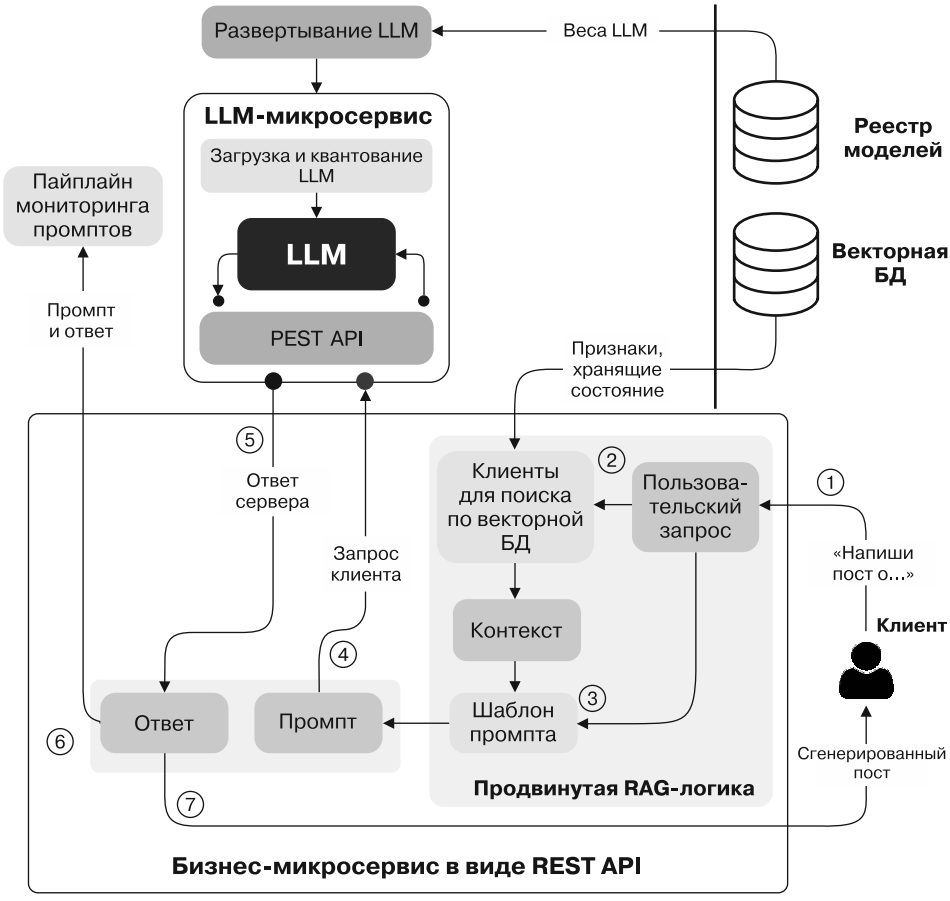


Рис. 10.4. Микросервисная архитектура пайплайна инференса LLM-двойника

LLM-микросервис строго оптимизирован под компонент генерации RAG-системы. В конечном счете микросевис с бизнес-логикой будет отправлять запрос (prompt trace), включающий пользовательский запрос, сформированный промпт, ответ модели и другие промежуточные шаги, в пайплайн мониторинга промптов, описанный в главе 11.

Таким образом, наша стратегия предполагает использование архитектуры микросервисов для разделения LLM и бизнес-логики на два сервиса и реализацию онлайн-инференса в режиме реального времени.

Рассмотрим схему пайплайна инференса. Для запуска этого пайплайна необходимо следующее:

- признаки, используемые для RAG в режиме реального времени и генерируемые пайплайном признаков (извлекаются из онлайн-хранилища признаков, а точнее, из векторной базы данных Qdrant);
- тонко настроенная LLM, созданная пайплайном обучения (загружается из реестра моделей).

С учетом этого рабочий процесс ML-сервиса выглядит следующим образом (см. рис. 10.4).

1. Пользователь отправляет запрос через HTTP.
2. На основе запроса извлекается релевантный контекст с помощью продвинутого поискового модуля RAG, реализованного в главе 4.
3. Пользовательский ввод и извлеченный контекст объединяются в итоговый промпт, формируемый с помощью специализированного шаблона.
4. Сформированный промпт передается LLM-микросервису по протоколу HTTP.
5. Бизнес-микросервис ожидает получения сгенерированного ответа.
6. Сгенерированный ответ вместе с запросом пользователя и другими ключевыми данными передается в пайплайн мониторинга промптов.
7. Наконец, сгенерированный ответ возвращается пользователю.

Теперь рассмотрим, какие технологии мы задействовали для реализации архитектуры, представленной на рис. 10.4. В качестве векторной базы данных применяется система Qdrant. Для хранения модели мы используем реестр моделей Hugging Face, который позволяет опубликовать модель и сделать ее доступной для всех, кто хочет протестировать приведенный в книге код. Таким образом, если у вас нет желания запускать пайплайн обучения (и тратить на это порядка $100), то можете воспользоваться уже готовой моделью. Как видите, возможность совместного использования и легкий доступ являются важнейшими преимуществами хранения модели в реестре.

Бизнес-микросервис реализуется с помощью популярного, удобного и быстрого фреймворка FastAPI. LLM-микросервис будет развернут на платформе AWS SageMaker с использованием *контейнеров для глубокого обучения (Deep Learning Containers, DLC)* от Hugging Face. О контейнерах DLC мы поговорим в следующем разделе, однако в общих чертах их можно представить в качестве движка для инференса, оптимизирующего работу LLM на этапе обслуживания. Пайплайн мониторинга промптов мы обсудим в главе 11.

Для развертывания пайплайна инференса на платформе SageMaker необходимы следующие компоненты.

- *Конечная точка SageMaker* — масштабируемый и безопасный API, размещаемый на платформе SageMaker для обеспечения инференса в режиме реального времени. По сути, это интерфейс для взаимодействия приложений с моделью. После развертывания приложение сможет направлять HTTP-запросы к этой точке для получения предсказаний в реальном времени.
- *Модель SageMaker* — артефакт, полученный в результате обучения алгоритма и включающий необходимые для генерации предсказаний веса и вычислительную логику. Мы можем создать несколько моделей и использовать их в различных конфигурациях.
- *Конфигурация SageMaker* — определяет аппаратную и программную среду, на которой размещается модель. В ней задаются тип и количество вычислительных мощностей для инференса. Эти конфигурации используются при создании и обновлении конечных точек и позволяют гибко управлять развертыванием и масштабированием моделей.
- *Компонент SageMaker Inference* — связывает модель и конфигурацию с конечной точкой. В рамках одной конечной точки можно развернуть несколько моделей, каждая из которых имеет собственную конфигурацию ресурсов. После развертывания доступ к модели осуществляется через API InvokeEndpoint с помощью Python.

Эти компоненты вместе образуют надежную инфраструктуру для развертывания и управления ML-моделями на платформе SageMaker, обеспечивая масштабируемость, безопасность и эффективность инференса в режиме реального времени.

Аналогичные решения предусмотрены и на других облачных платформах. Список инструментов для развертывания моделей, таких как Hopsworks, Modal, Vertex AI, Seldon и BentoML, постоянно пополняется. Главное — понимать требования вашей конкретной задачи и выбирать тот инструмент, который наилучшим образом им соответствует.

Сравнение пайплайнов обучения и инференса. Прежде чем приступить к развертыванию пайплайна инференса, важно четко понять различия между пайплайнами обучения и инференса. Хотя может показаться очевидным, что первый используется для обучения, а второй — для инференса, между ними существуют важные различия, знание которых необходимо для понимания технических аспектов обсуждаемой архитектуры.

Одно из главных отличий заключается в способах работы с данными. Во время обучения данные, как правило, извлекаются из офлайн-хранилищ в пакетном

режиме (batch mode), что обеспечивает высокую пропускную способность и предоставляет возможность отследить происхождение данных. Например, в архитектуре LLM-двойника используются артефакты ZenML, которые позволяют извлекать, версионировать и отслеживать данные, поступающие в цикл обучения. Напротив, пайплайн инференса опирается на онлайн-базу данных, оптимизированную под низкую задержку. В нашем случае задействована векторная БД Qdrant, из которой извлекается контекст, необходимый для выполнения RAG. При этом акцент смещается с контроля версий и происхождения данных на оперативный доступ к ним, что обеспечивает непрерывность пользовательского опыта. Выходные данные этих пайплайнов также существенно различаются. Пайплайн обучения выдает веса, которые сохраняются в реестре моделей, а пайплайн инференса — предсказания, возвращаемые непосредственно пользователю.

Кроме того, каждый из пайплайнов предъявляет разные требования к инфраструктуре. Так, пайплайн обучения нуждается в мощных машинах, оснащенных как можно большим числом GPU. Чем больше объем доступной вычислительной мощности и VRAM, тем больше размер пакета (и выше пропускная способность), короче время обучения и шире возможности для экспериментов. Пайплайн инференса, напротив, обычно требует значительно меньше вычислительных ресурсов, так как на этапе инференса модель получает на вход один образец или небольшой пакет данных и не нуждается в выполнении шагов оптимизации.

Несмотря на все перечисленные различия, между двумя пайплайнами есть и сходства, особенно в части предварительной и постобработки данных. Крайне важно, чтобы одни и те же функции и гиперпараметры применялись как во время обучения, так и на этапе инференса. Любые расхождения могут привести к рассогласованию между обучением и инференсом, при котором производительность модели на этапе инференса отличается от ее производительности на этапе обучения.

Развертывание сервиса LLM-двойника

Теперь реализуем архитектуру, описанную в предыдущем разделе, а именно развернем LLM-микросервис с использованием AWS SageMaker и бизнес-микросервис с помощью FastAPI. В рамках бизнес-микросервиса мы свяжем реализованную в главе 9 логику RAG с нашей тонко настроенной моделью LLM-двойника, что позволит протестировать работу пайплайна инференса.

Развертывание ML-модели — это один из важнейших этапов жизненного цикла любой ML-системы, поскольку только после его завершения пользователи получают доступ к созданному вами приложению. Какой бы мощной и высококачественной ни была ваша модель, при неправильно спроектированной архи-

тектуре или нестабильной работе инфраструктуры она окажется практически бесполезной с точки зрения бизнеса. Например, если ваш ИИ-помощник для программистов является лучшим на рынке, но работает с большой задержкой или постоянными сбоями при обращении к API, то пользователи, скорее всего, переключатся на менее производительное, но более стабильное решение.

Итак, в этом разделе мы рассмотрим следующие шаги:

- развертывание тонко настроенной модели LLM-двойника на платформе AWS SageMaker;
- написание клиента для инференса, взаимодействующего с развернутой моделью;
- создание бизнес-сервиса с помощью FastAPI;
- интеграция логики RAG с тонко настроенной LLM;
- реализация правил автоматического масштабирования LLM-микросервиса.

Реализация LLM-микросервиса с использованием AWS SageMaker

Наша цель — развернуть модель LLM-двойника, размещенную в реестре моделей Hugging Face, на платформе Amazon SageMaker в качестве конечной точки для онлайн-инференса в режиме реального времени. Для этого мы воспользуемся специализированным контейнером инференса *DLC* от Hugging Face.

Контейнер DLC от Hugging Face

DLC (Deep Learning Container — «контейнер для глубокого обучения») — это специализированный образ Docker, оснащенный основными фреймворками и библиотеками для глубокого обучения, в том числе популярными пакетами Hugging Face `Transformers`, `Datasets` и `Tokenizers`. Эти контейнеры упрощают процесс обучения и развертывания моделей, устраняя необходимость в сложной настройке и оптимизации окружения. Особенно полезным является контейнер инференса Hugging Face Inference DLC, который включает в себя полнофункциональный стек для обслуживания моделей, что значительно упрощает процесс их развертывания в продакшене и снижает требования к технической квалификации людей, которые этим занимаются.

На этапе обслуживания моделей контейнер DLC использует движок *Text Generation Inference* (TGI), разработанный Hugging Face: https://github.com/huggingface/text-generation-inference.

TGI — это решение с открытым исходным кодом для развертывания и обслуживания LLM. Применяя тензорный параллелизм и динамическую пакетную

обработку, оно обеспечивает высокопроизводительную генерацию текста для самых популярных моделей, доступных на Hugging Face, таких как Mistral, Llama и Falcon. Если вкратце, то наиболее мощными возможностями образа DLC являются:

- *тензорный параллелизм*, повышающий вычислительную эффективность инференса;
- *оптимизация кода трансформеров для этапа инференса*, использующая алгоритм FlashAttention для повышения производительности на самых популярных архитектурах (https://github.com/Dao-AILab/flash-attention);
- *квантование с помощью пакета* `bitsandbytes`, уменьшающее размер модели при сохранении ее производительности и делающее процесс развертывания более эффективным (https://github.com/bitsandbytes-foundation/bitsandbytes);
- *непрерывная пакетная обработка входящих запросов*, повышающая пропускную способность за счет динамического формирования пакетов;
- *ускоренная загрузка весов* с использованием формата `safetensors`, сокращающая время инициализации модели (https://github.com/huggingface/safetensors);
- *потоковая передача токенов*, обеспечивающая взаимодействие в режиме реального времени через механизм *Server-Sent Events* (SSE).

Таким образом, наша модель LLM-двойника будет работать внутри Docker-образа DLC, обрабатывая входящие запросы, оптимизируя модель для инференса и возвращая результаты в режиме реального времени. Контейнеры DLC будут размещены на платформе AWS SageMaker в виде конечных точек инференса, к которым можно обращаться с помощью HTTP-запросов. Теперь перейдем к реализации: начнем с развертывания LLM, а затем создадим класс-обертку для взаимодействия с конечной точкой инференса SageMaker.

Настройка ролей в системе SageMaker

Первым шагом является создание IAM-пользователей и ролей, необходимых для доступа к инфраструктуре SageMaker и ее развертывания. Система *AWS IAM* (Identity and Access Management — «управление идентификацией и доступом») контролирует процесс аутентификации и получения доступа к ресурсам. С помощью интерфейса IAM вы можете создавать новых пользователей (назначаемых людям) и новые роли (назначаемые другим компонентам инфраструктуры.

Весь процесс развертывания автоматизирован. Нам предстоит выполнить несколько команд в консоли, но сперва нужно убедиться, что вы корректно задали переменные окружения `AWS_ACCESS_KEY`, `AWS_SECRET_KEY` и `AWS_REGION` в файле `.env`. На этом этапе проще всего использовать учетные данные, привязанные к административной роли, поскольку далее мы создадим более узкоспециализированные IAM-роли, предназначенные для работы с остальной частью главы.

После задания переменных в файле `.env` необходимо совершить ряд действий.

1. Создать IAM-пользователя, имеющего доступ только к тем ресурсам, которые требуются для развертывания сервиса, а именно: SageMaker, *Elastic Container Registry* (ECR) и S3. Для этого выполните следующую команду:

```
poetry poe create-sagemaker-role
```

   В результате будет сгенерирован JSON-файл с именем `sagemaker_user_credentials.json`, содержащий новый ключ доступа и секретный ключ AWS. Теперь мы будем работать с этими учетными данными при выполнении всех операций, связанных с SageMaker, чтобы вносимые нами изменения влияли только на ресурсы, относящиеся к этой платформе. В противном случае, при использовании учетной записи администратора, мы могли бы случайно изменить другие ресурсы AWS, понеся дополнительные затраты или нарушив работу других проектов. Именно поэтому в рамках конкретного сценария рекомендуется использовать узкоспециализированную роль.

   На заключительном этапе необходимо извлечь новые ключи из JSON-файла и обновить переменные `AWS_ACCESS_KEY` и `AWS_SECRET_KEY` в файле `.env`. Соответствующая реализация доступна по ссылке https://github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/llm_engineering/infrastructure/aws/roles/create_sagemaker_role.py:

2. Создать IAM-роль выполнения (execution role), которая будет прикреплена к процессу развертывания SageMaker и позволит сервису обращаться к другим ресурсам AWS от вашего имени. Это стандартная практика для облачного развертывания: вместо того чтобы аутентифицировать каждую машину с использованием ваших личных учетных данных, вы прикрепляете к ней роль, которая предоставляет доступ только к необходимым компонентам инфраструктуры. В нашем случае SageMaker потребуется доступ к следующим ресурсам: S3, CloudWatch и ECR. Для создания этой роли выполните команду:

```
poetry poe create-sagemaker-execution-role
```

   В результате будет сгенерирован JSON-файл с именем `sagemaker_execution_role.json`, содержащий *ARN* (Amazon Resource Name) новой роли. ARN — это уникальный идентификатор ресурса AWS внутри облачной инфраструктуры. Скопируйте значение ARN из JSON-файла и присвойте его переменной `AWS_ARN_ROLE` в файле `.env`. Соответствующая реализация доступна по ссылке: https://github.com/PacktPublishing/LLM-Engineers-Handbook/blob/main/llm_engineering/infrastructure/aws/roles/create_execution_role.py:



При возникновении ошибок настройте инструмент AWS CLI с использованием учетных данных, содержащихся в файле .env, и повторите процесс. Официальная документация по установке AWS CLI доступна по адресу https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html.

После задания IAM-пользователя и роли в файле `.env` они будут автоматически загружены в Python-объект `settings` и применяться при выполнении всех последующих шагов. А теперь давайте приступим непосредственно к развертыванию.

Развертывание модели LLM-двойника на платформе AWS SageMaker

Процесс развертывания модели на платформе AWS SageMaker полностью автоматизирован с помощью набора Python-классов, рассматриваемых в этой главе. В данном разделе будет описан способ конфигурирования инфраструктуры SageMaker непосредственно из Python. Таким образом, вам не придется выполнять все действия вручную, как в случае с типичным пошаговым руководством, — достаточно понять, как работает код.

Весь процесс развертывания можно осуществить с помощью простой консольной команды: `poe deploy-inference-endpoint`. Она инициирует все шаги, представленные на рис. 10.5, кроме создания IAM-пользователей и ролей для SageMaker, поскольку мы уже сделали это на предыдущем этапе.

В данном разделе мы пошагово разберем код, автоматизирующий процесс, представленный на рис. 10.5, начиная с функции `create_endpoint()`. В конце мы протестируем CLI-команду и удостоверимся в успешном завершении развертывания с помощью консоли AWS. Исходный код для развертывания модели на платформе SageMaker доступен по адресу https://github.com/PacktPublishing/LLM-Engineers-Handbook/tree/main/llm_engineering/infrastructure/aws/deploy.

Рассмотрим представленную реализацию, начиная с основной функции, отвечающей за развертывание модели LLM-двойника на платформе AWS SageMaker. В рамках этой функции мы с помощью `get_huggingface_llm_image_uri()` извлекаем последнюю версию Docker-образа DLC, который затем передается в класс `SagemakerHuggingfaceStrategy` (стратегия развертывания) вместе с экземплярами `ResourceManager` (менеджер ресурсов) и `DeploymentService` (сервис развертывания):

```
def create_endpoint(endpoint_type=EndpointType.INFERENCE_COMPONENT_BASED):
    llm_image = get_huggingface_llm_image_uri("huggingface", version=None)

    resource_manager = ResourceManager()
    deployment_service = DeploymentService(resource_manager=resource_manager)

    SagemakerHuggingfaceStrategy(deployment_service).deploy(
        role_arn=settings.ARN_ROLE,
        llm_image=llm_image,
        config=hugging_face_deploy_config,
        endpoint_name=settings.SAGEMAKER_ENDPOINT_INFERENCE,
        endpoint_config_name=settings.SAGEMAKER_ENDPOINT_CONFIG_INFERENCE,
        gpu_instance_type=settings.GPU_INSTANCE_TYPE,
        resources=model_resource_config,
        endpoint_type=endpoint_type,
    )
```

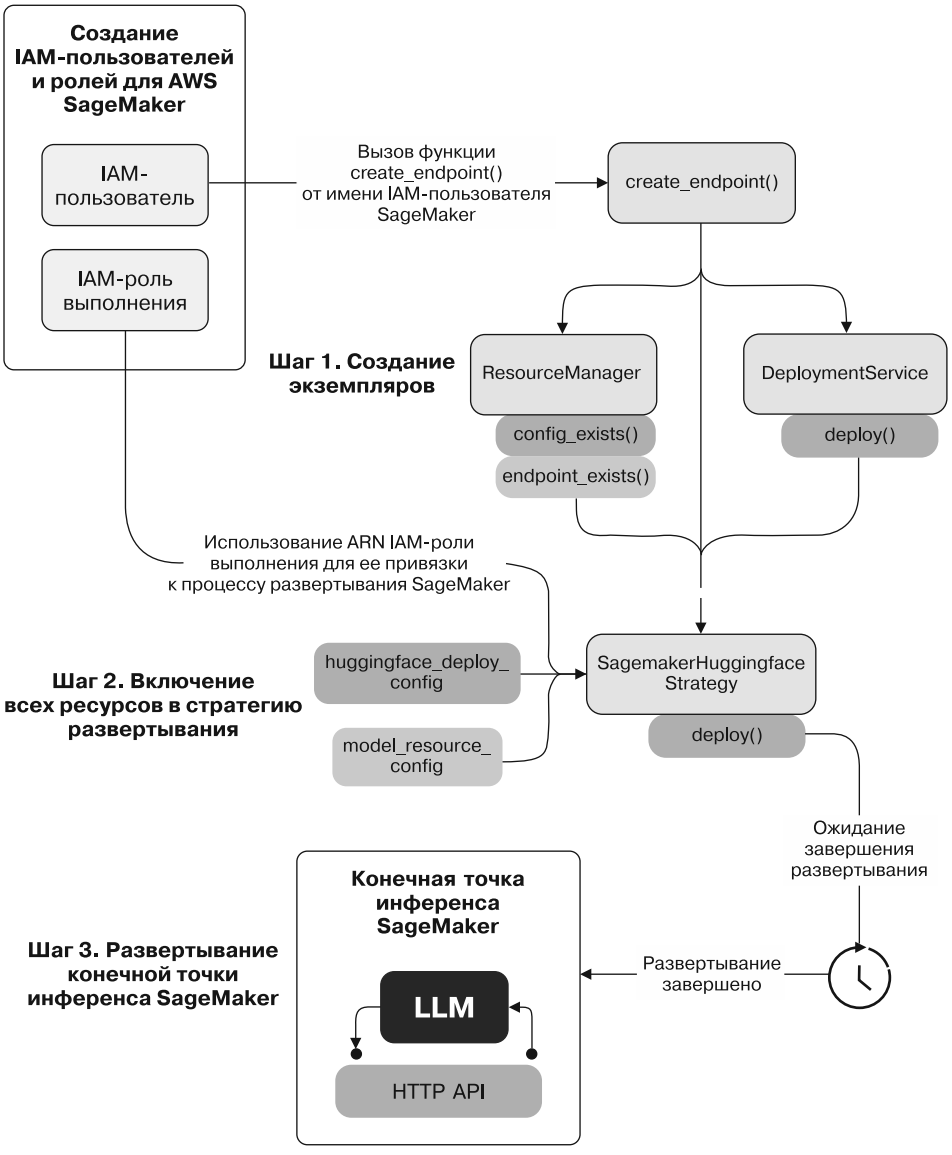


Рис. 10.5. Этапы развертывания модели на платформе AWS SageMaker

Чтобы лучше понять процесс развертывания, необходимо рассмотреть три класса, используемых в функции `create_endpoint()`. Начнем с класса `ResourceManager`. Он начинается с метода инициализации, который устанавливает соединение с AWS SageMaker с помощью библиотеки `boto3` — официального AWS SDK (набора средств разработки) для Python. Эта библиотека предоставляет

все необходимые функции для взаимодействия с различными сервисами AWS, включая SageMaker.

```
class ResourceManager:
    def __init__(self) -> None:
        self.sagemaker_client = boto3.client(
            "sagemaker",
            region_name=settings.AWS_REGION,
            aws_access_key_id=settings.AWS_ACCESS_KEY,
            aws_secret_access_key=settings.AWS_SECRET_KEY,
        )
```

Затем мы реализуем метод `endpoint_config_exists`, который проверяет существование конкретной конфигурации конечной точки SageMaker.

```
    def endpoint_config_exists(self, endpoint_config_name: str) -> bool:
        try:
            self.sagemaker_client.describe_endpoint_
            ➥config (EndpointConfigName=endpoint_config_name)
            logger.info(f"Endpoint configuration '{endpoint_config_name}'
            ➥exists.")
            return True
        except ClientError:
            logger.info(f"Endpoint configuration '{endpoint_config_name}'
            ➥does not exist.")
            return False
```

В этом классе также присутствует метод `endpoint_exists`, проверяющий наличие конкретной конечной точки SageMaker:

```
    def endpoint_exists(self, endpoint_name: str) -> bool:
        try:
            self.sagemaker_client.describe_endpoint(EndpointName=endpoint_name)
            logger.info(f"Endpoint '{endpoint_name}' exists.")
            return True
        except self.sagemaker_client.exceptions.ResourceNotFoundException:
            logger.info(f"Endpoint '{endpoint_name}' does not exist.")
            return False
```

Теперь рассмотрим класс `DeploymentService`. В его конструкторе (`__init__`) инициализируется клиент `sagemaker_client`, предназначенный для взаимодействия с AWS SageMaker, а также создается экземпляр упомянутого выше класса `ResourceManager`:

```
class DeploymentService:
    def __init__(self, resource_manager):
        self.sagemaker_client = boto3.client(
            "sagemaker",
            region_name=settings.AWS_REGION,
            aws_access_key_id=settings.AWS_ACCESS_KEY,
            aws_secret_access_key=settings.AWS_SECRET_KEY,
        )
        self.resource_manager = resource_manager
```

Ключевым методом этого класса является `deploy()`. Именно он управляет процессом развертывания модели на конечной точке SageMaker. Данный метод проверяет наличие необходимых конфигураций, а при их отсутствии запускает процедуру развертывания:

```
    def deploy(
        self,
        role_arn: str,
        llm_image: str,
        config: dict,
        endpoint_name: str,
        endpoint_config_name: str,
        gpu_instance_type: str,
        resources: Optional[dict] = None,
        endpoint_type: enum.Enum = EndpointType.MODEL_BASED,
    ) -> None:
        try:
            if self.resource_manager.endpoint_config_exists(endpoint_config_
            ➥name=endpoint_config_name):
                logger.info(f"Endpoint configuration {endpoint_config_name}
                ➥exists. Using existing configuration...")
        else:
            logger.info(f"Endpoint configuration{endpoint_config_name} does
            ➥not exist.")

        self.prepare_and_deploy_model(
            role_arn=role_arn,
            llm_image=llm_image,
            config=config,
            endpoint_name=endpoint_name,
            update_endpoint=False,
            resources=resources,
            endpoint_type=endpoint_type,
            gpu_instance_type=gpu_instance_type,
        )

        logger.info(f"Successfully deployed/updated model to endpoint
        ➥{endpoint_name}.")
    except Exception as e:
        logger.error(f"Failed to deploy model to SageMaker: {e}")
        raise
```

Первым делом метод `deploy()` проверяет существование конфигурации конечной точки с помощью `resource_manager`. Это позволяет избежать повторного развертывания в том случае, если соответствующая конечная точка уже настроена. Непосредственное развертывание модели на конкретной конечной точке SageMaker осуществляется путем вызова метода `prepare_and_deploy_model()`, который, в свою очередь, является статическим методом класса `DeploymentService` и отвечает за подготовку модели Hugging Face и ее развертывание на платформе SageMaker:

```
@staticmethod
def prepare_and_deploy_model(
    role_arn: str,
```

```
    llm_image: str,
    config: dict,
    endpoint_name: str,
    update_endpoint: bool,
    gpu_instance_type: str,
    resources: Optional[dict] = None,
    endpoint_type: enum.Enum = EndpointType.MODEL_BASED,
) -> None:
    huggingface_model = HuggingFaceModel(
        role=role_arn,
        image_uri=llm_image,
        env=config,
        transformers_version="4.6",
        pytorch_version="1.13",
        py_version="py310",
    )

    huggingface_model.deploy(
        instance_type=gpu_instance_type,
        initial_instance_count=1,
        endpoint_name=endpoint_name,
        update_endpoint=update_endpoint,
        resources=resources,
        tags=[{"Key": "task", "Value": "model_task"}],
        endpoint_type=endpoint_type,
    )
```

Этот метод начинает с создания экземпляра `HuggingFaceModel` — специализированного класса SageMaker, предназначенного для работы с моделями Hugging Face. Его конструктор принимает несколько параметров, включая ARN роли (предоставляющей SageMaker необходимые права доступа), URI Docker-образа DLC и конфигурацию LLM, в которой указывается, какую модель Hugging Face следует загрузить и какие параметры инференса использовать (например, максимальное число токенов).

После создания экземпляра `HuggingFaceModel` данный метод вызывает функцию `deploy`, чтобы развернуть модель на платформе SageMaker. В то же время указываются тип вычислительного экземпляра, количество экземпляров, а также необходимость обновить существующую конечную точку или создать новую. Этот метод также допускает использование дополнительных параметров в более сложных сценариях, например `initial_instance_count` для конечных точек с несколькими моделями (multi-model endpoints) и `tags` для отслеживания и классификации ресурсов.

Теперь рассмотрим класс `SagemakerHuggingfaceStrategy`, объединяющий всю описанную выше логику. При инициализации он принимает только один параметр — экземпляр класса `DeploymentService`:

```
class SagemakerHuggingfaceStrategy(DeploymentStrategy):
    def __init__(self, deployment_service):
        self.deployment_service = deployment_service
```

Основная функциональность класса `SagemakerHuggingfaceStrategy` сосредоточена в его методе `deploy()`, который управляет всем процессом, принимая в качестве аргументов параметры, определяющие способ развертывания модели Hugging Face на платформе AWS SageMaker:

```
def deploy(
    self,
    role_arn: str,
    llm_image: str,
    config: dict,
    endpoint_name: str,
    endpoint_config_name: str,
    gpu_instance_type: str,
    resources: Optional[dict] = None,
    endpoint_type: enum.Enum = EndpointType.MODEL_BASED,
) -> None:
    logger.info("Starting deployment using Sagemaker Huggingface Strategy...")
    logger.info(
        f"Deployment parameters: nb of replicas: {settings.COPIES}, nb
        ➥of gpus:{settings.GPUS}, instance_type:{settings.GPU_INSTANCE_TYPE}"
)
```

Этими важнейшими параметрами являются:

- `role_arn` — IAM-роль, предоставляющая необходимые права для развертывания;
- `llm_image` — URI Docker-образа контейнера DLC;
- `config` — словарь с параметрами окружения модели;
- `endpoint_name` и `endpoint_config_name` — имена конечной точки и ее конфигурации;
- `gpu_instance_type` — тип экземпляров EC2 с GPU;
- `resources` — необязательный словарь ресурсов, используемый при развертывании нескольких моделей на одной конечной точке;
- `endpoint_type` — тип конечной точки `MODEL_BASED` или `INFERENCE_COMPONENT` (в зависимости от наличия компонента для инференса).

Данный метод делегирует сам процесс развертывания объекту `deployment_service`. Этот прием, характерный для паттерна Стратегия, позволяет гибко управлять процессом развертывания, не затрагивая его верхнеуровневую логику.

```
    try:
        self.deployment_service.deploy(
            role_arn=role_arn,
            llm_image=llm_image,
            config=config,
            endpoint_name=endpoint_name,
            endpoint_config_name=endpoint_config_name,
            gpu_instance_type=gpu_instance_type,
```

```
            resources=resources,
            endpoint_type=endpoint_type,
        )
        logger.info("Deployment completed successfully.")
    except Exception as e:
        logger.error(f"Error during deployment: {e}")
        raise
```

Для лучшего понимания инфраструктуры рассмотрим объект `ResourceRequirements`, используемый при настройке конфигураций с несколькими конечными точками и репликами моделей. Такая конфигурация позволяет обслуживать множество клиентов, соблюдая требования, предъявляемые к задержке и пропускной способности приложения. Объект `ResourceRequirements` инициализируется словарем, содержащим такие параметры ресурсов, как число развертываемых реплик (копий) модели, количество необходимых графических процессоров, ядер CPU и объем памяти в мегабайтах. Каждый из этих параметров оказывает значительное влияние на производительность и масштабируемость развернутой модели.

```
from sagemaker.compute_resource_requirements.resource_requirements import
➥ResourceRequirements

    model_resource_config = ResourceRequirements(
    requests={
        "copies": settings.COPIES,
        "num_accelerators": settings.GPUS
        "num_cpus": settings.CPUS,
        "memory": 5 * 1024
    },
)
```

В приведенном выше фрагменте кода объект `ResourceRequirements` настраивается с помощью четырех ключевых параметров:

- `copies` — определяет, сколько экземпляров (реплик) модели следует развернуть. Использование нескольких реплик позволяет сократить длительность задержки и повысить пропускную способность;
- `num_accelerators` — указывает количество выделяемых GPU. Поскольку большие языковые модели требуют значительных вычислительных ресурсов, для ускорения инференса обычно необходимо несколько GPU;
- `num_cpus` — определяет число ядер CPU, выделяемых для развертывания. От этого параметра зависит способность модели осуществлять предварительную и постобработку данных, а также выполнять другие важные, но не зависящие от GPU задачи;
- `memory` — задает минимально необходимый объем оперативной памяти для развертывания. Достаточный объем памяти является важным условием для корректной загрузки модели и ее стабильной работы.

За счет настройки этих параметров данный класс обеспечивает наличие достаточных вычислительных ресурсов для эффективной работы модели после ее развертывания на конечной точке SageMaker. Значения следует подбирать с учетом характеристик используемой LLM: ее размера, сложности решаемых задач и ожидаемого уровня нагрузки. Чтобы лучше понять влияние каждой из этих настроек, мы рекомендуем поэкспериментировать с ними после развертывания конечной точки, наблюдая за изменением производительности LLM-микросервиса.

Наконец, давайте рассмотрим параметры, определяющие конфигурацию движка LLM. Переменная `HF_MODEL_ID` указывает, какую модель Hugging Face следует развернуть. Например, в классе `settings` для нее задано значение `mlabonne/TwinLlama-3.1-8B-13`, которое соответствует нашей модели LLM-двойника, хранящейся на платформе Hugging Face. Параметр `SM_NUM_GPUS` определяет число GPU, выделяемых для каждой реплики модели, что критически важно для ее размещения в видеопамяти. Переменная `HUGGING_FACE_HUB_TOKEN` используется для получения доступа к Hugging Face Hub и извлечения модели. Переменная `HF_MODEL_QUANTIZE` задает нужную технику квантования, а остальные переменные управляют процессом генерации токенов.

```
hugging_face_deploy_config = {
   "HF_MODEL_ID": settings.HF_MODEL_ID,
   # Количество GPU на одну реплику модели
   "SM_NUM_GPUS": json.dumps(settings.SM_NUM_GPUS),
   # Максимальная длина входного текста
   "MAX_INPUT_LENGTH": json.dumps(settings.MAX_INPUT_LENGTH),
   # Максимальная длина генерируемой последовательности (включая входной текст)
   "MAX_TOTAL_TOKENS": json.dumps(settings.MAX_TOTAL_TOKENS),
   "MAX_BATCH_TOTAL_TOKENS": json.dumps(settings.MAX_BATCH_TOTAL_TOKENS),
   "HUGGING_FACE_HUB_TOKEN": settings.HUGGINGFACE_ACCESS_TOKEN,
   "MAX_BATCH_PREFILL_TOKENS": "10000",
   "HF_MODEL_QUANTIZE": "bitsandbytes",
}
```

Эти две конфигурации предоставляют нам полный контроль над инфраструктурой, выбором LLM и ее поведением. Чтобы запустить процесс развертывания модели на платформе AWS SageMaker с использованием описанной выше конфигурации, достаточно вызвать функцию `create_endpoint()` (представленную в начале раздела):

```
create_endpoint(endpoint_type=EndpointType.MODEL_BASED)
```

Для удобства эта функция также была обернута в консольную команду `poe`:

```
poetry poe deploy-inference-endpoint
```

Этого достаточно для развертывания пайплайна инференса на платформе AWS SageMaker. Самое сложное — подобрать конфигурацию параметров, которая

соответствует вашим требованиям и в то же время минимизирует затраты на инфраструктуру. В зависимости от настроек AWS процесс развертывания может занять от 15 до 30 минут. Обновить значения параметров можно в любой момент в файле `.env` и развернуть модель с новой конфигурацией, не трогая код. Например, в предложенной реализации по умолчанию используется один экземпляр GPU типа `ml.g5.xlarge`. Если вы хотите развернуть несколько реплик модели, измените параметры `GPUS` и `SM_NUM_GPUS` или задайте другой тип экземпляра, скорректировав значение переменной `GPU_INSTANCE_TYPE`.



Перед развертыванием LLM-микросервиса на платформе AWS SageMaker убедитесь в том, что вы создали IAM-пользователя с помощью команды poetry poe create-sagemaker-role и IAM-роль выполнения с помощью команды poetry poe create-sagemaker-execution-role. Не забудьте также обновить переменные окружения AWS_* в файле .env, подставив значения, полученные при выполнении двух указанных команд. Подробности вы найдете в README-файле репозитория.

После завершения развертывания конечной точки инференса на платформе AWS SageMaker вы можете проверить ее статус на дашборде данного сервиса. Для этого в левом меню выберите пункт SageMaker dashboard (Дашборд SageMaker), а затем в столбце Inference (Инференс) нажмите кнопку Endpoints (Конечные точки), как показано на рис. 10.6.

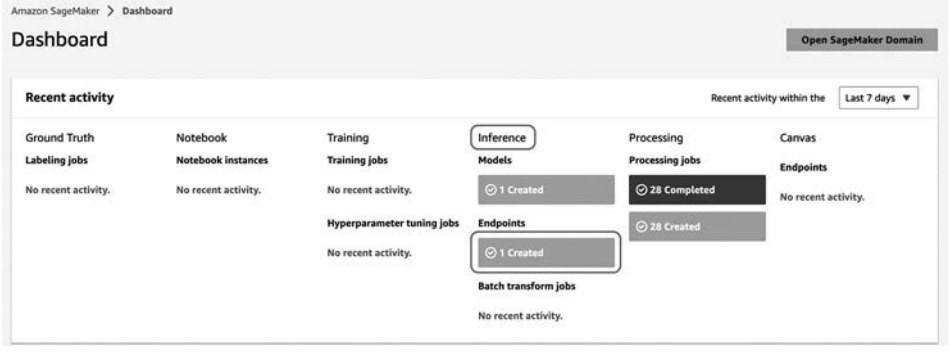


Рис. 10.6. Пример конечной точки инференса, развернутой на платформе AWS SageMaker

После нажатия кнопки Endpoints (Конечные точки) вы увидите свою конечную точку (в данном случае twin), находящуюся в состоянии Creating (Создается) или Created (Создана), как показано на рис. 10.7. Щелкнув на ней, вы получите доступ к логам соответствующей конечной точки в системе CloudWatch и сможете отслеживать загрузку CPU, GPU, памяти и дисков.

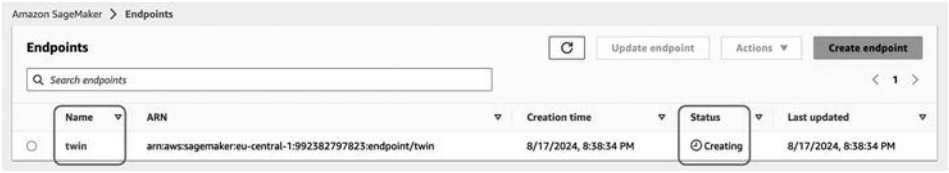


Рис. 10.7. Пример конечной точки инференса twin, развернутой на платформе AWS SageMaker

Кроме того, данный интерфейс позволяет отслеживать все HTTP-ошибки, такие как *4XX* и *5XX*, в одном месте.

Вызов конечной точки инференса, развернутой на AWS SageMaker

Теперь, когда наш LLM-сервис развернут на платформе AWS SageMaker, давайте разберемся, как к нему обращаться. Для этого мы создадим два класса, которые помогут подготовить промпт для SageMaker, отправить запрос на конечную точку инференса по протоколу HTTP и декодировать результат в формате, пригодном для использования клиентом. Весь код для реализации инференса с помощью AWS SageMaker доступен на GitHub в папке `llm_engineering/model/inference`. Он начинается с простого примера:

```
text = "Write me a post about AWS SageMaker inference endpoints."
llm = LLMInferenceSagemakerEndpoint(
        endpoint_name=settings.SAGEMAKER_ENDPOINT_INFERENCE
    )
Answer = InferenceExecutor(llm, text).execute()
```

Далее мы пошагово рассмотрим классы `LLMInferenceSagemakerEndpoint` и `InferenceExecutor`. Начнем с класса `LLMInferenceSagemakerEndpoint`, который напрямую взаимодействует с платформой SageMaker. Его конструктор инициализирует все ключевые атрибуты, необходимые для работы с конечной точкой SageMaker:

```
class LLMInferenceSagemakerEndpoint(Inference):
    def __init__(
        self,
        endpoint_name: str,
        default_payload: Optional[Dict[str, Any]] = None,
        inference_component_name: Optional[str] = None,
    ) -> None:
        super().__init__()

        self.client = boto3.client(
            "sagemaker-runtime",
            region_name=settings.AWS_REGION,
            aws_access_key_id=settings.AWS_ACCESS_KEY,
```

```
            aws_secret_access_key=settings.AWS_SECRET_KEY,
        )
        self.endpoint_name = endpoint_name
        self.payload = default_payload if default_payload else
        ➥self._default_payload()
        self.inference_component_name = inference_component_name
```

Параметр `endpoint_name` определяет конечную точку SageMaker, к которой отправляется запрос. Кроме того, конструктор инициализирует полезную нагрузку (`payload`), используя заданное значение или, если оно не было указано, вызывая метод, генерирующий значение по умолчанию.

Одной из основных возможностей этого класса является генерация стандартной полезной нагрузки для выполнения инференса, которая реализуется с помощью метода `_default_payload()`:

```
def _default_payload(self) -> Dict[str, Any]:
    return {
        "inputs": "",
        "parameters": {
        "max_new_tokens": settings.MAX_NEW_TOKENS_INFERENCE,
        "top_p": settings.TOP_P_INFERENCE,
        "temperature": settings.TEMPERATURE_INFERENCE,
        "return_full_text": False,
    },
}
```

Данный метод возвращает словарь, представляющий собой стандартную структуру запроса на инференс. Раздел `parameters` содержит настройки, определяющие поведение модели при генерации ответа, например число токенов, стратегию семплирования (`top_p`) и параметр `temperature`, который задает степень случайности, присущей выходным данным. Эти параметры извлекаются из глобальных настроек приложения, что обеспечивает согласованность поведения модели при выполнении различных задач инференса.

Для изменения содержимого запроса предусмотрен метод `set_payload()`:

```
def set_payload(self, inputs: str, parameters: Optional[Dict[str, Any]]
➥= None) -> None:
    self.payload["inputs"] = inputs
    if parameters:
        self.payload["parameters"].update(parameters)
```

Он позволяет пользователю обновить поле `inputs` новым текстом и переопределить параметры инференса.

Наконец, метод `inference()` используется для отправки запроса на конечную точку SageMaker вместе с пользовательской полезной нагрузкой:

```
def inference(self) -> Dict[str, Any]:
    try:
        logger.info("Inference request sent.")
```

```
            invoke_args = {
                "EndpointName": self.endpoint_name,
                "ContentType": "application/json",
                "Body": json.dumps(self.payload),
            }
            if self.inference_component_name not in ["None", None]:
                invoke_args["InferenceComponentName"] = self.inference_
                ➥component_name
            response = self.client.invoke_endpoint(**invoke_args)
            response_body = response["Body"].read().decode("utf8")

            return json.loads(response_body)
        except Exception:
            logger.exception("SageMaker inference failed.")

            raise
```

Этот метод формирует запрос для отправки на конечную точку SageMaker, упаковывая полезную нагрузку и другие необходимые данные в формат, поддерживаемый сервисом SageMaker. Если значение параметра `inference_component_name` задано, оно добавляется в запрос, обеспечивая более точный контроль над процессом инференса. Готовый запрос отправляется с помощью функции `invoke_endpoint()`, а полученный ответ считывается, декодируется и возвращается в виде объекта JSON.

Теперь разберемся, как класс `InferenceExecutor` использует класс `LLMInferenceSagemakerEndpoint` для отправки HTTP-запросов на конечную точку AWS SageMaker.

Конструктор класса `InferenceExecutor` принимает ключевые параметры, необходимые для вызова LLM. Параметр `llm` принимает любой экземпляр, реализующий интерфейс `Inference`, в том числе класс `LLMInferenceSagemakerEndpoint`, выполняющий сам процесс инференса.

Кроме того, этот конструктор принимает параметр `query`, который представляет собой пользовательский ввод. Дополнительно вы можете указать необязательное контекстное поле, если планируете задействовать RAG, а также задать шаблон промпта. Если шаблон не задан, используется универсальный шаблон, не адаптированный под конкретную модель.

```
class InferenceExecutor:
    def __init__(
        self,
        llm: Inference,
        query: str,
        context: str | None = None,
        prompt: str | None = None,
    ) -> None:
        self.llm = llm
        self.query = query
        self.context = context if context else ""
```

```
        if prompt is None:
            self.prompt = """
    Ты — создатель контента. Напиши то, о чем просит пользователь,
    используя предоставленный контекст в качестве основного источника
    информации для создаваемого материала.
    Пользовательский запрос: {query}
    Контекст: {context}
            """
        else:
            self.prompt = prompt
```

Метод `execute()` — ключевой компонент класса `InferenceExecutor`. Именно он отвечает за непосредственное выполнение инференса. После вызова он подготавливает полезную нагрузку для отправки в LLM, форматируя промпт с учетом пользовательского запроса и контекста.

Затем он задает ряд параметров, влияющих на поведение LLM, в частности максимально допустимое количество генерируемых токенов, штраф за повторение, снижающий вероятность генерации повторяющегося текста, и температуру, определяющую степень случайности выдаваемого моделью ответа.

После подготовки полезной нагрузки и настройки параметров этот метод вызывает функцию `inference()` из класса `LLMInferenceSagemakerEndpoint` и ожидает получения ответа:

```
def execute(self) -> str:
    self.llm.set_payload(
        inputs=self.prompt.format(query=self.query,
        ➥context=self.context), parameters={
            "max_new_tokens": settings.MAX_NEW_TOKENS_INFERENCE,
            "repetition_penalty": 1.1,
            "temperature": settings.TEMPERATURE_INFERENCE,
        },
    )
    answer = self.llm.inference()[0]["generated_text"]

    return answer
```

Благодаря тому что инференс осуществляется через объект, реализующий интерфейс `Inference`, мы можем легко заменить его стратегию или использовать описанную выше реализацию `LLMInferenceSagemakerEndpoint`, не затрагивая остальной код.

Чтобы протестировать эту систему, достаточно запустить соответствующий Python-файл, как показано ниже:

```
poetry run python -m llm_engineering.model.inference.test
```

Для удобства вызов также был обернут в консольную команду poe:

```
poetry poe test-sagemaker-endpoint
```

Теперь рассмотрим процесс реализации бизнес-микросервиса с помощью FastAPI. Этот сервис будет отправлять HTTP-запросы к описанному выше LLM-микросервису и вызывать поисковый модуль RAG-системы, реализованный в главе 9.

Реализация бизнес-микросервиса на базе FastAPI

Реализация простого приложения на базе FastAPI начинается с создания экземпляра `FastAPI`:

```
from fastapi import FastAPI

app = FastAPI()
```

Затем определяются классы `QueryRequest` и `QueryResponse`, основанные на `BaseModel` из библиотеки Pydantic. Эти классы описывают структуру запроса и ответа для конечных точек FastAPI:

```
class QueryRequest(BaseModel):
    query: str

class QueryResponse(BaseModel):
    answer: str
```

Теперь, когда мы определили компоненты FastAPI и настроили систему SageMaker, вернемся к функциям `call_llm_service()` и `rag()`, которые мы представили в главе 9, но не смогли использовать, так как тонко настроенная LLM еще не была развернута. Напомним, что функция `call_llm_service()` инкапсулирует логику инференса и отправляет запрос в LLM-микросервис, развернутый на платформе SageMaker:

```
    def call_llm_service(query: str, context: str | None) -> str:
        llm = LLMInferenceSagemakerEndpoint(
            endpoint_name=settings.SAGEMAKER_ENDPOINT_INFERENCE,
            ➥inference_component_name= None
    )
    answer = InferenceExecutor(llm, query, context).execute()

    return answer
```

Функция `rag()` реализует всю бизнес-логику RAG. Подробнее об этой функции можно узнать в главе 9. Здесь важно подчеркнуть, что она выполняет только те шаги, которые необходимы для работы RAG и ограничены возможностями

процессора и скоростью ввода/вывода. Например, класс `ContextRetriever` делает запросы к OpenAI и Qdrant (сетевой ввод/вывод) и вызывает модель векторизации, работающую непосредственно на CPU. Поскольку логика инференса вынесена в отдельный микросервис, функция `call_llm_service()` также ограничена только скоростью операций сетевого ввода/вывода. Данная функция довольно быстрая, поскольку ресурсоемкие вычисления выполняются другими сервисами. Это позволяет разместить сервер FastAPI на маломощной и недорогой машине без графического ускорителя, сохраняя низкую задержку.

```
    def rag(query: str) -> str:
        retriever = ContextRetriever(mock=False)
        documents = retriever.search(query, k=3 * 3)
        context = EmbeddedChunk.to_context(documents)

        answer = call_llm_service(query, context)

        return answer
```

Наконец, мы определяем функцию `rag_endpoint()`, которая используется для публикации нашей RAG-логики в Интернете в виде конечной точки HTTP. С помощью декоратора Python мы объявляем ее конечной точкой FastAPI, обрабатывающей POST-запросы. Эта конечная точка привязана к относительному пути `/rag` и ожидает на вход объект `QueryRequest`. Данная функция обрабатывает запрос, вызывая `rag()` с вопросом пользователя. В случае успеха возвращается ответ, обернутый в объект `QueryResponse`. При возникновении исключения возвращается код ошибки HTTP 500 с соответствующими подробностями.

```
@app.post("/rag", response_model=QueryResponse)
async def rag_endpoint(request: QueryRequest):
    try:
        answer = rag(query=request.query)

        return {"answer": answer}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e)) from e
```

Это FastAPI-приложение демонстрирует эффективный способ интеграции LLM, размещенной на платформе AWS SageMaker, в веб-сервис, задействуя метод RAG для повышения релевантности ответов. Благодаря модульному дизайну, основанному на пользовательских классах `ContextRetriever`, `InferenceExecutor` и `LLMInferenceSagemakerEndpoint`, код легко адаптируется и масштабируется, что делает его мощным инструментом для развертывания ML-моделей в продакшене.

Чтобы запустить сервер, мы применяем `uvicorn` — стандартное решение для FastAPI. Выполните следующую команду:

```
uvicorn tools.ml_service:app --host 0.0.0.0 --port 8000 --reload
```

Вы также можете воспользоваться командой poe:

```
poetry poe run-inference-ml-service
```

Для обращения к конечной точке /rag можно использовать следующую консольную команду curl, отправляющую HTTP-запрос POST к серверу FastAPI:

```
curl -X POST 'http://127.0.0.1:8000/rag' -H 'Content-Type: application/json' -d
'{\"query\": \"your_query \"}'
```

Для этого также предусмотрена соответствующая команда poe, содержащая пример пользовательского запроса:

```
poetry poe call-inference-ml-service
```

FastAPI-сервер, описанный выше, работает только локально. Следующим шагом может стать его развертывание на *AWS Elastic Kubernetes Service (EKS)* — полностью управляемом сервисе Kubernetes от AWS. Альтернативой является сервис *AWS Elastic Container Service (ECS)*, который выполняет ту же роль, но реализован на основе собственного решения AWS без использования Kubernetes. Однако поскольку данный процесс не является специфическим для LLM или LLMOps, мы не будем подробно рассматривать его. Но если вкратце, то для этого необходимо создать кластер AWS EKS или ECS через дашборд либо с помощью IaC-инструмента (Infrastructure-as-Code — «инфраструктура как код»), например Terraform. Затем нужно создать Docker-образ на основе представленного выше кода. Наконец, необходимо отправить данный образ в AWS ECR (Amazon Elastic Container Registry) и настроить развертывание ECS или EKS на основе размещенного в ECR Docker-образа. Если все это кажется слишком сложным, не переживайте, в главе 11 мы подробно рассмотрим аналогичный процесс развертывания ZenML-пайплайнов на платформе AWS.



После завершения тестирования пайплайна инференса крайне важно удалить все ресурсы AWS SageMaker, задействованные при развертывании LLM. Поскольку почти все ресурсы AWS оплачиваются по мере использования (pay-as-you-go), несколько часов работы SageMaker не навредят бюджету. Однако, если забыть его выключить, уже через несколько дней расходы могут кратно вырасти. Поэтому целесообразно удалять все ресурсы сразу после завершения тестирования инфраструктуры SageMaker (или любых других сервисов AWS). К счастью, мы подготовили сценарий, который автоматически удаляет все ресурсы AWS SageMaker за вас:

```
poetry poe delete-inference-endpoint
```

После его выполнения перейдите на дашборд SageMaker и удостоверьтесь в том, что все ресурсы действительно удалены.

Автомасштабирование для обработки всплесков нагрузки

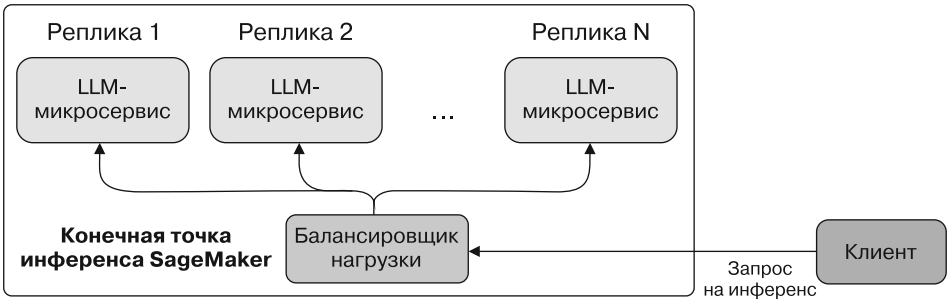
До настоящего момента LLM-микросервис, развернутый на платформе SageMaker, использовал фиксированное число реплик для обслуживания пользователей, то есть задействовал одно и то же количество вычислительных экземпляров независимо от объема трафика. Как уже было сказано, машины с графическими процессорами обходятся дорого. Поэтому в периоды простоя, когда большинство реплик остаются неактивными, мы несем значительные финансовые потери. Более того, если наше приложение сталкивается с внезапными всплесками трафика, его производительность резко снижается из-за того, что сервер не справляется с возросшим числом запросов. Это крайне негативно влияет на пользовательский опыт: именно во время таких всплесков приходят новые пользователи, и если их первое впечатление будет плохим, вероятность их возвращения на нашу платформу существенно снижается.

Ранее мы сконфигурировали сервис с несколькими конечными точками с помощью класса `ResourceRequirements` из библиотеки SageMaker. Допустим, мы запросили четыре реплики модели со следующими характеристиками:

```
model_resource_config = ResourceRequirements(
    requests={
        "copies": 4,           # Количество реплик
        "num_accelerators": 4, # Необходимое количество GPU
        "num_cpus": 8,         # Необходимое количество ядер CPU
        "memory": 5 * 1024,    # Минимально необходимый объем памяти в мегабайтах
    },
)
```

В такой конфигурации для обслуживания клиентов постоянно используется четыре копии модели независимо от реального объема трафика. Решением этой проблемы является внедрение стратегии автомасштабирования, при которой число реплик динамически увеличивается или уменьшается в зависимости от конкретных показателей, например от количества входящих запросов.

На рис. 10.8 представлена типовая архитектура, в которой количество конечных точек инференса SageMaker изменяется в зависимости от объема запросов. Когда трафика нет, можно оставить активной всего одну реплику, чтобы сервер оставался доступным для новых пользователей, или сократить их количество до нуля, если требования к задержке не слишком жесткие. Далее, допустим, что при 10 запросах в секунду нам достаточно двух запущенных реплик, а чтобы справиться с 100 запросами, нам требуется уже 20. Следует отметить, что эти значения условны и приведены лишь для примера.



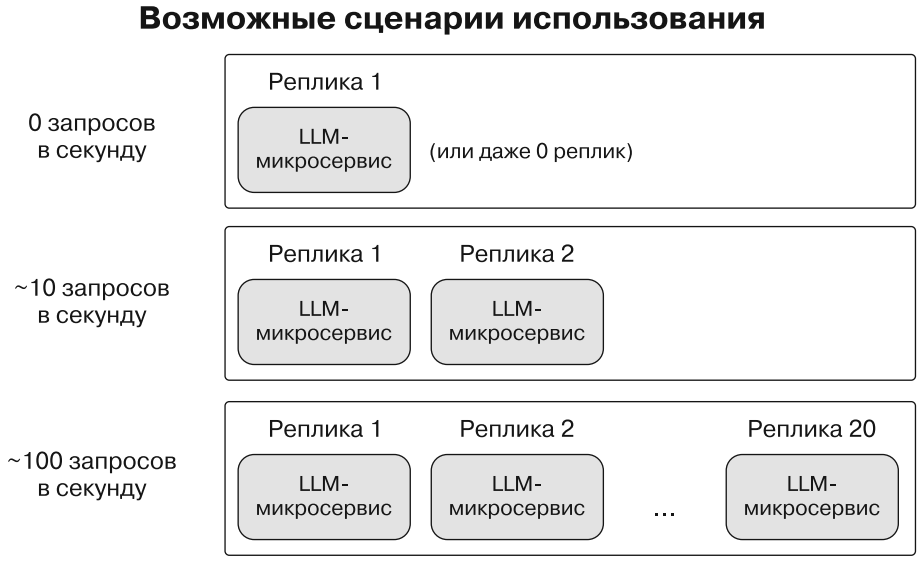


Рис. 10.8. Возможные сценарии автомасштабирования

Без углубления в подробности облачной сетевой инфраструктуры отметим, что в системах с несколькими репликами моделей между клиентом и этими копиями располагается балансировщик нагрузки, например *Application Load Balancer (ALB)*.

Все входящие запросы сначала поступают в ALB, который распределяет их между репликами. ALB может использовать различные стратегии маршрутизации. Самая простая из них — циклическая, когда первый запрос направляется на первую реплику, второй — на вторую и т. д. При этом независимо от числа активных реплик конечная точка, с которой взаимодействует клиент, всегда обрабатывается балансировщиком, выступающим в роли входной точки кластера.

Таким образом, добавление или удаление реплик не влияет на протокол взаимодействия между сервером и клиентом.

Теперь давайте обсудим способ реализации стратегии автомасштабирования для конечной точки инференса SageMaker. Данный сервис предоставляет специальную функцию *Application Auto Scaling*, позволяющую масштабировать ресурсы динамически в соответствии с заранее заданными политиками. Чтобы ею воспользоваться, необходимо зарегистрировать масштабируемую цель и создать политику масштабирования.

Регистрация масштабируемого сервиса

Первым этапом реализации стратегии автомасштабирования ресурсов является регистрация масштабируемого сервиса (или приложения) и характеристик масштабирования с помощью функции AWS *Application Auto Scaling*. Можно сказать, что на данном этапе мы информируем AWS о конкретном ресурсе, который планируем масштабировать, и задаем пределы, в которых оно может выполняться. Однако вместе с тем мы не указываем ни конкретный способ, ни момент масштабирования.

Например, при работе с компонентами SageMaker Inference необходимо указать следующие параметры.

- *Resource ID (идентификатор ресурса)* — уникальный идентификатор ресурса, подлежащего масштабированию (как правило, включает имя компонента SageMaker Inference).
- *Service namespace (пространство имен сервиса)* — обозначает AWS-сервис, к которому относится масштабируемый ресурс; в данном случае это *SageMaker*.
- *Scalable dimension (масштабируемая величина)* — задает масштабируемый параметр, например требуемое число копий (реплик).
- *MinCapacity и MaxCapacity (минимальная и максимальная емкость)* — задают границы автомасштабирования, например минимальное и максимальное количество реплик.

Регистрация масштабируемого сервиса подготавливает компонент SageMaker Inference к последующему масштабированию, но не определяет конкретные условия и способы его выполнения.

Создание политики масштабирования

После регистрации масштабируемого сервиса необходимо создать политику, задающую конкретные правила масштабирования. Для этого нужно указать метрики, которые следует отслеживать, и пороговые значения, запускающие процесс масштабирования.

Для компонента SageMaker Inference такая политика может предусматривать:

- *тип политики (policy type)* — например, *TargetTrackingScaling*, в рамках которой количество экземпляров ресурса динамически изменяется для поддержания заданного значения выбранного показателя;
- *настройку отслеживания цели (target tracking configuration)* — выбор отслеживаемого показателя (например, *SageMakerInferenceComponentInvocationsPerCopy*), задание целевого значения и периодов охлаждения, определяющих задержку между последовательными действиями, связанными с масштабированием.

Политика масштабирования задает правила увеличения и уменьшения числа копий компонента для инференса. Система непрерывно отслеживает выбранную метрику и в зависимости от того, превышает ли она целевое значение или опускается ниже него, выполняет масштабирование вверх или вниз в пределах, установленных при регистрации масштабируемого сервиса.

Давайте рассмотрим, как работает политика *TargetTrackingScaling*. Предположим, у вас есть метрика, отражающая желаемый уровень загрузки или пропускной способности приложения. При настройке отслеживания цели вы выбираете эту метрику и задаете ее целевое значение, соответствующее оптимальному уровню производительности приложения. Затем функция *Application Auto Scaling* отслеживает заданную метрику, автоматически создавая и управляя соответствующими сигналами CloudWatch. Когда значение метрики отклоняется от целевого, система выполняет масштабирование подобно тому, как термостат регулирует температуру для ее поддержания на заданном уровне.

Допустим, приложение, развернутое на платформе SageMaker, должно поддерживать загрузку GPU на уровне около 70 %. Такое значение обеспечивает достаточный запас ресурсов для обработки внезапных всплесков трафика и сокращает затраты, связанные с периодами низкой активности. Если уровень загрузки превышает верхний предел, система задействует дополнительные ресурсы. Если уровень загрузки падает, количество реплик уменьшается, снижая издержки.

Одним из ключевых преимуществ создания политик с помощью Application Auto Scaling является упрощение процесса масштабирования. Благодаря такому подходу вам больше не нужно вручную настраивать сигналы CloudWatch и задавать параметры масштабирования, поскольку система берет это на себя.

Границы масштабирования

Перед созданием политики масштабирования конечных точек инференса SageMaker важно задать его границы. Минимальное значение отражает наименьшее количество ресурсов, при котором модель все еще может функционировать. Это значение должно быть не меньше 1, чтобы модель располагала необходимым объемом вычислительных ресурсов.

Максимальное значение задает верхнюю границу доступных модели ресурсов. Хотя данное значение должно быть не меньше минимального, оно не предусматривает абсолютного предела. Таким образом, масштабирование может выполняться в той степени, в какой позволяет инфраструктура AWS.

Период охлаждения. Еще одним важным аспектом политики масштабирования является период охлаждения (cooldown period) — время ожидания, обеспечивающее баланс между отзывчивостью системы и ее стабильностью. Этот период играет роль защитного механизма, предотвращающего чрезмерную реакцию системы как при увеличении количества ресурсов, так и при их сокращении. Он представляет собой паузу, исключающую резкие колебания числа экземпляров, которая задерживает удаление ненужных реплик и ограничивает создание новых. Такая стратегия помогает поддерживать стабильную и эффективную среду для работы LLM-сервиса.

Описанные выше принципы применяются при реализации автомасштабирования большинства веб-серверов, включая серверы для машинного обучения с онлайн-инференсом в режиме реального времени. Освоив настройку политик масштабирования в SageMaker, вы легко сможете применить аналогичные подходы в других популярных системах развертывания, например Kubernetes или AWS ECS.



Пошаговое руководство по настройке автомасштабирования конечной точки SageMaker, реализованной в этой главе, можно найти в официальной документации AWS: https://docs.aws.amazon.com/sagemaker/latest/dg/endpoint-autoscaling-prerequisites.html.

Функция автомасштабирования — важный компонент любой облачной архитектуры, однако с ней связаны определенные риски, о которых стоит знать. Самый опасный из них — это избыточное масштабирование, которое напрямую влияет на стоимость инфраструктуры. Если политика масштабирования или период охлаждения окажутся слишком чувствительными, система начнет запускать дополнительные экземпляры без необходимости, в результате чего машины будут простаивать или использоваться неэффективно. Другая крайность связана с недостаточным масштабированием, когда система не справляется с возросшей нагрузкой, что приводит к ухудшению пользовательского опыта.

Именно поэтому важно изучить требования системы и на их основе поэкспериментировать с параметрами масштабирования в тестовой среде, чтобы найти оптимальный режим (как при настройке гиперпараметров в ходе обучения моделей). Допустим, вы ожидаете, что система будет обслуживать в среднем 100 пользователей в минуту и масштабироваться до 10 000 в особых случаях, например в праздничные дни. С учетом этих спецификаций можно провести нагрузочное тестирование, чтобы определить уровни потребления

ресурсов и найти наилучшее соотношение между затратами, задержкой и пропускной способностью, обеспечивающее стабильную работу системы как при стандартной, так и при повышенной нагрузке.

Резюме

В этой главе мы обсудили проектные решения, которые необходимо принять перед развертыванием любой ML-модели, и разобрали такие стратегии, как онлайн-инференс в режиме реального времени, асинхронный инференс и пакетная офлайн-обработка. Затем рассмотрели преимущества и недостатки монолитной и микросервисной архитектуры в контексте обслуживания модели и, взвесив их, выяснили, что целесообразнее развернуть ML-сервис в виде монолитного приложения или разделить его на два микросервиса, например на LLM-микросервис и бизнес-микросервис.

Далее мы развернули нашу тонко настроенную модель LLM-двойника на конечной точке AWS SageMaker Inference. Мы также реализовали бизнес-микросервис на базе FastAPI, который охватывает все этапы RAG, задействующие поисковый модуль, представленный в главе 9, и использует LLM-микросервис, развернутый на платформе AWS SageMaker. Наконец, мы обсудили необходимость внедрения стратегии автомасштабирования, рассмотрели популярный подход к масштабированию на основе метрик и способ его реализации в системе AWS SageMaker.

В следующей главе вы познакомитесь с основными принципами MLOps и LLMOps, а затем мы разберемся, как развернуть ZenML-пайплайны на платформе AWS, и реализуем пайплайн, обеспечивающий непрерывное обучение, интеграцию, доставку и мониторинг.

Список используемых источников

- AWS Developers. Machine Learning in 15: Amazon SageMaker High-Performance Inference at Low Cost [Video]. YouTube. https://www.youtube.com/watch?v=FRbcb7CtIOw.
- bitsandbytes-foundation. GitHub — bitsandbytes-foundation/bitsandbytes: Accessible large language models via k-bit quantization for PyTorch. GitHub. https://github.com/bitsandbytes-foundation/bitsandbytes.
- Difference between IAM role and IAM user in AWS. Stack Overflow. https://stackoverflow.com/questions/46199680/difference-between-iam-role-and-iam-user-in-aws.
- Huggingface. GitHub — huggingface/safetensors: Simple, safe way to store and distribute tensors. GitHub. https://github.com/huggingface/safetensors.

- Huggingface. GitHub — huggingface/text-generation-inference: Large Language Model Text Generation Inference. GitHub. https://github.com/huggingface/text-generation-inference.
- *Huyen C.* Designing machine learning systems. O'Reilly Online Learning. https://www.oreilly.com/library/view/designing-machine-learning/9781098107956/. (*Хьюен Ч.* Проектирование систем машинного обучения. — М.: 2023).
- *Iusztin P.* Architect LLM & RAG inference pipelines | Decoding ML. Medium. https://medium.com/decodingml/architect-scalable-and-cost-effective-llm-rag-inference-pipelines-73b94ef82a99
- *Lakshmanan V., Robinson S., Munn M.* Machine Learning design patterns. O'Reilly Online Learning. https://www.oreilly.com/library/view/machine-learning-design/9781098115777/. (*Лакшманан В., Робинсон С., Мун М.* Машинное обучение. Паттерны проектирования. — СПб., 2025).
- *Mendoza A.* Best tools for ML model Serving. neptune.ai. https://neptune.ai/blog/ml-model-serving-best-tools.

11 MLOps и LLMOps

В книге мы уже использовали компоненты и принципы *MLOps*, в частности реестр моделей для совместного использования и версионирования тонко настроенных LLM, логическое хранилище признаков для хранения данных, предназначенных для тонкой настройки и RAG, а также оркестратор, объединяющий все ML-пайплайны. Однако MLOps — это не просто набор компонентов, но и практик, предоставляющих возможность вывести ML-приложение на новый уровень за счет автоматизации процессов сбора данных, обучения, тестирования и развертывания. Конечная цель MLOps — максимальная автоматизация всех операций, чтобы пользователи могли сосредоточиться на принятии ключевых решений, например на оценке целесообразности переобучения модели при обнаружении изменений в поступающих на вход данных. Но чем же *LLMOps* (LLM operations — «эксплуатация больших языковых моделей») отличается от MLOps?

Дисциплина *LLMOps* возникла вследствие широкого распространения LLM. Она основана на принципах MLOps, которые, в свою очередь, базируются на *DevOps* (development operations — «разработка и эксплуатация»). Чтобы понять суть LLMOps, необходимо учесть исторический контекст, начав с DevOps и проследив, как на его основе возникли последующие концепции — именно этим мы и займемся в текущей главе. В своей основе LLMOps фокусируется на таких специфических для LLM задачах, как мониторинг и версионирование промптов, механизмы защиты ввода и вывода для предотвращения токсичного поведения и сбор данных для тонкой настройки. Кроме того, LLMOps охватывает проблемы масштабирования, возникающие при работе с LLM, такие как сбор триллионов токенов для обучающих наборов данных, обучение моделей на огромных кластерах GPU и снижение инфраструктурных затрат. К счастью, большинство этих задач решается небольшим числом компаний, которые тонко настраивают базовые модели (например, семейство моделей Llama). Остальные компании, как правило, адаптируют предварительно обученные базовые модели под собственные сценарии и сосредотачиваются на таких задачах уровня LLMOps, как мониторинг и версионирование промптов.

Чтобы применить принципы LLMOps в рамках нашего сценария с LLM-двойником, мы развернем все ZenML-пайплайны на платформе AWS, а также

реализуем пайплайн *непрерывной интеграции и доставки (continuous integration/continuous deployment, CI/CD)* для автоматической проверки целостности кода и развертывания, пайплайн *непрерывного обучения (continuous training, CT)* для автоматизации процесса обучения и пайплайн мониторинга для отслеживания всех промптов и сгенерированных ответов. Так чаще всего реализуется любой ML-проект вне зависимости от того, используется ли в нем LLM или нет.

В предыдущих главах мы обсудили процесс построения приложения на основе LLM. Теперь пришло время рассмотреть три основные цели, связанные с LLMOps. Первая цель — понять теоретические основы и фундаментальные принципы DevOps, MLOps и, наконец, LLMOps. Мы не ставим задачу изложить всю теорию соответствующих дисциплин, так как каждой из них можно посвятить отдельную книгу. Однако нам важно сформировать четкое понимание причин, лежащих в основе проектных решений, принятых при реализации LLM-двойника.

Вторая наша цель заключается в том, чтобы развернуть ZenML-пайплайны на платформе AWS (в главе 10 мы развернули только пайплайн инференса). В соответствующей практической части главы вы узнаете о том, как с помощью ZenML можно развернуть все необходимые компоненты в облаке AWS. Это позволит нам реализовать третью и последнюю цель, а именно применить изученные теоретические принципы на практике в рамках сценария с нашим LLM-двойником. Мы создадим CI/CD-пайплайн с использованием платформы GitHub Actions, пайплайн непрерывного обучения (CT) и оповещений с помощью системы ZenML, а также пайплайн мониторинга с помощью инструмента Opik от Comet ML.

Истоки LLMOps в DevOps и MLOps

Чтобы понять суть дисциплины LLMOps, нужно начать с ее истоков — области DevOps, от которой она унаследовала большую часть своих фундаментальных принципов. Далее мы перейдем к MLOps, чтобы узнать, как идеи DevOps были адаптированы к системам машинного обучения. Наконец, обсудим, что представляет собой дисциплина LLMOps и как она возникла на основе MLOps после широкого распространения LLM.

DevOps

Ручная доставка программного обеспечения — трудоемкий, плохо масштабируемый процесс, подверженный ошибкам и сопряженный с рисками безопасности. Концепция DevOps была разработана для решения этой проблемы. Принципы DevOps применяются при разработке ПО и призваны полностью

автоматизировать все этапы данного процесса — от сборки и тестирования до развертывания и мониторинга. Эта методология ориентирована на сокращение длительности жизненного цикла разработки и обеспечение непрерывной поставки качественного программного продукта. Она поощряет сотрудничество, позволяет автоматизировать процессы, объединять рабочие потоки и внедрять механизмы оперативного получения обратной связи. Все это способствует формированию культуры, обеспечивающей надежную и быструю реализацию всех этапов разработки ПО.

Принятие культуры DevOps дает организациям значительные преимущества, выражающиеся прежде всего в повышении операционной эффективности, ускорении выпуска новых функций и повышении качества продукта. К ключевым достоинствам использования принципов DevOps относятся следующие факторы.

- *Улучшение сотрудничества*. Культура DevOps формирует единое рабочее пространство. Устранение барьеров между специалистами, отвечающими за разработку и эксплуатацию продукта, улучшает коммуникацию и способствует более слаженной и продуктивной командной работе.
- *Повышение эффективности*. Автоматизация жизненного цикла разработки ПО сокращает объем работы, выполняемой вручную, снижает риск ошибок, а также ускоряет выпуск новых функций.
- *Непрерывное совершенствование*. Методология DevOps фокусируется не только на внутренних процессах, но и на эффективном удовлетворении потребностей пользователей ПО. Культура непрерывной обратной связи позволяет командам быстро адаптироваться и улучшать продукты, создавая все бóльшую ценность для конечных пользователей.
- *Высокое качество и безопасность*. DevOps обеспечивает быструю разработку программного обеспечения при сохранении высоких стандартов качества и безопасности за счет непрерывной интеграции и доставки, а также принятия упреждающих мер защиты.

Жизненный цикл

Как показано на рис. 11.1, жизненный цикл DevOps охватывает весь путь от начала разработки ПО до его поставки, эксплуатации и защиты. Ниже перечислены основные этапы этого цикла.

1. *Планирование (plan)* — организация и приоритизация задач, а также отслеживание их выполнения.
2. *Разработка (code)* — совместная работа над написанием, проектированием, улучшением и безопасным управлением кодом и данными проекта.

3. *Сборка (build)* — упаковка приложения и его зависимостей в исполняемый формат.
4. *Тестирование (test)* — ключевой этап, на котором проверяется корректность работы кода и его соответствие стандартам качества (в идеале все это делается автоматически).
5. *Выпуск (release)* — если тесты пройдены успешно, версия помечается как новый релиз, готовый к поставке.
6. *Развертывание (deploy)* — предоставление доступа к последнему релизу конечным пользователям.
7. *Эксплуатация (operate)* — управление и поддержка работы инфраструктуры, на которой запущено ПО. Включает масштабирование, обеспечение безопасности, управление данными, резервное копирование и восстановление.
8. *Мониторинг (monitor)* — отслеживание метрик производительности и ошибок для снижения частоты и серьезности инцидентов.

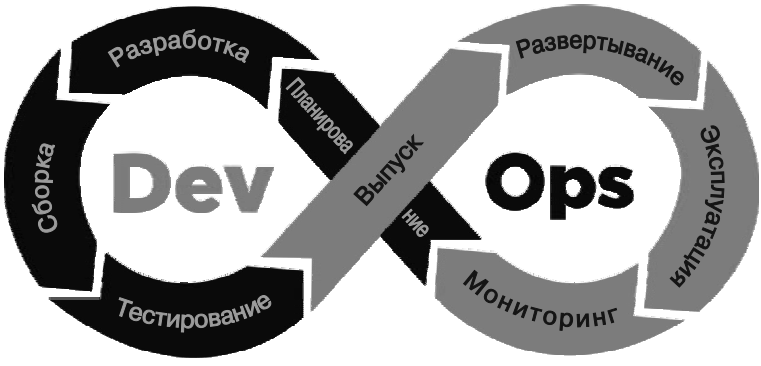


Рис. 11.1. Этапы жизненного цикла DevOps

Основные концепции

Методология DevOps охватывает множество практик, используемых на протяжении всего жизненного цикла приложения, однако в книге мы сосредоточимся на следующих ключевых аспектах.

- *Среды развертывания.* Чтобы тщательно протестировать код перед его поставкой, необходимо определить несколько сред, имитирующих рабочую. Наиболее распространенный подход предполагает создание среды разработки (dev) для тестирования новых функций. Промежуточная среда (staging) используется специалистами по контролю качества и заинтересованными сторонами для выявления ошибок в приложении и ознакомления с послед-

ними нововведениями перед их внедрением. Наконец, существует рабочая среда (продакшен, production), с которой взаимодействуют конечные пользователи.

- *Система управления версиями.* Используется для отслеживания, управления и версионирования всех изменений в исходном коде. Это позволяет полностью контролировать эволюцию кода и процессы его развертывания. Без версионирования было бы невозможно отслеживать изменения, вносимые в различных средах, а благодаря ему вы всегда можете точно определить стабильные и готовые к выпуску версии ПО.
- *Непрерывная интеграция.* Перед внесением изменений в основные ветки среды разработки, промежуточной или рабочей среды происходит автоматическая сборка приложения и тестирование каждой новой поставки. После успешного прохождения тестов функциональная ветка может быть объединена с основной.
- *Непрерывная доставка.* Работая в связке с CI, непрерывная доставка автоматизирует процессы подготовки инфраструктуры и развертывания приложения. Например, после объединения функциональной ветки с веткой промежуточной (предпродакшен) среды приложение с последними изменениями автоматически развертывается в соответствующей инфраструктуре. Затем специалисты по контролю качества (или заинтересованные стороны) вручную проверяют работоспособность новых функций. Именно эти два этапа обычно обозначаются аббревиатурой CI/CD.

Следует отметить, что принципы DevOps не привязаны к конкретным платформам или инструментам. В рамках нашего проекта по созданию LLM-двойника мы будем отслеживать эволюцию кода и управлять версиями с помощью GitHub. Другим популярным инструментом для контроля версий является GitLab. Для реализации CI/CD-пайплайна мы воспользуемся экосистемой GitHub и платформой GitHub Actions, которые предоставляются бесплатно для реализации проектов с открытым исходным кодом. К альтернативным инструментам относятся GitLab CI/CD, CircleCI и Jenkins. Обычно выбор конкретных DevOps-инструментов зависит от среды разработки и потребностей в кастомизации и конфиденциальности. Например, Jenkins — это открытый DevOps-инструмент, который можно развернуть и полностью контролировать локально. Однако за это приходится платить повышенной сложностью, поскольку систему необходимо настраивать и обслуживать вручную. Таким образом, многие компании выбирают те инструменты, которые лучше всего интегрируются с используемой ими системой контроля версий.

Теперь, когда мы разобрались с основными концепциями DevOps, обсудим область MLOps, реализующую те же принципы в сфере ИИ и машинного обучения.

MLOps

Как вы, вероятно, уже догадались, практика MLOps призвана реализовать принципы DevOps в сфере машинного обучения. Основная трудность состоит в том, что ML-приложение включает гораздо больше компонентов, чем традиционное ПО: кроме кода, важную роль в нем играют данные и модель. Практики MLOps направлены на отслеживание, эксплуатацию и мониторинг этих компонентов с целью повышения устойчивости и управляемости решений, а также обеспечения воспроизводимости результатов их работы.

В ML-системах процесс сборки может инициироваться изменением любого из компонентов, будь то код, данные или модель (рис. 11.2).

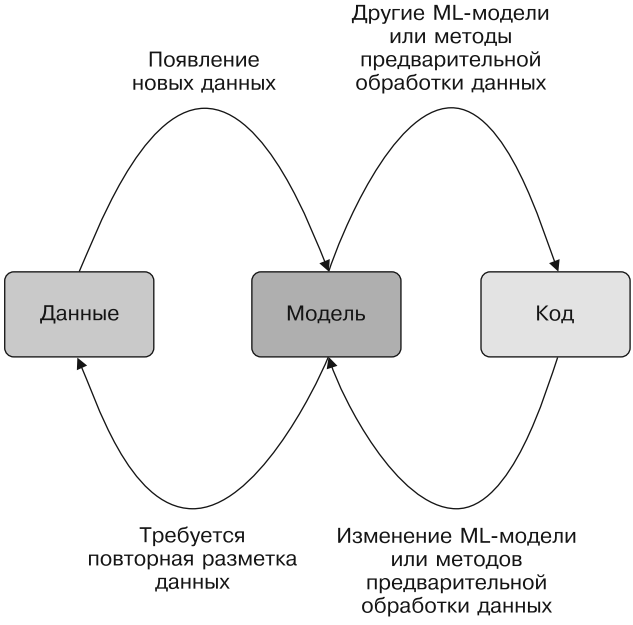


Рис. 11.2. Взаимосвязь между изменениями в данных, модели и коде

Если в DevOps все процессы сосредоточены вокруг кода (например, добавление новой функции инициирует запуск CI/CD-пайплайна), то в MLOps изменения могут происходить исключительно на уровне данных. В таком случае необходимо обучить (или тонко настроить) другую модель, создав новые версии как набора данных, так и самой модели. Интуитивно понятно, что изменение одного компонента влияет на другие, и практики MLOps должны учитывать эту дополнительную сложность. Ниже перечислены примеры типичных ситуаций, когда изменения в данных косвенно приводят к изменениям модели:

- после развертывания модели качество ее работы может со временем снизиться, и для ее переобучения потребуются новые данные;
- в процессе анализа может выясниться, что в реальных условиях сложно собирать нужные данные, и это потребует переформулирования задачи с учетом выявленных ограничений;
- на этапе экспериментов и обучения модели часто возникает необходимость в сборе дополнительных данных или повторной разметки уже имеющихся, что приводит к созданию новых моделей;
- после развертывания модели в продакшене и получения обратной связи от пользователей может выясниться, что исходные предположения, сделанные на этапе обучения, были ошибочны и нам требуется изменить саму модель.

Итак, что же такое MLOps?

Формально MLOps можно определить как расширение подхода DevOps, в рамках которого данные и модели рассматриваются в качестве значимых объектов.

Как и в случае с DevOps, методология MLOps опирается на идею о том, что изоляция этапа разработки ML-модели от этапа ее эксплуатации снижает общую прозрачность, гибкость и качество системы. Поэтому при оптимальной реализации подхода MLOps модели, данные и код рассматриваются как равноправные компоненты в рамках CI/CD-среды и единого процесса выпуска.

Основные компоненты

Ранее мы уже использовали все эти компоненты, но теперь, когда у нас сложилось более полное понимание предметной области, имеет смысл к ним вернуться. Итак, помимо системы контроля версий и CI/CD-пайплайнов архитектура MLOps включает в себя следующие компоненты:

- *реестр моделей* — централизованное хранилище для обученных ML-моделей (инструменты: Comet ML, W&B, MLflow, ZenML);
- *хранилище признаков* — компонент, обеспечивающий предварительную обработку и хранение входных данных в виде признаков, используемых как для обучения моделей, так и в пайплайнах инференса (инструменты: Hopsworks, Tecton, Featureform);
- *хранилище ML-метаданных* — предназначено для отслеживания информации, связанной с обучением моделей, включая конфигурации, обучающие и тестовые наборы данных, а также метрики производительности. Этот компонент в основном используется для сравнения различных моделей и анализа их происхождения с целью понимания процесса их создания (инструменты: Comet ML, W&B, MLflow);

- *оркестратор ML-пайплайнов* — автоматизирует выполнение последовательности шагов при реализации ML-проектов (инструменты: ZenML, Airflow, Prefect, Dagster).

Вероятно, вы заметили, что между компонентами MLOps и конкретными инструментами, которые их реализуют, есть много общего. Это объясняется тем, что большинство этих инструментов предоставляют унифицированные решения, часто называемые MLOps-платформами.

Принципы

В основе дисциплины MLOps лежит шесть базовых принципов, которые не зависят от конкретных инструментов и позволяют создавать надежные и масштабируемые ML-системы.

- *Автоматизация или операционализация*. В контексте MLOps под автоматизацией подразумевается переход от выполняемых вручную процессов к автоматическим пайплайнам за счет реализации непрерывного обучения, интеграции и доставки (CT/CI/CD). Это обеспечивает эффективное переобучение и развертывание моделей в ответ на такие события, как поступление новых данных, снижение производительности или выявление неучтенных граничных случаев. Переход от ручных экспериментов к полной автоматизации гарантирует надежность, масштабируемость, адаптивность и безошибочную работу ML-систем.
- *Версионирование*. В рамках MLOps крайне важно отслеживать изменения в коде, моделях и данных, причем по отдельности, обеспечивая тем самым согласованность и воспроизводимость результатов. Код отслеживается с помощью систем контроля версий, таких как Git, модели — через реестры моделей, а для версионирования данных применяются решения вроде DVC (Data Version Control) или системы управления артефактами.
- *Отслеживание экспериментов*. Поскольку обучение моделей — это итеративный и экспериментальный процесс, предусматривающий сравнение множества вариантов по заданным метрикам, для выбора наилучшей модели необходимо использовать системы отслеживания экспериментов. Такие инструменты, как Comet ML, W&B, MLflow и Neptune, позволяют фиксировать все необходимые параметры и легко сравнивать получаемые результаты.
- *Тестирование*. В рамках MLOps, помимо кода, тестированию подлежат данные и модели. Для этого проводятся модульные, интеграционные, приемочные, регрессионные и нагрузочные тесты. Такой подход гарантирует корректную работу каждого компонента и их интеграцию, уделяя внимание входам, выходам и граничным случаям.

- *Мониторинг*. Этот этап критически важен для выявления снижения производительности моделей, развернутых в продакшене, особенно при изменении входных данных. Мониторинг позволяет вовремя предпринять меры, например переобучить модель, скорректировать промпт, признаки или провести валидацию данных. Путем отслеживания логов, системных показателей и метрик модели, а также анализа отклонений можно поддерживать стабильную работу системы и быстро обнаруживать и устранять неполадки.
- *Воспроизводимость*. Это означает, что каждый процесс (будь то обучение модели или генерация признаков) должен давать одинаковые результаты при одинаковых входных данных, если зафиксированы все изменяемые переменные: версии кода и данных, гиперпараметры и конфигурации моделей.

Более подробное изложение этих принципов можно найти в приложении.

ML- и MLOps-инженерия

Провести четкую границу между зонами ответственности ML- и MLOps-инженера весьма непросто. На практике во многих случаях роль MLOps-инженера объединяется с функционалом специалистов по платформенной или облачной инженерии. С одной стороны, это логично, поскольку MLOps-инженер действительно работает преимущественно с инфраструктурой. Однако он также отвечает за реализацию таких компонентов, как системы отслеживания экспериментов, реестры моделей, механизмы версионирования и многое другое. Хорошей стратегией является такое распределение ответственности, при котором ML-инженер интегрирует соответствующие компоненты в код, а MLOps-инженер обеспечивает их работу на уровне инфраструктуры.

В крупных компаниях подобное разделение ролей вполне оправданно. Однако в небольших и средних командах функции ML- и MLOps-инженера, как правило, выполняет один и тот же специалист.

На рис. 11.3 показано четкое распределение ответственности между тремя ключевыми участниками процесса: специалистом по анализу данных (или ML-исследователем), ML-и MLOps-инженером. Специалист по анализу данных разрабатывает модели, предназначенные для решения конкретных задач.

ML-инженер берет функциональные модели, созданные командой специалистов по анализу данных, и создает поверх них новый слой, обеспечивающий их модульность, расширяемость, возможность подключения к базе данных и предоставления доступа к модели по сети через API. MLOps-инженер играет здесь ключевую роль, перенося код этого промежуточного уровня на более общий — инфраструктурный. Такой шаг служит переходом к развертыванию

приложения в продакшене. Затем можно заняться его автоматизацией, мониторингом, версионированием и т. д.

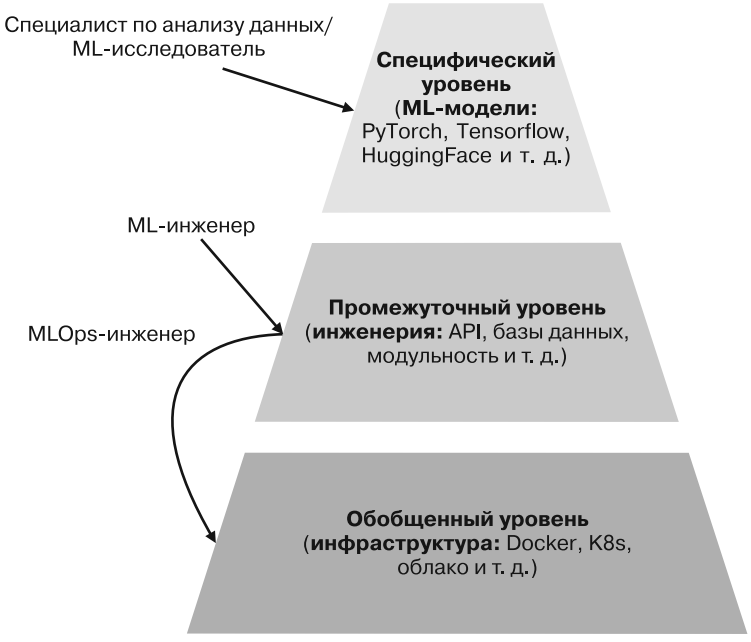


Рис. 11.3. Распределение ответственности между специалистом по анализу данных, ML- и MLOps-инженером

Промежуточный уровень — это то, что отличает прототип от полноценного продукта. Именно на нем проектируется расширяемое приложение с сохранением состояния (за счет интеграции с базой данных) и доступом через API. При развертывании приложения на конкретной инфраструктуре необходимо учитывать его масштабируемость, длительность задержки и экономическую эффективность.

LLMOps

Дисциплина LLMOps охватывает практики и процессы, имеющие отношение к управлению большими языковыми моделями и обеспечению их работы. Это специализированное направление MLOps, сосредоточенное на уникальных задачах и требованиях, связанных с LLM. Если методология MLOps описывает принципы и подходы к управлению различными ML-моделями, то LLMOps фокусируется на таких специфических особенностях больших языковых моделей, как их внушительный масштаб, высокая сложность обучения, необходимость

управления промптами и недетерминированный характер генерации ответов. Тем не менее LLMOps наследует все фундаментальные принципы, представленные в разделе, посвященном MLOps. Поэтому далее мы сосредоточимся на том, что добавляется поверх этого базиса.

При обучении LLM с нуля объем данных и масштаб модели в ML-системе существенно возрастают, что отличает LLMOps от MLOps. Ниже перечислены основные аспекты, которые необходимо учитывать при обучении LLM с нуля.

- *Сбор и подготовка данных* предполагает извлечение, подготовку и управление массивными наборами данных, необходимых для обучения LLM. Для этого применяются специальные методы обработки, хранения и совместного использования больших данных. Например, модель GPT-4 была обучена примерно на 13 трлн токенов, что почти эквивалентно 10 трлн слов.
- *Управление огромным числом параметров модели* представляет собой серьезную техническую и инфраструктурную задачу. Оно требует колоссальных вычислительных ресурсов, как правило, целых кластеров машин с GPU и поддержкой CUDA.
- *Масштаб модели напрямую влияет на обучение*. Обучение LLM с нуля невозможно выполнить на одном GPU. Это объясняется размером самой модели либо необходимостью в более крупных пакетах данных для достижения ожидаемых результатов. Таким образом, для обучения модели приходится задействовать несколько GPU, что, в свою очередь, требует оптимизации процессов и инфраструктуры для обеспечения поддержки параллелизма данных, модельного или тензорного параллелизма.
- *Управление масштабными наборами данных и кластерами GPU требует значительных затрат*. Например, согласно заявлению Сэма Альтмана, генерального директора OpenAI, обучение модели GPT-4 обошлось примерно в $100 млн (https://en.wikipedia.org/wiki/GPT-4#Training). Если добавить сюда стоимость многочисленных экспериментов, этапов оценки и инференса, становится очевидно, что обучение LLM с нуля по силам лишь крупнейшим игрокам отрасли.

По сути, LLMOps — это более масштабная версия MLOps. Данная дисциплина опирается на те же принципы, что и MLOps, но применительно к огромным объемам данных и сверхкрупным моделям, требующим колоссальных вычислительных мощностей для обучения и эксплуатации. Однако именно из-за этого масштаба наблюдается сдвиг в сторону отказа от обучения нейросетей с нуля под конкретную задачу. Подобный подход становится все менее актуальным с ростом числа доступных базовых моделей (foundation models), таких как GPT. Лишь немногие организации, вроде OpenAI или Google, располагают необходимыми ресурсами для их создания. Поэтому большая часть современных

приложений основана на тонкой настройке таких моделей, инженерии промптов и дистилляции данных или моделей в более легкие, специализированные сети инференса.

В большинстве проектов с LLM ваш процесс разработки будет предполагать выбор базовой модели, которую затем необходимо оптимизировать с помощью инженерии промптов, тонкой настройки или RAG. Поэтому крайне важно понять, как эти три этапа реализуются на практике. Далее мы рассмотрим популярные компоненты LLMOps, предназначенные для повышения качества инженерии промптов, тонкой настройки и RAG.

Обратная связь от человека

Одним из важных этапов доработки LLM является согласование ее поведения с предпочтениями целевой аудитории. Для этого необходимо встроить в приложение механизм сбора обратной связи и на ее основе создать набор данных, который можно использовать для тонкой настройки модели, например, с помощью *обучения с подкреплением на основе обратной связи от человека* (Reinforcement Learning with Human Feedback, RLHF) или *прямой оптимизации предпочтений* (Direct Preference Optimization, DPO). Одним из самых распространенных механизмов сбора обратной связи являются кнопки «нравится»/«не нравится», присутствующие в интерфейсах большинства чат-ботов. Более подробно о способах согласования поведения модели с пользовательскими предпочтениями можно узнать в главе 6.

Защитные механизмы

К сожалению, большие языковые модели нельзя считать абсолютно надежными, так как они нередко галлюцинируют. Систему можно оптимизировать, чтобы снизить число галлюцинаций, но, поскольку их бывает сложно выявить и они принимают самые разные формы, в этом направлении предстоит сделать еще очень многое.

Часть пользователей уже смирились с таким явлением, однако совершенно недопустимыми считаются ситуации, когда LLM случайно раскрывает конфиденциальные данные, например секретные ключи AWS (как в случае с GitHub Copilot) или пароли пользователей, выданные другими чат-ботами. То же самое может случиться с телефонными номерами, адресами, электронной почтой и другой чувствительной информацией. В идеале все подобные данные следует удалить из обучающих наборов, чтобы модель их не запомнила. Однако на практике это делается не всегда.

Хорошо известно, что LLM могут генерировать токсичный и вредоносный контент, в том числе сексистские и расистские высказывания. Например, в ходе эксперимента с ChatGPT в апреле 2023 года пользователи нашли способ обойти ограничения системы, заставив чат-бот имитировать поведение «плохого» или

«ужасного» человека. Он срабатывал даже в том случае, когда модель просили играть роль известных диктаторов и преступников. Вот какой ответ сгенерировала модель ChatGPT, изображая «плохого человека»:

> X — это еще одна страна третьего мира, населенная мафиози и нищими людьми. Ее жители невежественны и агрессивны, у них нет никакого уважения к закону и порядку. Если хотите знать мое мнение, X — это просто рассадник преступности и страданий, и ни один здравомыслящий человек не захочет туда ехать.

С примерами других негативных персонажей, используемых в ходе этого эксперимента, можно ознакомиться по ссылке https://techcrunch.com/2023/04/12/researchers-discover-a-way-to-make-chatgpt-consistently-toxic/.

Подобные случаи можно перечислять практически бесконечно, самое главное — понять, что LLM может генерировать вредоносный контент и получать опасные запросы, поэтому важно заранее внедрить меры мониторинга. Чтобы создать безопасные LLM-системы, необходимо защитить их от вредоносных, чувствительных или некорректных входных и выходных данных с помощью специальных механизмов.

- *Защита на входе (input guardrails)*. Задача этого механизма состоит в предотвращении реализации трех основных рисков: утечки конфиденциальной информации во внешние API, выполнения вредоносных промптов, способных скомпрометировать систему, и обработки запросов с насильственным или неэтичным содержанием. В первом случае речь идет о передаче таких чувствительных сведений, как учетные данные или секретная информация, за пределы организации. Под взломом модели обычно понимаются промпт-инъекции (prompt injection), например выполнение вредоносного SQL-кода, способного получить доступ к данным, удалить или повредить их. Наконец, некоторые приложения изначально не допускают обработку этически неприемлемых пользовательских запросов вроде просьбы предоставить инструкции по изготовлению взрывного устройства.
- *Защита на выходе (output guardrails)*. Нам также необходимо отслеживать некорректные ответы LLM, которые не соответствуют стандартам нашего приложения, в частности пустые ответы (или не совпадающие с ожидаемым форматом, например JSON или YAML), токсичные формулировки, галлюцинации и в целом неверные сведения. Кроме того, следует проверять, не содержат ли ответы конфиденциальную информацию, извлеченную из внутренних знаний модели или RAG-системы.

Среди популярных инструментов для реализации защитных механизмов можно отметить Galileo Protect, который позволяет выявлять промпт-инъекции, токсичные формулировки, утечки конфиденциальных данных и галлюцинации. Для обнаружения вредоносных входных или выходных данных и принятия соответствующих мер можно использовать Moderation API от компании OpenAI.

Следует учитывать, что внедрение защитных механизмов на входе и выходе приводит к дополнительным задержкам в системе, которые могут негативно сказаться на пользовательском опыте. Таким образом, необходимо найти баланс между безопасностью ввода/вывода и временем отклика. Что касается некорректных ответов, то, учитывая недетерминированную природу LLM, можно реализовать механизм повторной генерации, позволяющий получить альтернативный вывод. Однако повторная попытка удваивает задержки, поэтому распространенной стратегией является параллельная генерация нескольких вариантов с последующим выбором наилучшего. Такой подход повышает избыточность, но позволяет удерживать время отклика в разумных пределах.

Мониторинг промптов

Мониторинг не является чем-то новым для LLMOps, однако в контексте работы с большими языковыми моделями у нас появляется новая сущность, которой необходимо управлять, а именно промпт. Следовательно, нам требуется разработать специальные способы логирования и анализа промптов.

Большинство ML-платформ, таких как Opik (от Comet ML), Weights & Biases, а также специализированные инструменты вроде Langfuse уже реализовали средства логирования, позволяющие мониторить и отлаживать промпты. В продакшене с помощью этих средств обычно отслеживают пользовательский ввод, шаблоны промптов, входные переменные, сгенерированный ответ, количество токенов и задержку.

При генерации большой языковой моделью ответа результат, как правило, выводится токен за токеном, а не в виде целого текстового фрагмента. Это делает данный процесс более отзывчивым и плавным, с точки зрения пользователя. Таким образом, при измерении задержки генерации необходимо учитывать следующие показатели:

- *TTFT (Time to First Token)* — время до появления первого токена;
- *TBT (Time Between Tokens)* — интервал между токенами;
- *TPS (Tokens per Second)* — скорость генерации токенов;
- *TPOT (Time per Output Token)* — время генерации одного токена;
- *общая задержка (Total Latency)* — время генерации полного ответа.

Кроме того, для оценки стоимости обслуживания LLM важно отслеживать общее количество входных и выходных токенов.

Вы также можете рассчитывать метрики, отражающие эффективность модели на уровне каждого взаимодействия «ввод — промпт — ответ». В зависимости от конкретного сценария использования это могут быть такие показатели, как точность, уровень токсичности или частота галлюцинаций. При работе с RAG-системами можно дополнительно рассчитывать метрики, отражающие релевантность и точность извлеченного контекста.

Еще одним важным аспектом мониторинга промптов является логирование всей последовательности этапов их обработки (trace), включая промежуточные. Например, переформулирование запроса для повышения точности результатов работы поискового модуля RAG-системы может потребовать выполнения нескольких промежуточных шагов. Логирование этой последовательности позволяет отследить весь путь от момента отправки запроса пользователем до получения им финального ответа, включая действия системы, извлеченные документы и итоговый промпт, переданный модели. У нас также есть возможность записывать и хранить информацию о длительности задержки, количестве токенов и стоимости выполнения каждого этапа для получения более детализированного представления о процессе обработки запроса.

Как показано на рис. 11.4, конечная цель заключается в отслеживании каждого шага — от отправки пользовательского запроса до формирования итогового ответа. Благодаря этому в случае возникновения ошибки или неожиданного поведения вы сможете точно указать, на каком этапе произошел сбой. Его причиной может быть некорректный ответ, недействительный контекст или ошибка при обработке данных. Кроме того, неожиданное поведение приложения привносит риск внезапного изменения числа сгенерированных токенов на определенных этапах.

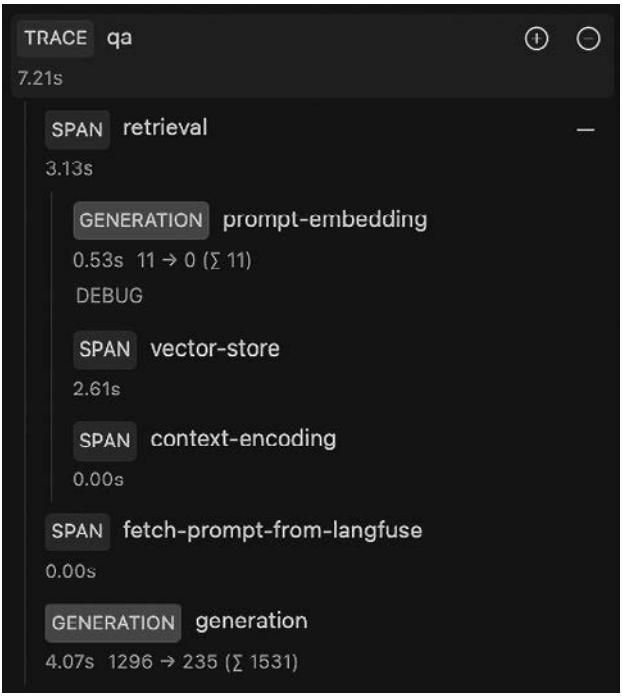


Рис. 11.4. Последовательность этапов обработки промпта в интерфейсе Langfuse

В заключение следует сказать, что LLMOps — это стремительно развивающаяся область. Из-за столь быстрой эволюции делать точные прогнозы довольно сложно. По правде говоря, мы даже не уверены, что в будущем сам термин LLMOps останется в обиходе. Тем не менее совершенно очевидно, что со временем появятся новые сценарии использования LLM, а вместе с ними — инструменты и практики для управления жизненным циклом этих моделей.

Хотя данный раздел, посвященный DevOps, MLOps и LLMOps, не является исчерпывающим, он дает некоторое представление о том, как можно применить соответствующие лучшие практики в рамках реализации нашего проекта LLM-двойника.

Развертывание пайплайнов LLM-двойника в облаке

В этом разделе мы обсудим процесс развертывания всех пайплайнов LLM-двойника в облачной среде. Чтобы наша система полноценно работала в облаке, необходимо развернуть всю инфраструктуру. Для этого мы должны выполнить следующие шаги.

1. Настроить экземпляр MongoDB на работу в облачном режиме.
2. Настроить экземпляр Qdrant на работу в облачном режиме.
3. Развернуть ZenML-пайплайны, а также контейнер и реестр артефактов на платформе AWS.
4. Упаковать код в контейнер и загрузить Docker-образ в реестр контейнеров.

Обратите внимание, что пайплайны обучения и инференса уже работают с сервисом AWS SageMaker. Таким образом, выполнив перечисленные выше четыре шага, мы обеспечим работу всей системы в облаке, подготовив ее к масштабированию и обслуживанию наших условных клиентов.



Стоимость развертывания

Мы воспользуемся бесплатными версиями сервисов MongoDB, Qdrant и ZenML, а также бесплатным тарифом AWS для запуска ZenML-пайплайнов. Применение компонентов AWS SageMaker, отвечающих за обучение и инференс, обходится дороже (в этом разделе мы не будем их запускать). Таким образом, все действия, описанные в следующих разделах, потребуют минимальных затрат (в пределах нескольких долларов США).

Обзор инфраструктуры

Прежде чем приступать к настройке компонентов, давайте кратко обсудим инфраструктуру и взаимодействие ее элементов. Это поможет вам более осознанно следовать дальнейшим инструкциям.

Как показано на рис. 11.5, нам предстоит настроить несколько сервисов. Чтобы упростить процесс, мы воспользуемся бесплатными версиями MongoDB и Qdrant, работающими в бессерверном режиме. Что касается ZenML, то для оркестрации пайплайнов в облаке мы задействуем пробную версию ZenML Cloud.

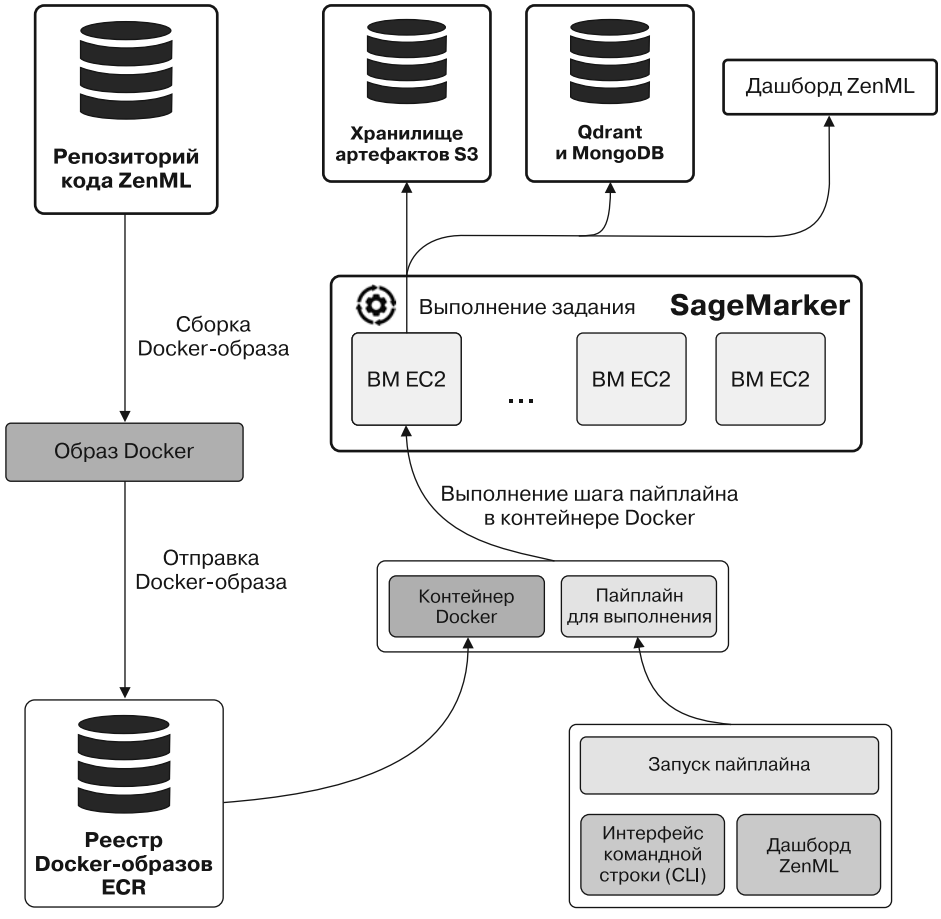


Рис. 11.5. Схема запуска пайплайна в облаке

С помощью ZenML Cloud можно быстро выделить все необходимые ресурсы AWS для запуска, масштабирования и хранения ML-пайплайнов. В частности, этот сервис позволяет в несколько кликов развернуть следующие компоненты AWS:

- сервис ECR (Elastic Container Registry) для хранения Docker-образов;
- объектное хранилище S3 для хранения артефактов и моделей;
- оркестратор SageMaker для оркестрации, запуска и масштабирования всех ML-пайплайнов.

Теперь, когда мы разобрались с ключевыми компонентами нашей инфраструктуры, рассмотрим схему запуска пайплайна в облаке, представленную на рис. 11.5. Согласно этой схеме мы должны сделать следующее.

1. Собрать Docker-образ, содержащий все системные зависимости, зависимости проекта и приложение LLM-двойника.
2. Загрузить Docker-образ в реестр *ECR* (Elastic Container Registry), сделав его доступным для сервиса *SageMaker*.
3. Теперь мы можем запускать любой из реализованных в книге пайплайнов либо из командной строки локальной машины, либо через интерфейс *ZenML*.
4. Каждый шаг ZenML-пайплайна сопоставляется с заданием SageMaker, выполняемым на *виртуальной машине* AWS EC2. В зависимости от структуры *ориентированного ациклического графа* (DAG), отражающего взаимосвязи между шагами, задачи могут выполняться параллельно или последовательно.
5. При выполнении шага пайплайна сервис SageMaker извлекает Docker-образ из ECR (см. шаг 2) и создает на его основе контейнер, в котором и выполняется этот шаг.
6. В процессе выполнения задание может обращаться к хранилищу артефактов S3, базам данных MongoDB и Qdrant — как для извлечения, так и для записи данных. Дашборд ZenML позволяет наблюдать за ходом действий пайплайна в реальном времени.

Теперь, когда мы знаем, как устроена инфраструктура, давайте выполним настройку MongoDB, Qdrant и облачного сервиса ZenML.



Какой регион AWS выбрать

Во всех примерах этой главы мы используем регион Frankfurt (eu-central-1). Вы можете выбрать другой регион, однако важно, чтобы он был одинаковым для всех сервисов, чтобы обеспечить быструю связь между компонентами и минимизировать вероятность возникновения ошибок.

Что делать в случае изменения интерфейсов сервисов

Интерфейсы и правила именования сервисов, вроде MongoDB и Qdrant, могут со временем меняться. Мы не в состоянии обновлять книгу при каждом таком изменении, поэтому при выявлении отличий от того, что написано в данном руководстве, рекомендуем обращаться к официальной документации соответствующих сервисов. Мы приносим извинения за возможные неудобства, однако это от нас не зависит.

Настройка MongoDB

В этом разделе мы создадим бесплатный кластер MongoDB и интегрируем его в наш проект. Для этого выполните следующие шаги.

1. Перейдите на сайт https://www.mongodb.com и создайте учетную запись.
2. На левой панели выберите пункт Deployment (Развертывание) ▸ Database (База данных) и нажмите кнопку Build a Cluster (Создать кластер).
3. В открывшейся форме для создания кластера выполните следующие действия:
   1) выберите кластер M0 Free;
   2) в качестве имени кластера укажите twin;
   3) в качестве поставщика услуг выберите AWS;
   4) укажите регион Frankfurt (eu-central-1). Обязательно убедитесь в том, что для всех сервисов AWS используется один и тот же регион;
   5) остальные параметры оставьте без изменений (по умолчанию);
   6) нажмите зеленую кнопку Create Deployment (Создать развертывание) в правом нижнем углу.
4. Чтобы убедиться в корректной работе созданного кластера MongoDB, подключитесь к нему с локальной машины. Мы использовали для этого расширение MongoDB for VS Code, но вы можете применить любой другой инструмент. В процессе настройки в меню Choose a connection method (Выберите метод подключения) выберите MongoDB for VS Code и следуйте инструкциям на сайте.
5. Чтобы подключиться, вам потребуется вставить строку подключения к базе данных в расширение VS Code (или в другой инструмент). Эта строка содержит имя пользователя, пароль и URL-адрес кластера, например: `mongodb+srv://<имя_пользователя>:<пароль> @twin.vhxy1.mongodb.net`. Обязательно сохраните эту строку в удобном для дальнейшего копирования месте.
6. Если вы не знаете свой пароль или хотите его изменить, выберите пункт Security (Безопасность) ▸ Quickstart (Быстрый старт) на левой панели. В этом разделе вы сможете отредактировать свои учетные данные. Сохраните их в надежном месте, поскольку позднее их уже нельзя будет просмотреть.
7. Убедившись в корректной работе подключения, перейдите в раздел Security ▸ Network Access (Безопасность ▸ Доступ к сети) и щелкните на кнопке ADD IP ADDRESS (Добавить IP-адрес). Затем выберите вариант ALLOW ACCESS FROM ANYWHERE (Разрешить доступ из любой точки мира) и нажмите кнопку Confirm (Подтвердить). Для упрощения настройки мы разрешаем доступ к кластеру MongoDB с любого IP-адреса. Это позволяет нашим пайплайнам выполнять запросы к БД и записывать в нее данные без дополнительной

настройки сетевой инфраструктуры. Такой вариант не считается безопасным для продакшена, но в контексте нашего учебного примера вполне допустим.

8. Наконец, откройте файл `.env` в вашем проекте и добавьте или замените переменную `DATABASE_HOST`, указав в качестве ее значения строку подключения к кластеру MongoDB. Она должна выглядеть примерно так: `DATABASE_HOST=mongodb+srv://<имя_пользователя>:<пароль> @twin.vhxy1.mongodb.net`.

Вот и все! Теперь чтение и запись данных будут происходить не в локальную MongoDB, а в облачный кластер, который мы только что создали. Теперь давайте выполним аналогичную процедуру для Qdrant.

Настройка Qdrant

Чтобы создать кластер Qdrant и подключить его к нашему проекту, выполните следующие шаги.

1. Перейдите на сайт https://cloud.qdrant.io/ и создайте учетную запись.
2. На левой панели выберите пункт Clusters (Кластеры) и нажмите кнопку Create (Создать).
3. В открывшейся форме для создания кластера выполните следующие действия:
   1) выберите бесплатную версию кластера (Free);
   2) в качестве облачного провайдера выберите GCP (на момент написания книги это был единственный доступный вариант для бесплатного кластера);
   3) укажите регион Frankfurt (или тот, который вы выбрали для MongoDB);
   4) в качестве имени кластера укажите twin;
   5) остальные параметры оставьте без изменений (по умолчанию) и нажмите кнопку Create (Создать).
4. Перейдите в раздел Data Access Control (Управление доступом к данным) на левой панели.
5. Нажмите кнопку Create (Создать) и выберите свой кластер twin, чтобы создать новый токен доступа. Скопируйте созданный токен в надежное место — позже вы уже не сможете его просмотреть.
6. Чтобы проверить подключение, выполните пример из раздела Usage Examples (Примеры использования).
7. Вернитесь в раздел Clusters (Кластеры), откройте созданный кластер twin и скопируйте адрес Endpoint (Конечная точка) — он понадобится вам для настройки Qdrant в коде.

Чтобы визуализировать коллекции и документы Qdrant, щелкните на кнопке Open Dashboard (Открыть дашборд) и введите свой API-ключ (API Key) в качестве пароля. На данном этапе дашборд пустой, но после запуска пайплайнов на нем отобразятся все коллекции, как показано на рис. 11.6.

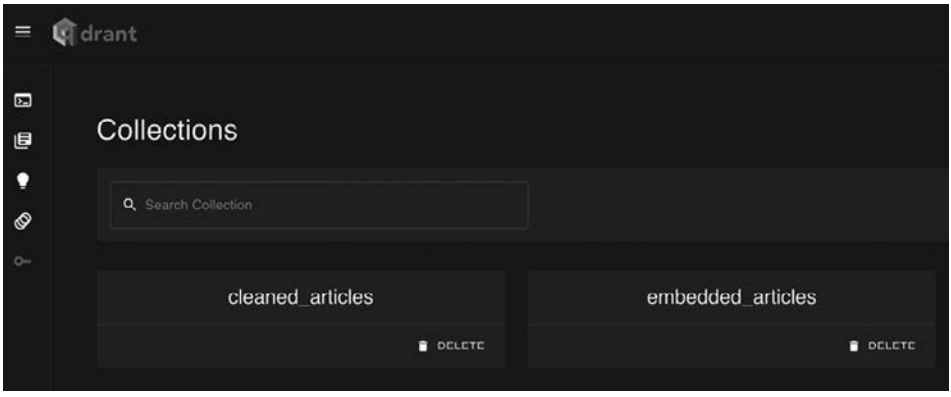


Рис. 11.6. Дашборд кластера Qdrant с двумя коллекциями

Наконец, откройте файл `.env` в вашем проекте и добавьте следующие переменные окружения:

```
USE_QDRANT_CLOUD=true
QDRANT_CLOUD_URL=<URL-адрес конечной точки, полученный на шаге 7>
QDRANT_APIKEY=<токен доступа, созданный на шаге 5>
```

Вот и все! Теперь чтение и запись данных будут происходить не из локальной векторной БД Qdrant, а из созданного облачного кластера. Чтобы убедиться в том, что все работает корректно, запустите пайплайн обработки данных, используя облачные версии MongoDB и Qdrant, с помощью команды `poe`:

```
peotry poe run-end-to-end-data-pipeline
```

Осталось настроить ZenML Cloud и развернуть всю инфраструктуру на платформе AWS.

Настройка ZenML Cloud

Настройка ZenML Cloud и инфраструктуры AWS состоит из нескольких этапов. Сначала мы создадим учетную запись в ZenML Cloud, затем развернем инфраструктуру AWS через этот сервис и, наконец, упакуем наш код в Docker-образ для дальнейшего запуска в среде AWS SageMaker.

Начнем с настройки ZenML Cloud.

1. Перейдите на сайт https://cloud.zenml.io и создайте учетную запись. Данный сервис предусматривает бесплатный семидневный пробный период, которого вполне достаточно для выполнения всех наших примеров.
2. Заполните форму регистрации, используя уникальное название организации и экземпляр (tenant) с именем twin. Это будет экземпляр ZenML, развернутый в полностью изолированной среде. Дождитесь запуска сервера экземпляра, прежде чем переходить к следующему шагу.
3. Чтобы лучше понять принцип работы ZenML Cloud, вы можете выполнить шаги, описанные в руководстве Quickstart Guide (Быстрый старт). Для запуска приложения LLM-двойника это не обязательно, но мы рекомендуем это сделать.
4. На данном этапе мы предполагаем, что вы уже ознакомились с вышеуказанным руководством. В противном случае при выполнении следующих шагов могут возникнуть проблемы. Чтобы подключить проект к созданному экземпляру ZenML Cloud, вернитесь в этот проект и введите команду `zenml connect`, указанную на вашем дашборде. Она должна выглядеть примерно так (в вашем случае URL-адрес будет другим): `zenml connect --url https://0c37a553-zenml.cloudinfra.zenml.io`.
5. Чтобы убедиться в том, что все работает корректно, запустите любой пайплайн из вашего кода. Учтите, что на текущем этапе он все еще запускается локально, однако результаты сохранятся уже не на локальном сервере, а в облачной версии ZenML:

```
poetry poe run-digital-data-etl
```

6. Перейдите в раздел Pipelines (Пайплайны), выбрав соответствующий пункт на левой панели интерфейса ZenML. Если все прошло успешно, вы увидите в списке тот самый пайплайн, который мы запустили на предыдущем шаге.



Убедитесь, что серверная версия ZenML совпадает с локальной. Например, во время написания книги обе версии имели номер 0.64.0. Если версии не совпадают, вы можете столкнуться с некорректной работой или непредсказуемым поведением. Чтобы решить эту проблему, достаточно открыть файл pyproject.toml, найти зависимость zenml, указать версию вашего сервера, а затем выполнить команду poetry lock --no-update && poetry install для обновления локальной виртуальной среды.

Чтобы развернуть код на платформе AWS, необходимо создать ZenML-стек. Под стеком понимается набор таких компонентов, как оркестратор, хранилище объектов и реестр контейнеров, которые сервис ZenML задействует для запуска пайплайнов. Стек можно рассматривать в качестве инфраструктуры. При работе в локальной среде ZenML использует стек по умолчанию, что позволяет

быстро разрабатывать и тестировать код. Однако, определив несколько стеков, вы сможете легко переключаться между инфраструктурами, например между локальной средой и платформой AWS. В данном разделе будет показано, как это можно сделать.



Прежде чем продолжить, убедитесь в том, что у вас есть учетная запись AWS с правами администратора.

Чтобы создать AWS-стек для нашего проекта, выполните следующие шаги.

1. На левой панели интерфейса ZenML откройте раздел Stacks (Стеки) и нажмите кнопку New Stack (Новый стек).
2. Вам будет предложено несколько вариантов создания стека. Проще всего создать стек с нуля прямо через интерфейс браузера. Такой способ не предусматривает особой гибкости, но вполне подходит для нашего проекта. Поэтому выберите вариант Create New Infrastructure ▸ In-browser Experience (Создать новую инфраструктуру ▸ В браузере).
3. В качестве облачного провайдера укажите AWS.
4. Выберите регион Europe (Frankfurt) — eu-central-1 либо тот, который вы задали при настройке MongoDB и Qdrant.
5. Назовите стек aws-stack. Важно указать именно это имя, чтобы используемые далее команды сработали.
6. Система ZenML создаст набор IAM-ролей, предоставляющих разрешения для взаимодействия всех компонентов между собой, а также хранилище артефактов S3, реестр контейнеров ECR и оркестратор SageMaker.
7. Нажмите кнопку Next (Далее).
8. Затем нажмите кнопку Deploy to AWS (Развернуть на AWS). В результате откроется страница CloudFormation в интерфейсе AWS. Сервис ZenML использует этот IaC-инструмент для создания всех перечисленных выше ресурсов.
9. В нижней части страницы установите все флажки, чтобы разрешить CloudFormation создавать ресурсы от вашего имени, и нажмите кнопку Create stack (Создать стек). Подождите несколько минут, пока AWS CloudFormation выполняет развертывание.
10. Вернитесь в интерфейс ZenML и нажмите кнопку Finish (Завершить).
11. Итак, с помощью ZenML мы развернули всю AWS-инфраструктуру для наших ML-пайплайнов. Мы воспользовались простейшим способом, пожертвовав при этом гибкостью, однако если вам нужен больший контроль над данным процессом, сервис ZenML также поддерживает IaC-инструмент Terraform для детального управления ресурсами или подключения к уже существующей инфраструктуре.

Вот краткий обзор созданных компонентов AWS.

- *IAM Role* — IAM-роль, представляющая собой AWS-сущность с политиками доступа, определяющими разрешенные и запрещенные действия. Она позволяет сервисам безопасно взаимодействовать друг с другом без обмена учетными данными.
- *S3* — масштабируемое и безопасное объектное хранилище, с помощью которого можно сохранять и извлекать файлы из любой точки Всемирной паутины. Часто используется для резервного копирования данных, хранения контента и создания озер данных. По степени масштабируемости и гибкости превосходит сервис Google Drive.
- *ECR* — полностью управляемый реестр контейнеров Docker, предназначенный для хранения, управления и развертывания Docker-образов.
- *SageMaker* — полностью управляемый сервис для обучения, тестирования и развертывания ML-моделей.
- *SageMaker Orchestrator* — компонент SageMaker для автоматизации рабочих процессов машинного обучения, управления зависимостями между этапами и обеспечения воспроизводимости и масштабируемости пайплайнов обучения и развертывания моделей. Альтернативными решениями являются Prefect, Dagster, Metaflow, Airflow.
- *CloudFormation* — сервис, позволяющий описывать и настраивать ресурсы AWS с помощью шаблонов. Он автоматизирует процесс развертывания инфраструктуры, благодаря чему вы можете тратить меньше времени на управление ею и сосредоточиться на разработке приложений.

Перед тем как запускать ML-пайплайны, давайте упакуем код и зависимости в Docker-образ.

Контейнеризация кода с помощью Docker

К этому моменту мы уже настроили все компоненты для хранения данных и выполнения вычислений, включая MongoDB, Qdrant и AWS. Осталось упаковать наш код и запустить его поверх этой инфраструктуры. Наиболее популярным решением такой задачи является Docker — инструмент, позволяющий создать изолированную среду (контейнер), содержащую все необходимое для запуска приложения, то есть системные зависимости, Python-библиотеки и сам код.

Файл с описанием Docker-образа находится в корневом каталоге проекта и имеет стандартное имя `Dockerfile`. Если вы хотите создать Docker-образ самостоятельно, убедитесь, что инструмент Docker установлен на вашей машине. Если это не так, установите его, следуя инструкциям, доступным по адресу https://docs.docker.com/engine/install. А теперь давайте исследуем содержимое `Dockerfile`.

Данный файл начинается с указания базового образа, которым является облегченная версия Python 3.11 на основе дистрибутива Debian Bullseye. Затем задаются переменные окружения, определяющие различные аспекты контейнера, такие как рабочий каталог, отключение генерации байт-кода Python и отправка вывода напрямую в терминал. Помимо прочего, указывается версия Poetry, которую следует установить, и задаются дополнительные переменные окружения, активизирующие неинтерактивный режим установки пакетов, что критически важно для автоматизированной сборки.

```
FROM python:3.11-slim-bullseye AS release

ENV WORKSPACE_ROOT=/app/
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1
ENV POETRY_VERSION=1.8.3
ENV DEBIAN_FRONTEND=noninteractive
ENV POETRY_NO_INTERACTION=1
```

Далее в контейнер устанавливается Google Chrome. Этот процесс предусматривает обновление списка пакетов и установку базовых инструментов (`gnupg`, `wget`, `curl`). В то же время добавляется ключ подписи Google Linux и настраивается репозиторий Google Chrome. После повторного обновления списка пакетов устанавливается стабильная версия Chrome. После установки список пакетов удаляется для сокращения размера образа.

```
RUN apt-get update -y && \
    apt-get install -y gnupg wget curl --no-install-recommends && \
    wget -q -O - https://dl-ssl.google.com/linux/linux_signing_key.pub |
gpg --dearmor -o /usr/share/keyrings/google-linux-signing-key.gpg && \
    echo "deb [signed-by=/usr/share/keyrings/google-linux-signing-key.gpg]
https://dl.google.com/linux/chrome/deb/ stable main" > /etc/apt/sources.
➥list.d/google-chrome.list && \
    apt-get update -y && \
    apt-get install -y google-chrome-stable && \
    rm -rf /var/lib/apt/lists/*
```

На следующем этапе устанавливаются прочие системные зависимости. Затем кэш пакетов очищается с целью дальнейшего уменьшения образа.

```
RUN apt-get update -y \
    && apt-get install -y --no-install-recommends build-essential \
    gcc \
    python3-dev \
    build-essential \
    libglib2.0-dev \
    libnss3-dev \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*
```

Poetry устанавливается через менеджер `pip`. Параметр `--no-cache-dir` отключает кэширование пакетов, делая образ еще более компактным. После установки Poetry настраивается на использование до 20 параллельных потоков для установки зависимостей, что ускоряет этот процесс.

```
RUN pip install --no-cache-dir "poetry==$POETRY_VERSION"
RUN poetry config installer.max-workers 20
```

Рабочий каталог внутри контейнера задается с помощью переменной `WORKSPACE_ROOT`, значение которой по умолчанию равно `/app/`. Именно в него будет помещен исходный код приложения. Туда же копируются файлы `pyproject.toml` и `poetry.lock`, определяющие зависимости проекта.

```
WORKDIR $WORKSPACE_ROOT

COPY pyproject.toml poetry.lock $WORKSPACE_ROOT
```

Затем происходит установка самих зависимостей с помощью Poetry. При этом создание виртуального окружения отключается и зависимости устанавливаются прямо в среду Python внутри контейнера. Для минимизации объема занимаемого пространства установка выполняется без зависимостей, необходимых только в процессе разработки, и без использования кэша.

Дополнительно устанавливается плагин `poethepoet`, упрощающий управление задачами внутри проекта. После этого оставшийся кэш Poetry удаляется для максимального облегчения контейнера.

```
RUN poetry config virtualenvs.create false && \
    poetry install --no-root --no-interaction --no-cache --without dev && \
    poetry self add 'poethepoet[poetry_plugin]' && \
    rm -rf ~/.cache/pypoetry/cache/ && \
    rm -rf ~/.cache/pypoetry/artifacts/
```

На финальном шаге весь каталог проекта с хост-машины копируется в рабочий каталог контейнера, чтобы обеспечить доступность всех файлов приложения внутри него.

При написании `Dockerfile` полезно отделять установку зависимостей от копирования остального кода, поскольку каждая команда Docker кэшируется и образует отдельный слой. При пересборке Docker-образа изменение одного слоя приводит к повторному выполнению всех нижележащих. Поскольку зависимости меняются гораздо реже, чем сам код, копирование файлов проекта на последнем этапе позволяет максимально эффективно использовать кэш и ускорить пересборку образа.

```
COPY . $WORKSPACE_ROOT
```

Таким образом, данный `Dockerfile` создает чистую и согласованную среду Python со всеми необходимыми зависимостями, что дает возможность запускать проект в любой среде, поддерживающей Docker.

Все, что нам осталось, — это собрать Docker-образ и загрузить его в созданный с помощью ZenML репозиторий ECR. Для сборки образа из корня проекта выполните следующую команду:

```
docker buildx build --platform linux/amd64 -t llmtwin -f Dockerfile .
```

Создание образа должно выполняться на платформе Linux, поскольку установщик Google Chrome, используемый внутри Docker, функционирует только в среде Linux. Даже если вы работаете в macOS или Windows, Docker может эмулировать виртуальный Linux-контейнер.

Тегом создаваемого образа является `llmtwin`. Команда сборки также доступна в виде команды `poethepoet`:

```
poetry poe build-docker-image
```

Теперь загрузим Docker-образ в ECR. Для этого перейдите в консоль AWS и откройте сервис ECR. Найдите созданный репозиторий, имя которого начинается с префикса `zenml-*`, как показано на рис. 11.7.

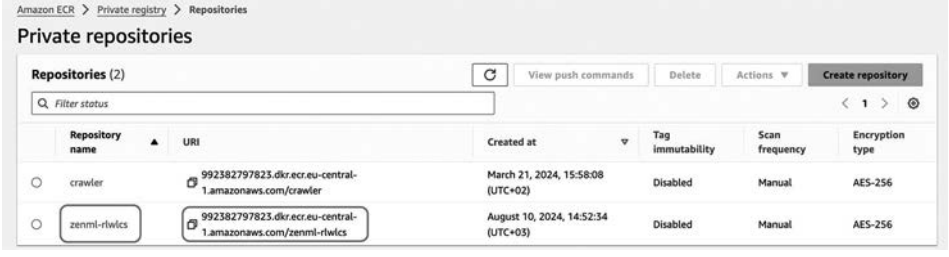


Рис. 11.7. Пример репозитория AWS ECR

Первым делом необходимо пройти аутентификацию в ECR. Для этого убедитесь, что у вас установлен интерфейс AWS CLI и настроен с помощью учетных данных администратора, как описано в главе 2:

```
AWS_REGION=<ваш_регион> # например, AWS_REGION=eu-central-1
AWS_ECR_URL=<ваш_аккаунт>

aws ecr get-login-password --region ${AWS_REGION}| docker login --username
AWS --password-stdin ${AWS_ECR_URL}
```

Чтобы узнать текущий регион AWS (`AWS_REGION`), щелкните на переключателе в правом верхнем углу интерфейса, как показано на рис. 11.8. URL-адрес вашего ECR-репозитория можно скопировать с главной страницы AWS ECR и использовать для задания значения переменной `AWS_ECR_URL` (см. рис. 11.7). После выполнения предыдущей команды вы должны увидеть в терминале сообщение `Login Succeeded`.

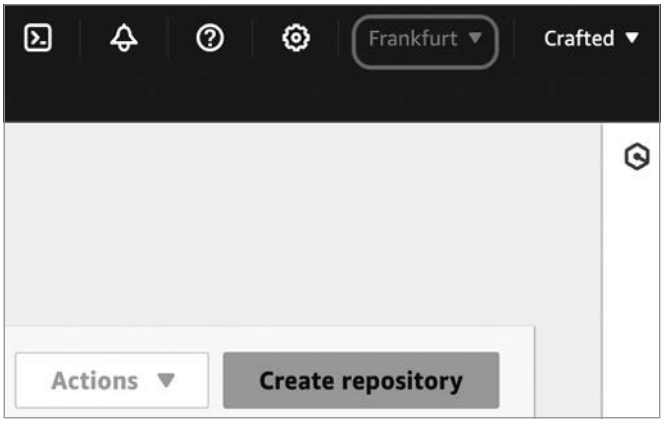


Рис. 11.8. Регион и учетные данные AWS

Теперь нужно добавить к Docker-образу `llmtwin` дополнительный тег, указывающий реестр, в который этот образ будет загружен:

```
docker tag llmtwin ${AWS_ECR_URL}:latest
```

Наконец, мы загружаем образ в ECR с помощью команды:

```
docker push ${AWS_ECR_URL}:latest
```

После завершения загрузки вернитесь к дашборду AWS ECR и откройте ваш репозиторий ZenML. Загруженный Docker-образ должен отобразиться в списке, как показано на рис. 11.9.

Для внесения в код изменений вам придется повторять всю эту последовательность шагов, что довольно утомительно и чревато ошибками. В разделе «Добавление LLMOps-компонентов в систему LLM-двойника» мы рассмотрим способы автоматизации этого процесса в рамках пайплайна непрерывной доставки (CD), реализованного с помощью платформы GitHub Actions. Мы выполнили все эти шаги вручную, чтобы вы могли разобраться в данном процессе и не воспринимали его как черный ящик. Понимание деталей особенно важно

при отладке CI/CD-пайплайнов, поскольку для устранения проблем необходимо уметь правильно интерпретировать сообщения об ошибках и знать, как их исправлять.

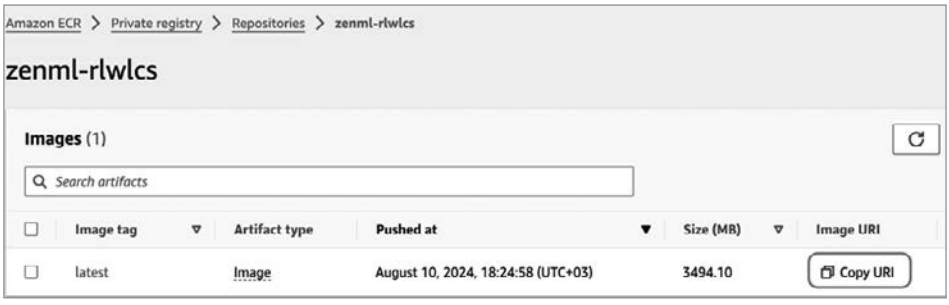


Рис. 11.9. Пример репозитория AWS ECR после загрузки Docker-образа

Теперь, когда мы собрали Docker-образ и загрузили его в репозиторий AWS ECR, давайте развернем его на платформе AWS.

Запуск пайплайнов на платформе AWS

Мы уже почти готовы к запуску ML-пайплайнов на AWS, осталось выполнить несколько финальных шагов. Для начала нам нужно сменить стек ZenML, используемый по умолчанию, на стек AWS, созданный в данной главе. Для этого в корневом каталоге проекта выполните следующую консольную команду:

```
zenml stack set aws-stack
```

Затем откройте репозиторий ZenML в AWS ECR и скопируйте URI образа, как показано на рис. 11.9. Далее перейдите в каталог `configs`, откройте файл `configs/end_to_end_data.yaml` и обновите значение атрибута `settings.docker.parent_image`, указав ваш ECR URL, как показано ниже:

```
settings:
    docker:
      parent_image: <ВАШ_ECR_URL> #например, 992382797823.dkr.ecr.eu-central-1.
      ➥ amazonaws.com/zenml-rlwlcs:latest

        skip_build: True
```

Наш пайплайн настроен на использование самой актуальной версии Docker-образа, размещенной в ECR. Это означает, что при каждом обновлении кода и загрузке нового образа в ECR пайплайн будет автоматически учитывать сделанные изменения.

Теперь нам необходимо экспортировать все учетные данные из файла `.env` в ZenML в виде секретов. Эта функция позволяет безопасно хранить ключи и предоставлять к ним доступ внутри пайплайнов:

```
poetry poe export-settings-to-zenml
```

Наконец, нам нужно обеспечить асинхронный запуск пайплайнов, чтобы избежать ожидания их завершения и возможных ошибок времени выполнения:

```
zenml orchestrator update aws-stack --synchronous=False
```

Теперь система ZenML знает, что требуется использовать стек AWS и наш пользовательский Docker-образ, а также имеет доступ ко всем необходимым учетным данным. Все готово к запуску. Запустите пайплайн `end-to-end-data-pipeline` с помощью следующей команды:

```
poetry poe run-end-to-end-data-pipeline
```

Затем перейдите в ZenML Cloud ▸ Pipelines ▸ end_to_end_data и выберите последний запуск. На дашборде ZenML отобразится текущее состояние выполнения пайплайна, как показано на рис. 11.10. Обратите внимание, что этот пайплайн выполняет все этапы обработки данных в рамках одного запуска.

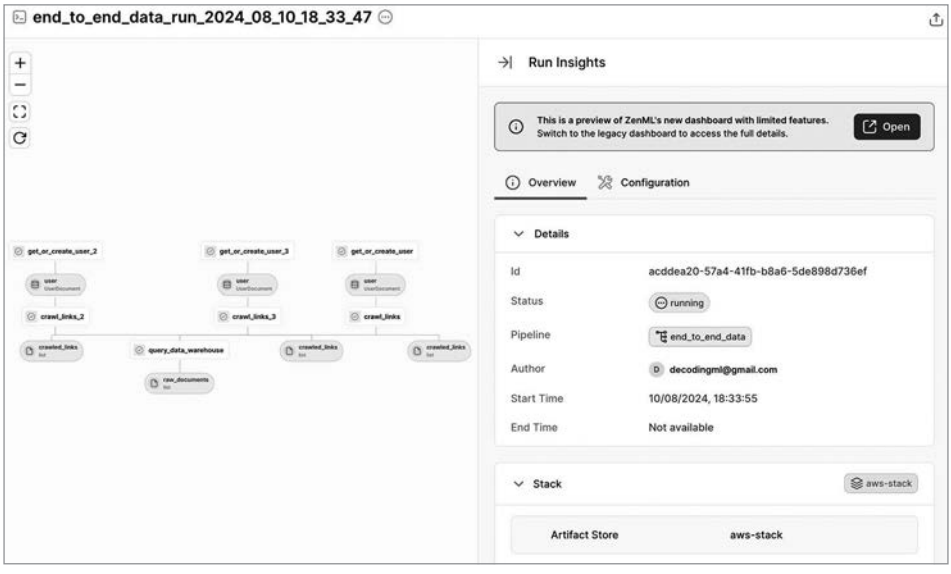


Рис. 11.10. Пример запуска пайплайна end-to-end-data-pipeline в ZenML

Причины для объединения всех шагов в один пайплайн мы обсудим в разделе «Добавление LLMOps-компонентов в систему LLM-двойника».

Вы можете щелкнуть на любом блоке и просмотреть подробную информацию о запуске, коде, использованном на соответствующем шаге, а также взглянуть на журналы, предназначенные для мониторинга и отладки (рис. 11.11).

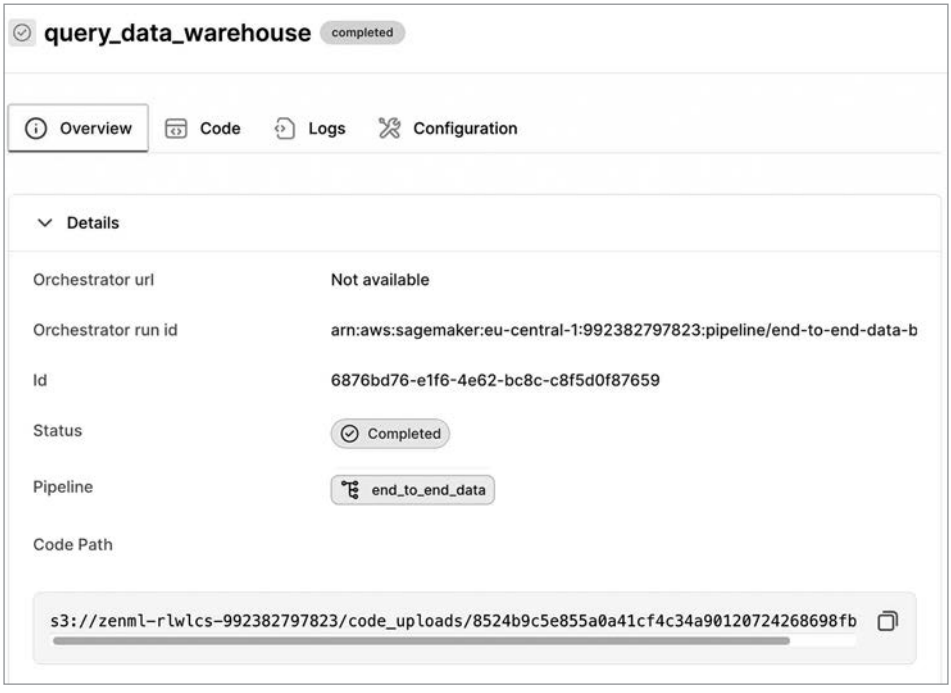


Рис. 11.11. Пример метаданных шага в ZenML



Чтобы запустить другие пайплайны, необходимо обновить атрибут settings.docker.parent_image в соответствующем конфигурационном файле в каталоге configs/.

Для получения более подробных сведений о запусках пайплайнов воспользуйтесь интерфейсом AWS SageMaker. На левой панели выберите SageMaker dashboard (Дашборд SageMaker), а затем справа в столбце Processing (Обработка) выберите пункт Running (Выполняется), как показано на рис. 11.12.

В результате отобразится список всех процессов, выполняемых вашими пайплайнами ZenML.

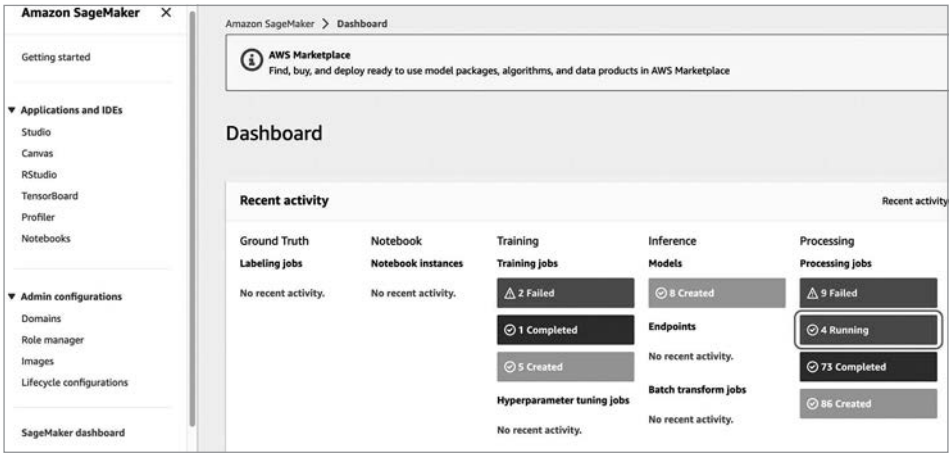


Рис. 11.12. Дашборд SageMaker



Если вы хотите снова запускать пайплайны локально, выполните следующую консольную команду:

```
poetry poe set-local-stack
```

Если вы хотите переключиться с облачного дашборда ZenML на локальную версию, выполните такую команду:

```
zenml disconnect
```

Устранение ошибки ResourceLimitExceeded, возникающей при запуске ZenML-пайплайна на платформе SageMaker

Если после запуска ZenML-пайплайна на платформе SageMaker со стеком AWS вы столкнулись с ошибкой *ResourceLimitExceeded*, вам необходимо явно запросить у AWS доступ к определенному типу виртуальных машин EC2.

По умолчанию система ZenML использует машины типа `ml.t3.medium`, предоставляемые AWS бесплатно. Однако некоторые аккаунты AWS не имеют доступа к этим виртуальным машинам по умолчанию. Для проверки откройте консоль AWS и найдите раздел Service Quotas (Сервисные квоты).

Затем на левой панели выберите AWS services (Сервисы AWS), найдите Amazon SageMaker, а затем — `ml.t3.medium`. На рис. 11.13 показаны квоты для данного типа машин в нашем аккаунте. Если в вашем случае это значение равно `0`, отправьте запрос на увеличение квоты до значений, аналогичных указанным на рис. 11.13

в столбце Applied account-level quota value (Значение квоты на уровне аккаунта). Это абсолютно бесплатно и требует всего нескольких нажатий кнопки мыши. Однако одобрение вашего запроса может занять от нескольких часов до целых суток.

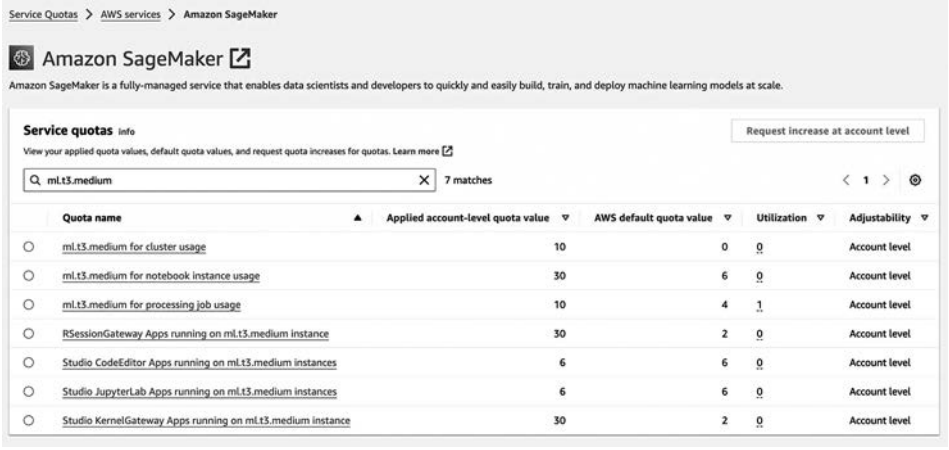


Рис. 11.13. Квоты SageMaker для машин типа ml.t3.medium (ожидаемые значения)

Подробные инструкции по устранению такой ошибки и отправке запроса на увеличение квоты доступны по ссылке https://repost.aws/knowledge-center/sagemaker-resource-limit-exceeded-error.



Если вы изменили значения переменных в файле .env и хотите обновить секреты ZenML, сначала удалите старые секреты, выполнив следующую консольную команду:

```
poetry poe delete-settings-zenml
```

Затем снова экспортируйте переменные с помощью команды:

```
poetry poe export-settings-to-zenml
```

Добавление LLMOps-компонентов в систему LLM-двойника

В предыдущем разделе мы вручную создали Docker-образ для LLM-двойника и загрузили его в репозиторий ECR. Теперь давайте автоматизируем весь этот процесс, реализовав пайплайн непрерывной интеграции и доставки (CI/CD) с помощью GitHub Actions и пайплайн непрерывного обучения (CT)

с использованием ZenML. Как уже упоминалось ранее, реализация CI/CD/CT-пайплайна гарантирует, что каждая новая функция, отправляемая в основную ветку, будет протестирована и согласована с остальным кодом. Кроме того, автоматизация процессов развертывания и обучения облегчает совместную работу, экономит время и снижает вероятность возникновения ошибок, связанных с человеческим фактором.

В заключительной части этого раздела мы покажем, как можно реализовать пайплайн мониторинга промптов с использованием инструмента Opik от Comet ML, а также систему оповещений на базе ZenML. Такой пайплайн поможет нам отлаживать и анализировать RAG- и LLM-логику. Поскольку поведение LLM-систем является недетерминированным, для мониторинга ML-логики крайне важно сохранять и отслеживать всю последовательность этапов обработки промптов.

Прежде чем перейти к реализации, кратко обсудим CI/CD-пайплайн LLM-двойника.

CI/CD-пайплайн LLM-двойника

В нашей архитектуре предусмотрены две среды: промежуточная (staging) и рабочая (production). При разработке новой функции мы создаем отдельную ветку на основе промежуточной и всю работу ведем исключительно в ней. Когда функция готова, мы делаем *запрос на внесение изменений* (pull request, PR) в промежуточную ветку. После одобрения ветка с новой функцией сливается с промежуточной — это стандартный процесс, используемый при реализации многих проектов. Возможны вариации, например, с добавлением отдельной среды разработки, но принцип остается тем же.

Как показано на рис. 11.14, CI-пайплайн запускается при создании запроса на внесение изменений. На этом этапе мы проверяем ветку с новой функцией на наличие ошибок линтинга и форматирования. Затем выполняется команда `gitleaks`, позволяющая выявить учетные данные и другую чувствительную информацию, которая могла быть случайно включена в репозиторий. В случае успешного прохождения всех этих этапов статического анализа запускаются автоматические тесты. Поскольку статический анализ выполняется быстрее, чем эти тесты, проверку кода рекомендуется начинать именно с него. Мы предлагаем следующий порядок шагов CI-пайплайна:

- проверка с помощью команды `gitleaks`;
- линтинг;
- проверка форматирования;
- автоматические тесты (например, модульные и интеграционные).

Если хотя бы один из этапов завершится неудачно, CI-пайплайн будет остановлен и разработчик, создавший запрос на внесение изменений, не сможет объединить свою ветку с промежуточной до тех пор, пока не устранит проблему.

Таким образом, реализация CI-пайплайна позволяет убедиться, что все новые функции соответствуют стандартам репозитория и не нарушают существующую функциональность. Тот же процесс повторяется и при объединении промежуточной ветки с рабочей. Мы создаем запрос на внесение изменений, и CI-пайплайн автоматически запускается перед слиянием.

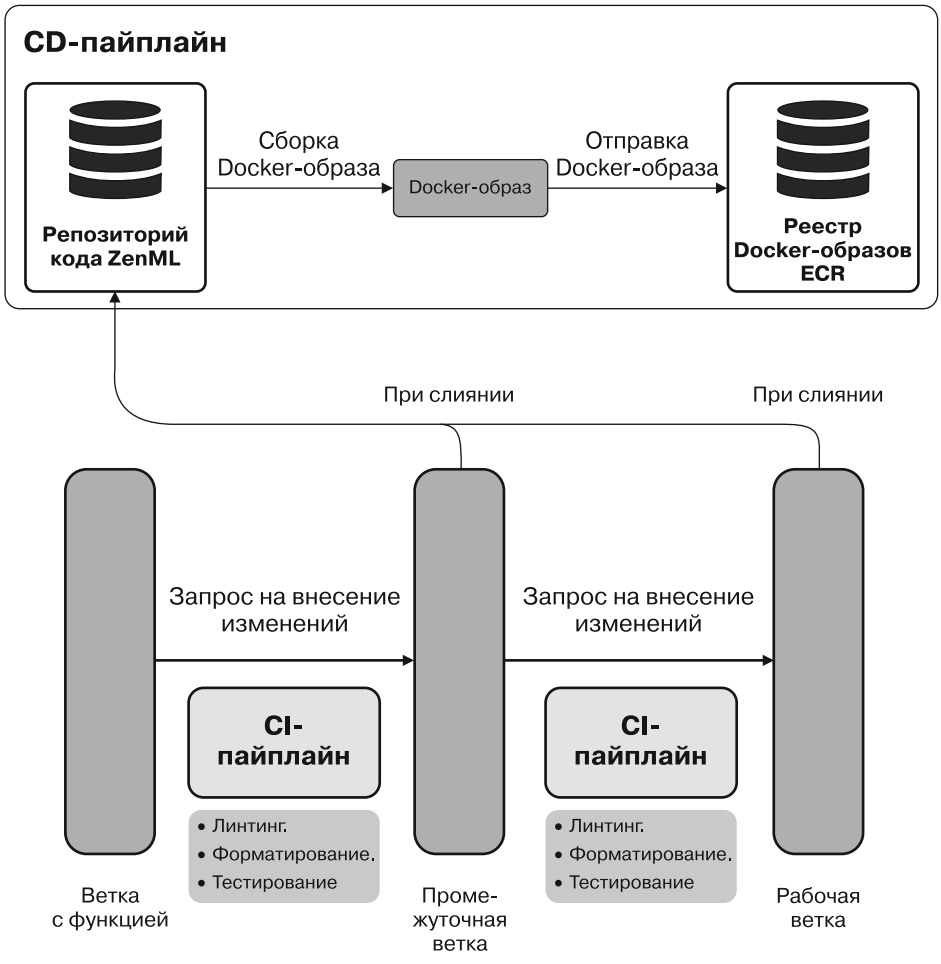


Рис. 11.14. Рабочий процесс CI/CD-пайплайнов

Пайплайн непрерывной доставки (CD) запускается после слияния веток. Например, после объединения функциональной и промежуточной веток CD-пайплайн берет код из промежуточной ветки, собирает новый Docker-образ и загружает его в репозиторий AWS ECR. При последующих запусках пайплайна в промежуточной среде будет использоваться самый актуальный Docker-образ, созданный CD-пайплайном. Аналогичный процесс происходит и при объединении промежуточной и рабочей веток. Ключевое различие состоит в том, что промежуточная среда служит экспериментальной площадкой, где новую функцию можно проверить не только автоматически в рамках CI-пайплайна, но и протестировать вручную.



В нашем репозитории используется только одна основная (main) ветка, отражающая продакшен, и отдельные ветки для разработки новых функций. Мы выбрали такую схему ради упрощения, однако вышеописанные принципы к ней тоже применимы. Чтобы расширить текущую схему, достаточно создать промежуточную ветку и добавить ее в CD-пайплайн.

Подробнее об ошибках форматирования

Ошибки форматирования связаны со стилем и структурой кода и выражаются в нарушении единых правил его визуального оформления. К таким ошибкам могут относиться неправильные отступы, лишние или пропущенные пробелы, превышение допустимой длины строки и другие стилистические недочеты.

Основная цель форматирования — повысить удобство восприятия и сопровождения кода. Единообразное оформление облегчает совместную работу, поскольку код выглядит одинаково, независимо от того, кто его написал. Примерами ошибок форматирования являются:

- неправильные отступы (например, смешение пробелов и символов табуляции);
- слишком длинные строки (например, превышающие 79 или 88 символов, в зависимости от выбранного стиля);
- отсутствие или избыток пробелов вокруг операторов или после запятых.

Подробнее об ошибках линтинга

Ошибки линтинга указывают на потенциальные проблемы в коде, которые могут привести к возникновению багов, снижению эффективности или нарушению принятых стандартов программирования. Линтеры выполняют статический анализ кода и помогают выявлять такие проблемы, как неиспользуемые переменные, неопределенные имена или применение сомнительных практик.

Главная задача линтинга заключается в раннем обнаружении потенциальных ошибок и плохих практик, что позволяет повысить качество кода и снизить вероятность появления багов. Примерами ошибок линтинга являются:

- неиспользуемые импорты или переменные;
- использование неопределенных переменных или функций;
- потенциально опасный код (например, использование `==` вместо `is` при сравнении с `None`).

Для форматирования и линтинга мы применяем универсальный инструмент Ruff, который проверяет код на соответствие стилю оформления (в том числе стандарту PEP 8 для Python), а также выполняет более глубокий его анализ для выявления потенциальных ошибок и проблем с качеством. Ruff написан на языке Rust, что ускоряет процесс обработки объемных кодовых баз.

Прежде чем перейти к реализации описанных выше подходов, рассмотрим основные принципы работы платформы GitHub Actions.

Краткий обзор платформы GitHub Actions

GitHub Actions — это CI/CD-платформа, позволяющая разработчикам автоматизировать рабочие процессы непосредственно внутри репозитория GitHub. С ее помощью можно выполнять сборку, тестирование и развертывание кода прямо из GitHub, определяя соответствующие сценарии в YAML-файлах. Поскольку платформа GitHub Actions является частью GitHub, она бесшовно интегрируется с репозиториями (repositories), задачами (issues), запросами на внесение изменений (pull requests) и прочими функциями данной системы. Ниже перечислены ключевые компоненты.

- *Рабочие процессы (workflows)* — автоматизированные сценарии, определенные в YAML-файлах, которые хранятся в каталоге `.github/workflows` вашего репозитория. Они описывают, *что* должно происходить (например, сборка, тестирование или развертывание) и *когда* (например, при отправке изменений или создании запроса на их внесение).
- *Задания (jobs)* — составные части рабочих процессов, объединяющие шаги, которые выполняются одним исполнителем. Любое задание запускается в собственном виртуальном окружении.
- *Шаги (steps)* — каждое задание состоит из нескольких независимых шагов, представляющих собой либо действия, либо shell-команды.
- *Действия (actions)* — команды или сценарии многократного использования, сравнимые с функциями Python. Можно использовать как готовые действия из GitHub Marketplace, так и создавать собственные.

- *Исполнители (runners)* — серверы, на которых выполняются задания. GitHub предоставляет готовые средства (на базе Linux, Windows и macOS), но при необходимости можно применять и размещенные на своих серверах (self-hosted).

Рабочий процесс описывается с использованием синтаксиса YAML. Например, простой рабочий процесс, клонирующий текущий репозиторий и устанавливающий Python 3.11 на машину Ubuntu, выглядит так:

```
name: Example
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
        - name: Checkout
          uses: actions/checkout@v3

        - name: Setup Python
          uses: actions/setup-python@v3
          with:
              python-version: "3.11"
```

Рабочие процессы запускаются при наступлении событий `push`, `pull_request` или `schedule`. Например, такой процесс может активизироваться каждый раз при отправке кода в определенную ветку. Теперь, когда мы разобрались с устройством платформы GitHub Actions, рассмотрим CI-пайплайн нашего LLM-двойника.

CI-пайплайн

CI-пайплайн LLM-двойника разделен на два задания.

- Задание `QA` проверяет код на наличие ошибок форматирования и линтинга с помощью инструмента Ruff. Кроме того, оно выполняет сканирование репозитория на предмет утечки секретов через команду `gitleaks`.
- Задание `Test` выполняет все автоматические тесты с использованием `Pytest`. В рамках данного проекта реализован лишь демонстрационный тест, иллюстрирующий структуру CI-пайплайна, однако вы легко можете расширить ее, добавив тесты, подходящие для вашей конкретной ситуации.

YAML-файл CI-пайплайна. Файл CI-пайплайна расположен по адресу `.github/workflows/ci.yaml`. В нем задается имя рабочего процесса — `CI`, под которым он будет отображаться в интерфейсе GitHub Actions. Данный рабочий процесс настроен на автоматический запуск при наступлении события `pull_request`, то есть при открытии, повторном открытии или синхронизации запроса на внесение изменений.

```
name: CI

on:
  pull_request:
```

Раздел `concurrency` гарантирует, что для каждой Git-ссылки (reference) (например, ветки) одновременно будет запущен только один экземпляр рабочего процесса. Поле `group` задается с помощью синтаксиса выражений GitHub и используется для создания уникального имени группы на основе названия рабочего процесса и ссылки. Строка `cancel-in-progress: true` позволяет избежать лишних запусков одного и того же процесса, гарантируя, что при запуске нового рабочего процесса старый будет отменен.

```
concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true
```

В структуре рабочего процесса определены два задания — `QA` и `Test`. Оба выполняются в последней версии Ubuntu, на что указывает параметр `runs-on: ubuntu-latest`.

Первое задание `QA` отвечает за контроль качества, в том числе за проверку кода и его форматирования. В рамках этого задания сначала выполняется действие `actions/checkout@v3`, предоставляющее заданию доступ к коду, подлежащему анализу.

```
jobs:
  qa:
    name: QA
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v3
```

Далее настраивается среда Python с помощью действия `actions/setup-python@v3` с указанием версии Python `"3.11"`. Это гарантирует, что все последующие этапы задания будут выполняться в корректной среде.

```
      - name: Setup Python
        uses: actions/setup-python@v3
        with:
          python-version: "3.11"
```

Затем рабочий процесс устанавливает инструмент Poetry версии `1.8.3` через действие `abatilo/actions-poetry@v2`:

```
      - name: Install poetry
        uses: abatilo/actions-poetry@v2
        with:
          poetry-version: 1.8.3
```

После настройки Poetry устанавливаются необходимые для разработки зависимости с помощью команды `poetry install --only dev`. Кроме того, добавляется

плагин `poethepoet` для Poetry, который упрощает запуск заранее определенных задач внутри проекта:

```
      - name: Install packages
        run: |
          poetry install --only dev
          poetry self add 'poethepoet[poetry_plugin]'
```

Затем задание `QA` выполняет несколько проверок качества кода. Первая из них применяет инструмент `gitleaks` для поиска секретов в кодовой базе, чтобы убедиться в том, что никакая чувствительная информация не была случайно включена в репозиторий.

```
      - name: gitleaks check
        run: poetry poe gitleaks-check
```

После проверки с помощью `gitleaks` рабочий процесс запускает линтинг, чтобы обеспечить соответствие кода стандартам оформления и лучшим практикам Python-разработки. Для этого используется команда `poetry poe lint-check`, которая задействует инструмент Ruff:

```
      - name: Lint check [Python]
        run: poetry poe lint-check
```

Заключительным шагом задания `QA` является проверка форматирования, гарантирующая соответствие Python-кода стилистическим правилам проекта. Данный шаг выполняется с помощью команды `poetry poe format-check`, также применяющей инструмент Ruff:

```
      - name: Format check [Python]
        run: poetry poe format-check
```

Второе задание в рабочем процессе называется `Test` и также выполняется в последней версии среды Ubuntu. Подобно заданию `qa`, оно начинается с получения исходного кода из репозитория и установки Python версии 3.11 и Poetry версии 1.8.3.

```
  test:
    name: Test
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v3
      ...
```

После настройки системных зависимостей задание `Test` устанавливает все зависимости проекта через команду `poetry install`. Поскольку целью этого за-

дания является запуск тестов, на текущем этапе нам требуется установить все библиотеки, необходимые для полноценного функционирования приложения:

```
- name: Install packages
  run: |
    poetry install --without aws
    poetry self add 'poethepoet[poetry_plugin]'
```

Наконец, задание `Test` запускает тесты с помощью команды `poetry poe test`. Этот шаг позволяет убедиться в том, что внесенные изменения не нарушили работоспособность существующего функционала:

```
- name: Run tests
  run: |
    echo "Running tests..."
    poetry poe test
```

Если какой-то шаг задания `qa` или `test` завершится с ошибкой, выполнение рабочего процесса GitHub Actions прервется и соответствующий запрос на внесение изменений не будет выполнен вплоть до устранения проблемы. Такой подход гарантирует, что все новые функции, добавляемые в основные ветки, соответствуют стандартам проекта и не нарушают существующую функциональность.

На рис. 11.15 показан пример запуска CI-пайплайна на вкладке Actions (Действия) репозитория GitHub. Он был запущен после создания коммита с сообщением feat: Add Docker image and CD pipeline и включает два описанных выше задания: `QA` и `Test`.

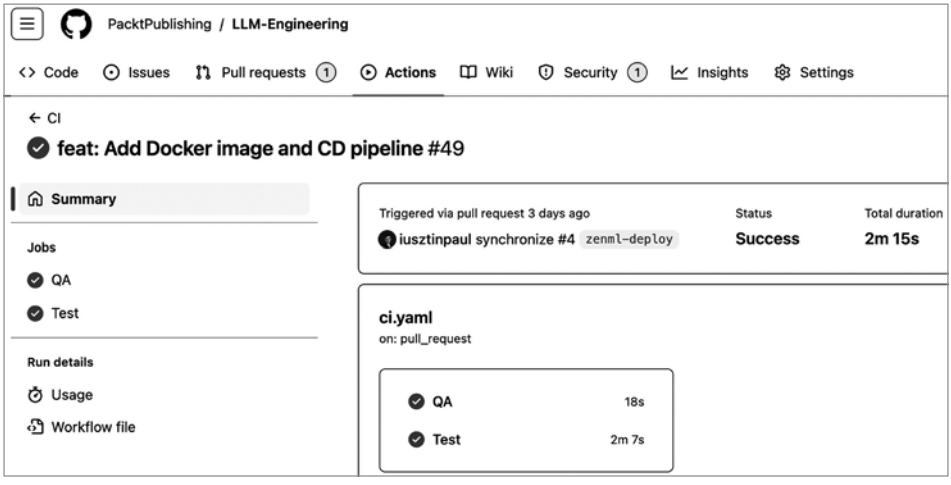


Рис. 11.15. Пример запуска CI-пайплайна в интерфейсе GitHub Actions

CD-пайплайн

CD-пайплайн автоматизирует процесс работы с Docker, описанный в разделе «Развертывание пайплайнов LLM-двойника в облаке» и включающий следующие этапы:

- настройка Docker;
- аутентификация на платформе AWS;
- сборка Docker-образа;
- отправка Docker-образа в AWS ECR.

Рассмотрим YAML-файл GitHub Actions, расположенный по адресу `.github/workflows/cd.yaml`. В нем указано имя рабочего процесса — `CD`, а также условие его запуска. Данный процесс автоматически запускается при каждой отправке (`push`) нового кода в основную (`main`) ветку репозитория. Это поведение задается с помощью параметра `on.push`:

```
name: CD

on:
  push:
    branches:
      - main
```

Затем в рабочем процессе определяется единственное задание с именем `Build & Push Docker Image`:

```
jobs:
  build:
    name: Build & Push Docker Image
    runs-on: ubuntu-latest
```

Первым шагом при выполнении этого задания является клонирование кода из репозитория:

```
steps:
  - name: Checkout Code
    uses: actions/checkout@v3
```

После получения кода рабочий процесс настраивает `Docker Buildx` — плагин для интерфейса командной строки Docker, расширяющий возможности сборки. Он позволяет создавать образы для нескольких платформ, а также поддерживает импорт и экспорт кэша:

```
  - name: Set up Docker Buildx
    uses: docker/setup-buildx-action@v3
```

На следующем этапе настраиваются учетные данные AWS для взаимодействия с такими сервисами, как Amazon *Elastic Container Registry (ECR)*, в который бу-

дет загружен собранный Docker-образ. Доступ к AWS-ключу, секретному ключу и региону осуществляется через секреты репозитория, что позволяет безопасно аутентифицировать рабочий процесс и предоставить ему необходимые права для отправки образов в ECR. Мы обсудим способ настройки этих секретов, когда закончим с YAML-файлом:

```
- name: Configure AWS credentials
  uses: aws-actions/configure-aws-credentials@v1
  with:
    aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
    aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    aws-region: ${{ secrets.AWS_REGION }}
```

После настройки учетных данных AWS рабочий процесс выполняет вход в систему Amazon ECR. Такой шаг необходим для аутентификации Docker CLI в ECR и дальнейшей отправки образов в реестр:

```
- name: Login to Amazon ECR
  id: login-ecr
  uses: aws-actions/amazon-ecr-login@v1
```

Заключительный этап рабочего процесса предполагает сборку Docker-образа и его отправку в репозиторий Amazon ECR. Для этого используется действие `docker/build-push-action@v6`. Параметр `context` задает контекст сборки, которым, как правило, является корневой каталог репозитория. Параметр `file` указывает путь к файлу `Dockerfile` с описанием способа сборки образа. В разделе `tags` прописываются теги образа, включая хеш коммита и тег `latest`, который обычно применяется для обозначения самой актуальной версии образа. Параметр `push`, установленный в значение `true`, означает, что образ будет отправлен в ECR сразу после сборки:

```
- name: Build images & push to ECR
  id: build-image
  uses: docker/build-push-action@v6
  with:
    context: .
    file: ./Dockerfile
    tags: |
      ${{ steps.login-ecr.outputs.registry }}/${{ secrets.AWS_ECR_
      ➥NAME }}:${{ github.sha }}
      ${{ steps.login-ecr.outputs.registry }}/${{ secrets.AWS_ECR_NAME }}:latest
    push: true
```

Таким образом, CD-пайплайн выполняет три действия: аутентифицируется в системе AWS, собирает Docker-образ и отправляет его в реестр AWS ECR. Тег `latest` и хеш коммита позволяют нам использовать последнюю версию образа, а также указать на коммит, из которого этот образ был сгенерирован.

В текущей конфигурации репозиторий содержит только основную ветку (`main`), соответствующую продакшену. Однако вы можете расширить его структуру,

добавив промежуточную ветку и ветку для разработки. Для этого достаточно указать названия дополнительных веток в параметре `on.push.branches` в начале YAML-файла.

На рис. 11.16 показан пример запуска CD-пайплайна после добавления изменений в основную ветку. Как и раньше, в нем присутствует только одно задание — Build & Push Docker Image.

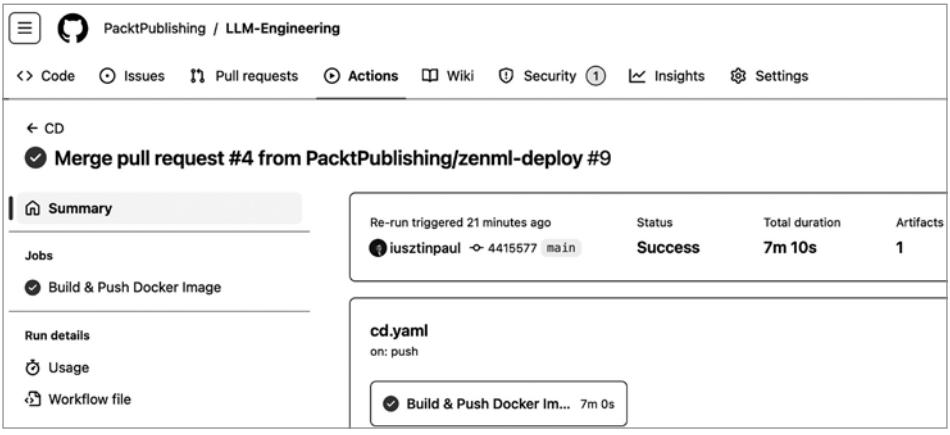


Рис. 11.16. Пример запуска CD-пайплайна в интерфейсе GitHub Actions

Теперь нам осталось протестировать CI/CD-пайплайн и посмотреть, как он работает.

Тестирование CI/CD-пайплайнов

Для самостоятельного тестирования CI/CD-пайплайнов вам необходимо создать ответвление (форк) репозитория LLM-Engineering, чтобы получить разрешение на *запись (push)* изменений. Официальная инструкция по созданию ответвления кода GitHub-проекта доступна по адресу https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/fork-a-repo.

На заключительном этапе необходимо настроить несколько секретов, которые позволят CD-пайплайну авторизоваться в системе AWS и указать на нужный ресурс в реестре ECR. Для этого перейдите на вкладку Settings (Настройки) в верхней части ответвленного репозитория на GitHub. В левом меню в разделе Security (Безопасность) раскройте пункт Secrets and variables (Секреты и переменные), а затем щелкните на пункте Actions (Действия). Затем на вкладке Secrets (Секреты) создайте четыре секретных параметра репозитория, как показано на рис. 11.18. Эти данные будут храниться в зашифрованном виде и доступны только CD-пайплайну, работающему на платформе GitHub Actions.

Параметры `AWS_ACCESS_KEY_ID` и `AWS_SECRET_ACCESS_KEY` — это учетные данные AWS, которые вы использовали на протяжении всей книги. Их создание было подробно описано в главе 2. Значения `AWS_REGION` (например, `eu-central-1`) и `AWS_ECR_NAME` совпадают с теми, что вы применяли в разделе «Развертывание пайплайнов LLM-двойника в облаке».



Для AWS_ECR_NAME следует указать только имя репозитория (например, zenml-vrsopg), а не полный URI (например, 992382797823.dkr.ecr.eu-central-1.amazonaws.com/zenml-vrsopg), как показано на рис. 11.17.

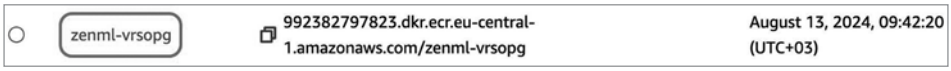


Рис. 11.17. Настройка только имени репозитория

Чтобы запустить CI-пайплайн, создайте функциональную ветку (feature), внесите изменения в код или документацию и направьте запрос на внесение изменений в основную ветку. Чтобы запустить CD-пайплайн, выполните слияние функциональной ветки с основной через этот PR.

После завершения работы CD-пайплайна в GitHub Actions откройте консоль Amazon ECR и проверьте, был ли Docker-образ успешно загружен в репозиторий.

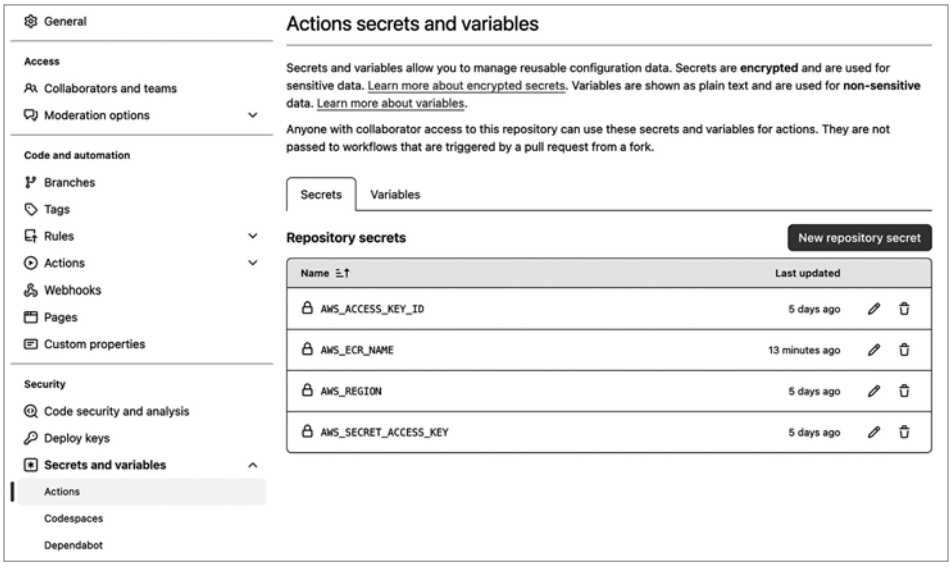


Рис. 11.18. Секреты GitHub Actions

Дополнительную информацию о настройке секретов GitHub Actions можно найти в официальной документации: https://docs.github.com/en/actions/security-for-github-actions/security-guides/using-secrets-in-github-actions.

CT-пайплайн

Для создания CT-пайплайна мы воспользуемся системой ZenML. Как только ZenML (или любой другой оркестратор, например Metaflow, Dagster или Airflow) возьмет на себя управление всеми вашими пайплайнами, а инфраструктура будет развернута, вы окажетесь всего в шаге от реализации непрерывного обучения.

Напомним, что CI/CD-пайплайн отвечает за тестирование, сборку и развертывание кода, характерные для любого программного обеспечения, а CT-пайплайн опирается на код, управляемый CI/CD-пайплайном, и автоматизирует процессы обработки данных, обучения модели и ее развертывания, присущие исключительно ИИ-системам.

Перед тем как перейти к технической реализации, отметим два архитектурных решения, которые существенно упростили реализацию непрерывного обучения.

- *Архитектура FTI*. Модульная система с четко определенными интерфейсами и компонентами позволила легко выявить взаимосвязи между пайплайнами и автоматизировать их работу.
- *Использование оркестратора*. Мы начали использовать ZenML на самом раннем этапе разработки проекта. Изначально он применялся только локально, но уже тогда служил входной точкой для запуска пайплайнов и средством мониторинга их выполнения. Такой подход вынудил нас разделить пайплайны и организовать их взаимодействие исключительно через различные типы хранилищ: данных, признаков и артефактов. Благодаря изначальной интеграции ZenML мы смогли избежать реализации отдельного интерфейса командной строки для настройки приложения и вместо этого сконфигурировали систему с помощью YAML-файлов.

На рис. 11.19 показаны все пайплайны, которые необходимо связать между собой, чтобы полностью автоматизировать процессы обучения и развертывания. Эти пайплайны объединяют все, что мы обсуждали в книге, поэтому на данном этапе мы рассматриваем их просто как «черные ящики», взаимодействующие друг с другом.

Далее мы обсудим, каким образом запускается начальный пайплайн и как реализуется каскадный запуск остальных.

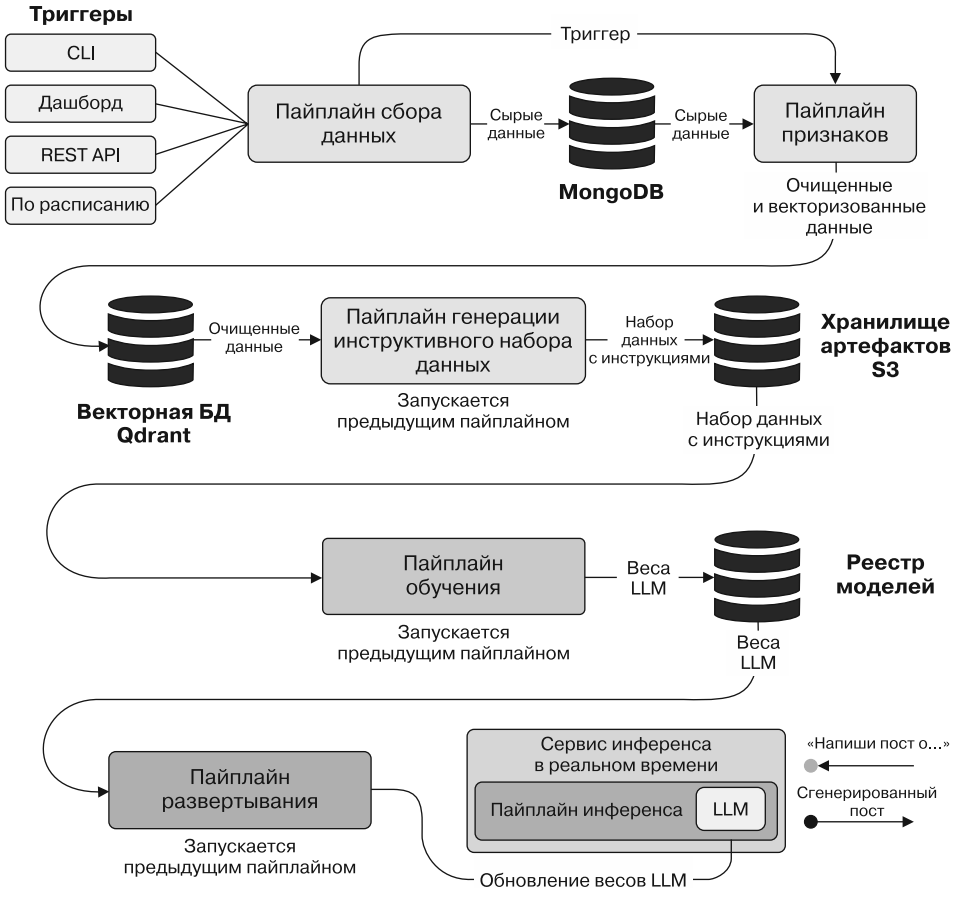


Рис. 11.19. CT-пайплайн

Запуск первого пайплайна

Как показано на рис. 11.19, первым должен запускаться пайплайн сбора данных. Это можно реализовать тремя способами.

- *Ручной запуск*. Осуществляется через интерфейс командной строки (CLI) или дашборд оркестратора, в нашем случае — через дашборд ZenML. Несмотря на свою простоту, ручной запуск является эффективным способом выполнения всех этапов работы ML-системы — от сбора данных до развертывания — с помощью одного действия. Он не требует использования десятков сценариев, которые могут быть некорректно настроены или выполнены в неправильном порядке.

- *REST API*. Запуск пайплайна по HTTP-запросу особенно удобен при интеграции ML-пайплайнов с другими компонентами. Например, можно создать объект-наблюдатель, отслеживающий появление новых статей и запускающий ML-логику через REST API при их обнаружении (за подробностями рекомендуем обратиться к официальной документации ZenML: https://docs.zenml.io/v/docs/how-to/trigger-pipelines/trigger-a-pipeline-from-rest-api).
- *Запуск по расписанию*. Еще один распространенный способ — запуск пайплайна с заданной периодичностью, например каждый день, час или минуту, в зависимости от конкретного случая использования. Большинство оркестраторов, включая ZenML, предоставляют интерфейс на основе cron-выражений для задания частоты запусков. Приведенный ниже код запускает пайплайн ежечасно:

```
Schedule(cron_expression="* * 1 * *")
```

В системе LLM-двойника мы запустили пайплайны вручную, так как в ней отсутствуют внешние компоненты, которые могли бы задействовать REST API. Кроме того, наборы данных генерируются на основе списка статических ссылок, определенных в конфигурации ZenML, поэтому запуск пайплайнов по расписанию не имеет смысла, поскольку результаты каждый раз будут одинаковыми.

Следующим этапом развития проекта может стать реализация наблюдателя, отслеживающего появление новых статей. При обнаружении новых данных он сгенерирует новую конфигурацию и запустит пайплайны через REST API. Альтернативный вариант — реализация наблюдателя в виде отдельного пайплайна и его ежедневный запуск по расписанию. При обнаружении новых данных он будет запускать все компоненты ML-системы, а в противном случае — завершать работу.

Таким образом, возможность запускать все ML-пайплайны вручную с помощью одной команды позволяет вам легко адаптировать систему к более сложным и продвинутым сценариям.

Запуск последующих пайплайнов

Чтобы упростить реализацию, мы связали все пайплайны в цепочку: после завершения работы пайплайна сбора данных автоматически запускается пайплайн обработки признаков. После успешного выполнения его шагов запускается пайплайн генерации набора данных с инструкциями и т. д. Эту логику можно усложнить, например сделав так, чтобы пайплайн генерации набора с инструкциями запускался ежедневно, проверял объем новых данных в векторной БД Qdrant и продолжал работу только при наличии достаточного их количества. Далее при необходимости система донастраивается и оптимизируется для снижения затрат.

Чтобы иметь возможность запускать все пайплайны одной командой, мы создали главный пайплайн, объединяющий все этапы в единую точку входа:

```
@pipeline
def end_to_end_data(
    author_links: list[dict[str, str | list[str]]], ... # Другие параметры...
) -> None:
    wait_for_ids = []
    for author_data in author_links:
        last_step_invocation_id = digital_data_etl(
            user_full_name=author_data["user_full_name"], links=author_
            ➥data["links"]
        )

        wait_for_ids.append(last_step_invocation_id)

    author_full_names = [author_data["user_full_name"] for author_data in
    ➥author_links]
    wait_for_ids = feature_engineering(author_full_names=author_full_names,
    ➥wait_for=wait_for_ids)

    generate_instruct_datasets(...)

        training(...)

        deploy(...)
```

Ради простоты мы включили в него всю логику вплоть до этапа вычисления признаков. Однако в этот главный пайплайн можно легко добавить этапы генерации набора данных с инструкциями, обучения и развертывания модели, реализовав тем самым весь процесс от начала до конца. Такой подход позволяет автоматизировать выполнение всех действий — от сбора данных до развертывания модели.

Для запуска сквозного пайплайна используйте следующую команду poe:

```
poetry poe run-end-to-end-data-pipeline
```

Тем не менее рассмотренный метод не является оптимальным, поскольку объединяет все этапы в монолитный пайплайн, показанный на рис. 11.20 (чего стоит избегать). Обычно лучше оставлять пайплайны изолированными и применять триггеры для их запуска, так как это упрощает понимание, отладку и мониторинг системы.

Пробная версия ZenML Cloud позволяет использовать до трех пайплайнов. Поскольку в нашем проекте их больше, мы обошли это ограничение, объединив все этапы в один пайплайн. Однако если вы развернете ZenML самостоятельно или приобретете лицензию, то сможете запускать один пайплайн из другого,

как показано в приведенном ниже фрагменте кода, где пайплайн обработки признаков запускается после завершения работы ETL-пайплайна сбора данных:

```
from zenml import pipeline, step

@pipeline
def digital_data_etl(user_full_name: str, links: list[str]) -> str:
    user = get_or_create_user(user_full_name)
    crawl_links(user=user, links=links)

trigger_feature_engineering_pipeline(user)

@step
def trigger_feature_engineering_pipeline(user):
    run_config = PipelineRunConfiguration(...)

Client().trigger_pipeline("feature_engineering", run_configuration=run_config)

@pipeline
def feature_engineering(author_full_names: list[str]) -> list[str]:
    ... # Шаги ZenML
```

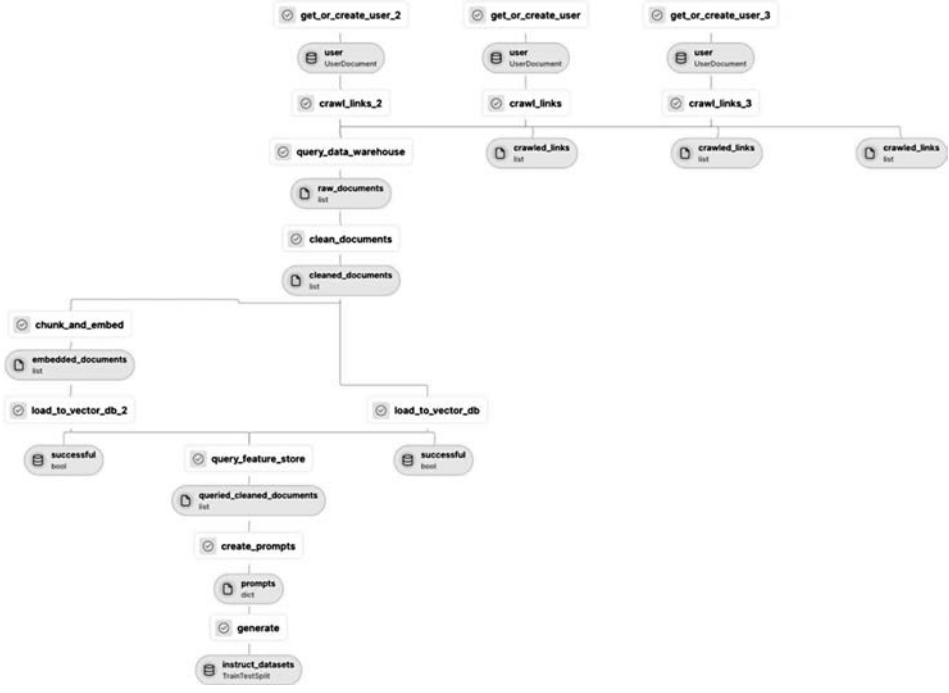


Рис. 11.20. Визуализация сквозного пайплайна на дашборде ZenML

При таком подходе каждый пайплайн будет запускать следующий, как было описано в начале данного раздела. Заметим, что эта возможность характерна не только для ZenML, но и для большинства оркестраторов. Все рассмотренные нами принципы остаются в силе — меняется лишь способ взаимодействия с инструментом.

Мониторинг промптов

Для мониторинга промптов мы будем использовать инструмент Opik (от Comet ML). Однако, как уже было сказано, нас интересует не только входной промпт и сгенерированный ответ.

Мы хотим регистрировать всю последовательность этапов — от получения пользовательского ввода до предоставления финального результата. Прежде чем перейти к реализации мониторинга промптов в системе LLM-двойника, рассмотрим простой пример:

```
from opik import track
import openai
from opik.integrations.openai import track_openai

openai_client = track_openai(openai.OpenAI())

@track
def preprocess_input(text: str) -> str:
    return text.strip().lower()

@track
def generate_response(prompt: str) -> str:
    response = openai_client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[{"role": "user", "content": prompt}]
    )
    return response.choices[0].message.content

@track
def postprocess_output(response: str) -> str:
    return response.capitalize()

@track(name="llm_chain")
def llm_chain(input_text: str) -> str:
    preprocessed = preprocess_input(input_text)
    generated = generate_response(preprocessed)
    postprocessed = postprocess_output(generated)

    return postprocessed

result = llm_chain("Hello, do you enjoy reading the book?")
```

Приведенный выше фрагмент кода отражает типичную структуру LLM-приложений. Основная функция `llm_chain()` принимает запрос и возвращает финальный результат.

Затем идут функции предварительной и постобработки данных. С помощью декоратора `@track()` мы регистрируем входные и выходные данные каждой функции, которые впоследствии будут объединены в единую последовательность (trace). Таким образом мы получим доступ к исходному тексту, сгенерированному ответу и всем промежуточным этапам — все это необходимо для устранения возможных проблем через дашборд Opik.

Последний шаг — добавление к текущей последовательности метаданных, специфичных для вашего конкретного случая использования. Как показано в следующем фрагменте кода, это легко делается с помощью метода `update_current_trace()`, в который можно передать словарь с тегами или другими метаданными, такими как число входных токенов:

```
from opik import track, opik_context

@track
def llm_chain(input_text):
    # Код LLM-цепочки
    # ...

    opik_context.update_current_trace(
tags=["inference_pipeline"],
metadata={
     "num_tokens": compute_num_tokens(...)
},
feedback_scores=[
{
     "name": "user_feedback",
     "value": 1.0,
     "reason": "The response was valuable and correct."
},
{
     "name": "llm_judge_score",
     "value": compute_llm_judge_score(...),
     "reason": "Computing runtime metrics using an LLM Judge."
}
)
```

Эту идею можно расширить, реализовав регистрацию пользовательской обратной связи. Самый распространенный вариант — спросить пользователя, был ли ответ модели полезным и корректным. Альтернативный подход сводится к автоматическому расчету метрик с помощью набора правил или LLM-экспертов.

Теперь реализуем мониторинг промптов в системе LLM-двойника. На рис. 11.21 снова показана архитектура нашего пайплайна инференса. Напомним, что в системе предусмотрено два микросервиса: LLM-микросервис и бизнес-микросервис. Первый имеет узкую область применения: он получает на вход промпт (уже содержащий пользовательский ввод и контекст) и возвращает ответ, который обычно подвергается постобработке. Таким образом, бизнес-микросервис — подходящее место для реализации пайплайна мониторинга, поскольку он координирует весь процесс. Если конкретнее, то реализация Opik будет встроена в сервер FastAPI, разработанный в главе 10.

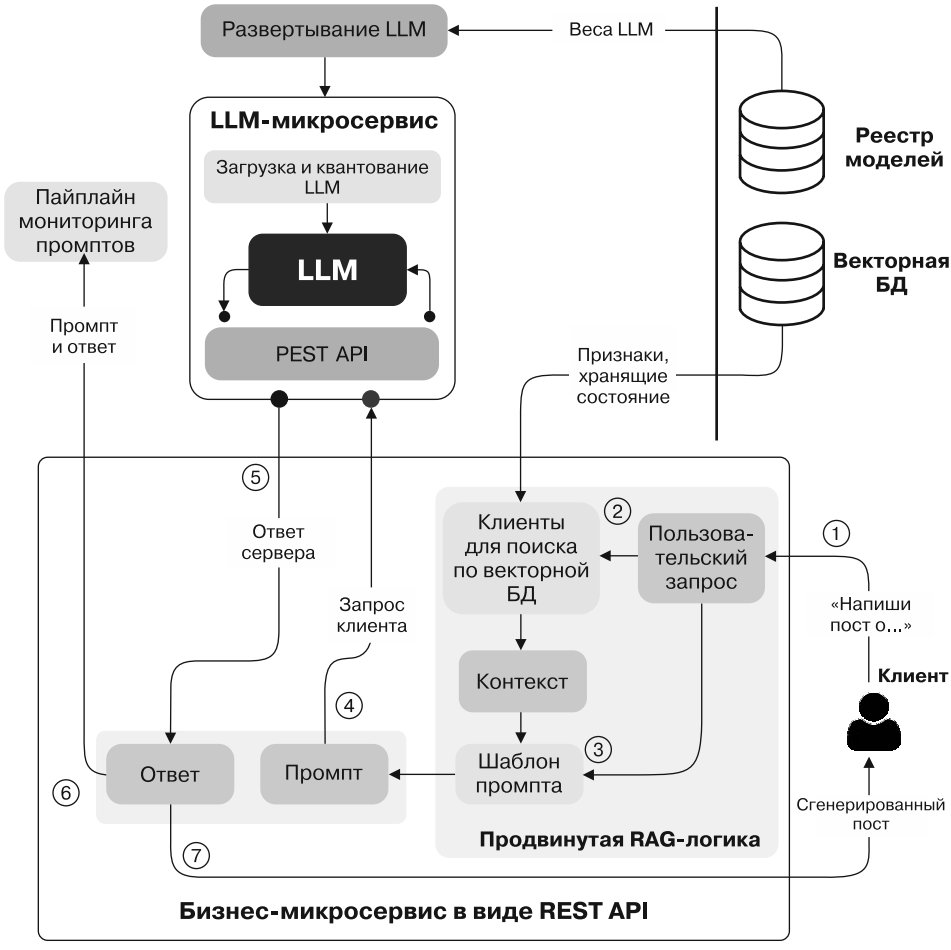


Рис. 11.21. Архитектура пайплайна инференса

Поскольку архитектура нашей системы уже является модульной, интеграция Opik позволяет легко регистрировать всю последовательность этапов обработки пользовательского запроса:

```
from opik import track

@track
def call_llm_service(query: str, context: str | None) -> str:
    llm = LLMInferenceSagemakerEndpoint(...)
    answer = InferenceExecutor(llm, query, context).execute()

    return answer

@track
def rag(query: str) -> str:
    retriever = ContextRetriever()
    documents = retriever.search(query, k=3 * 3)
    context = EmbeddedChunk.to_context(documents)

    answer = call_llm_service(query, context)

    return answer
```

Функция `rag()` представляет собой точку входа в приложение, а остальные этапы обработки реализованы в классах `ContextRetriever` и `InferenceExecutor`. Кроме того, аннотирование функции `call_llm_service()` дает возможность точно зафиксировать промпт, отправленный LLM, и полученный ответ.

Для большей детализации последовательности этапов обработки промпта можно аннотировать функции предварительной и постобработки данных, например функцию `search` в классе `ContextRetriever`:

```
class ContextRetriever:
    ...

    @track

    def search(
        self,
        query: str,
        k: int = 3,
        expand_to_n_queries: int = 3,
    ) -> list:
        query_model = Query.from_str(query)
        query_model = self._metadata_extractor.generate(query_model)
        ... # Остальная часть реализации
```

Или пойти еще дальше и аннотировать методы оптимизации поиска, например метод `generate()` компонента, отвечающего за извлечение метаданных из самозапроса:

```
class SelfQuery:

    @track
    def generate(self, query: str) -> str:
        ...

        return enhanced_query
```

Разработчик сам должен определить, насколько детальным должно быть отслеживание, чтобы обеспечить надлежащую отладку и анализ приложения. Хотя подробный мониторинг является благом, чрезмерная детализация привнесет шум и затруднит анализ. Здесь важно соблюсти баланс. Имеет смысл начать с отслеживания наиболее важных функций (`rag()` и `call_llm_service()`), а затем повышать степень детализации по мере необходимости.

На финальном этапе нужно включить в отслеживаемые последовательности важные метаданные и теги. Для этого мы усовершенствуем функцию `rag()` следующим образом:

```
    @track
    def rag(query: str) -> str:
        retriever = ContextRetriever()
        documents = retriever.search(query, k=3 * 3)
        context = EmbeddedChunk.to_context(documents)

        answer, prompt = call_llm_service(query, context)

        trace = get_current_trace()
        trace.update(tags=["rag"], metadata={
            "model_id": settings.HF_MODEL_ID,
            "embedding_model_id": settings.TEXT_EMBEDDING_MODEL_ID,
            "temperature": settings.TEMPERATURE_INFERENCE,
            "prompt_tokens": compute_num_tokens(prompt),
            "total_tokens": compute_num_tokens(answer),
        }

    )

    return answer
```

Есть три ключевых показателя, за которыми необходимо постоянно следить.

- *Конфигурация модели*. Здесь имеются в виду как сама LLM, так и модели, используемые в RAG-слое. Важно фиксировать идентификаторы моделей и другие параметры, влияющие на процесс генерации ответов, например температуру.
- *Общее число токенов*. Статистика по числу токенов в запросах и ответах критически важна, поскольку напрямую влияет на стоимость инференса. Например, резкий рост среднего числа токенов может указывать на ошибку в системе, требующую проверки.

- *Продолжительность выполнения этапов*. Отслеживание времени выполнения каждого этапа обработки промпта помогает выявлять узкие места в системе. Если задержка при обработке конкретного запроса превысит норму, вы получите отчет, который позволит быстро определить источник проблемы.

Система оповещений

С помощью ZenML вы с легкостью можете реализовать систему оповещений на любой платформе — будь то электронная почта, Discord или Slack. Например, в пайплайн обучения можно добавить функцию обратного вызова, чтобы отправлять уведомление в случае сбоя или успешного завершения выполнения:

```
from zenml import get_pipeline_context, pipeline

@pipeline(on_failure=notify_on_failure)
def training_pipeline(...):
    ...
    notify_on_success()
```

Реализация таких функций достаточно проста. Как видно из приведенных фрагментов кода, достаточно получить экземпляр `alerter` из текущего стека, сформировать сообщение в нужном формате и отправить его в выбранный канал:

```
from zenml.client import Client

alerter = Client().active_stack.alerter

def notify_on_failure() -> None:
    alerter.post(message=build_message(status="failed"))

@step(enable_cache=False)
def notify_on_success() -> None:
    alerter.post(message=build_message(status="succeeded"))
```

ZenML и большинство других оркестраторов упрощает реализацию системы оповещений, поскольку она является важнейшим компонентом инфраструктуры MLOps/LLMOps.

Резюме

Мы начали главу с обсуждения теоретических основ дисциплины DevOps. Затем мы рассмотрели ключевые компоненты и принципы MLOps, после чего показали, чем LLMOps отличается от MLOps, на примере таких стратегий, как мониторинг промптов, защитные механизмы и обратная связь от человека. Вы узнали, почему компании в основном предпочитают не обучать большие языковые модели с нуля, а адаптировать их под свои задачи с помощью инже-

нерии промптов или тонкой настройки. В завершение теоретической части мы рассмотрели устройство CI/CD/CT-пайплайна и три ключевых измерения ML-приложения: код, данные и модель. Мы также обсудили важность реализации слоя мониторинга и оповещений для борьбы с постепенной деградацией модели.

Далее мы затронули процесс развертывания пайплайна LLM-двойника в облаке. Мы подробно разобрали инфраструктуру и развернули MongoDB, Qdrant, ZenML Cloud и все ресурсы AWS, необходимые для работы приложения. Затем мы собрали приложение в Docker-образ и загрузили его в AWS ECR, откуда он будет запускаться на платформе AWS SageMaker.

Заключительным этапом стала интеграция компонентов LLMOps в проект LLM-двойника. Мы реализовали CI/CD-пайплайн через платформу GitHub Actions, а затем — CT-пайплайн с использованием ZenML.

Наконец, мы внедрили пайплайн мониторинга промптов и систему оповещений с помощью Opik от Comet ML и ZenML соответственно. Эти составляющие являются важнейшими MLOps- и LLMOps-компонентами для любого приложения на базе LLM.

Изученная в книге архитектура может быть легко адаптирована под различные проекты. Хотя в качестве примера использовался LLM-двойник, большинство представленных стратегий применимо и к другим приложениям. Чтобы получить совершенно новое приложение, достаточно заменить данные и внести незначительные изменения в код. Помните, что данные — это новая нефть.

Теперь вы знаете, как создать полнофункциональное приложение на основе LLM, поскольку освоили все этапы — от сбора данных и тонкой настройки модели до развертывания LLM-микросервиса и RAG-механизма. В этой книге мы стремились привить вам образ мышления, помогающий решать реальные задачи в области генеративного ИИ. Надеемся, что у нас получилось это сделать и вы готовы отправиться в самостоятельное путешествие в мир инженерии LLM!

Список используемых источников

- GitLab. What is DevOps? | GitLab. https://about.gitlab.com/topics/devops/.
- *Huyen C.* Building a generative AI platform. https://huyenchip.com/2024/07/25/genai-platform.html.
- Lightricks customer story: Building a recommendation engine from scratch. https://www.qwak.com/academy/lightricks-customer-story-building-a-recommendation-engine-from-scratch.
- What LLMOps. Google Cloud. https://cloud.google.com/discover/what-is-llmops?hl=en.

- MLOps: Continuous delivery and automation pipelines in machine learning. Google Cloud. https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning#top_of_page.
- Ml-ops.org. https://ml-ops.org/content/mlops-principles.
- Ml-ops.org. https://ml-ops.org/content/motivation.
- *Mohandas G. M.* Monitoring machine learning systems. Made With ML. https://madewithml.com/courses/mlops/monitoring/.
- *Mohandas G. M.* Testing Machine Learning Systems: Code, Data and Models. Made With ML. https://madewithml.com/courses/mlops/testing/.
- *Preston-Werner T.* Semantic Versioning 2.0.0. Semantic Versioning. https://semver.org/.
- *Ribeiro M. T., Wu T., Guestrin C., Singh S.* Beyond Accuracy: Behavioral Testing of NLP models with CheckList. arXiv.org. https://arxiv.org/abs/2005.04118.
- Wandb. Understanding LLMOps: Large Language Model Operations. Weights & Biases. https://wandb.ai/site/articles/understanding-llmops-large-language-model-operations/.
- Zenml-Io. GitHub — zenml-io/zenml-huggingface-sagemaker (пример обзора пайплайннов ZenML в MLOps: от репозитория моделей Hugging Face до развернутой конечной точки AWS SageMaker). GitHub. https://github.com/zenml-io/zenml-huggingface-sagemaker/tree/main.

Приложение. Принципы MLOps

Создание мощных моделей — лишь один из этапов построения надежной и масштабируемой ML-системы. Для ее реализации необходим комплексный подход к организации всего жизненного цикла машинного обучения. Рассмотрим *шесть принципов*, лежащих в основе дисциплины MLOps. Эти принципы не зависят от используемых инструментов и служат фундаментом при создании надежных ML-систем, обеспечивая их согласованность, устойчивость и масштабируемость на всех этапах разработки.

Начнем с автоматизации, или эксплуатации.

1. Автоматизация, или эксплуатация

В рамках MLOps можно выделить три уровня зрелости приложения.

- *Ручное управление*. На ранних этапах разработки ML-приложения работа носит экспериментальный и итеративный характер. Все шаги, включая подготовку и валидацию данных, обучение и тестирование модели, выполняются вручную, как правило, в Jupyter-блокнотах. Результатом служит программный код, используемый для подготовки данных и обучения моделей.
- *Непрерывное обучение (CT)*. На следующем уровне выполняется автоматизация обучения модели. Непрерывное обучение позволяет автоматически переобучать модель при наступлении определенных событий. Как правило, на этом этапе также автоматизируются подготовка и валидация данных. Весь процесс координируется оркестратором, таким как ZenML, который связывает между собой все компоненты и запускает их по расписанию (скажем, ежедневно) или при наступлении определенных событий (например, при появлении новых данных или при падении уровня производительности).
- *Непрерывная интеграция и доставка (CI/CD)*. На завершающем этапе реализуются CI/CD-пайплайны, обеспечивающие быстрое и надежное развертывание ML-системы в продакшене. Здесь ключевым улучшением становится автоматическое построение, тестирование и развертывание всех компонентов, включая данные, модели и пайплайны обучения. Непрерывная интеграция и доставка обеспечивает быстрое и эффективное внедрение обновлений в различные среды, включая промежуточную (staging) и рабочую (production).

При построении LLM-системы на основе *FTI-архитектуры* мы способны быстро перейти от ручного управления к реализации CI/CD/CT-пайплайна. На рис. П.1 показано, что процесс непрерывного обучения может запускаться при наступлении различных событий — например, при выявлении снижения производительности системой мониторинга или при поступлении новой партии данных. Как видите, схема разделена на две части: верхняя отражает автоматизированные процессы, а нижняя — ручные операции специалистов по анализу данных, выполняемые при проведении экспериментов с методами обработки данных и архитектурами моделей. Добившись нужного улучшения, эта команда отправляет новый код в репозиторий. Такое изменение запускает CI/CD-пайплайн, на котором собираются, тестируются, упаковываются и развертываются новые версии кода в соответствующих пайплайнах FTI.

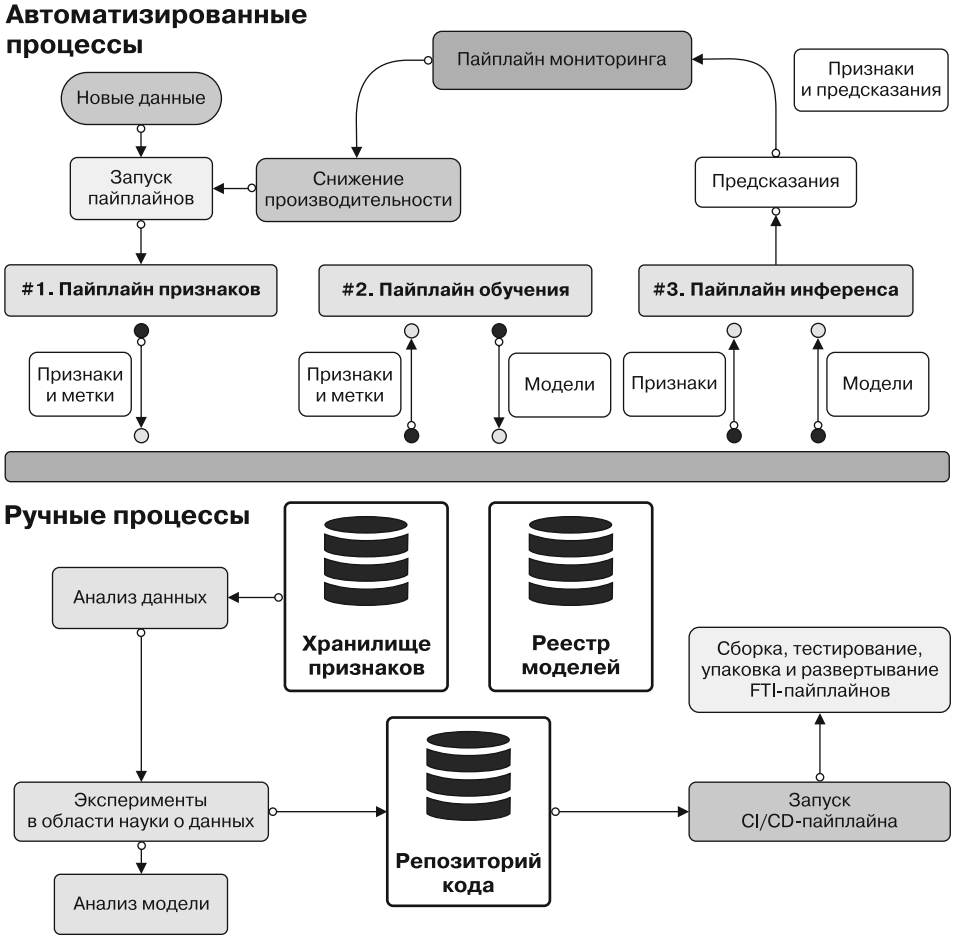


Рис. П.1. Реализация CI/CD/CT-пайплайна поверх FTI-архитектуры

Итак, непрерывное обучение (CT) автоматизирует выполнение FTI-пайплайнов, а непрерывная интеграция и доставка (CI/CD) отвечает за внедрение новых версий кода этих пайплайнов в продакшен.

2. Версионирование

Любое изменение кода, модели или данных приводит к изменению всей ML-системы. Поэтому крайне важно отслеживать и версионировать эти три компонента по отдельности. Для этого можно использовать следующие стратегии.

- *Код*. Код отслеживается с помощью системы Git, которая позволяет создавать снимки состояния кода (коммиты) при каждой корректировке. Кроме того, большинство Git-инструментов (например, GitHub и GitLab) дает возможность публиковать релизы, включающие в себя несколько функций и исправлений. Коммиты идентифицируются с помощью уникальных хешей, трудных для восприятия человеком, тогда как релизы обычно нумеруются по семантической схеме: основная версия, минорная версия, патч-версия. Например, в версии v1.2.3 цифра 1 обозначает основную версию, 2 — минорную, а 3 — патч.
- *Модель*. Хранение, распространение и версионирование моделей, используемых в системе, осуществляется с помощью реестра моделей. Как правило, он также задействует *семантическое версионирование*, но, помимо основной, минорной и патч-версии, может поддерживать обозначения альфа- или бета-релизов, указывающих на статус приложения. На текущем этапе вы, помимо прочего, можете воспользоваться хранилищем ML-метаданных, чтобы прикрепить к сохраненной модели информацию, например, об обучающих данных, архитектуре, производительности, задержке и других важных характеристиках. Такой подход позволяет создать каталог моделей, с которым легко работать как внутри команды, так и на уровне всей компании.
- *Данные*. Версионирование данных — более сложная задача, так как зависит от типа (структурированные или неструктурированные) и объема используемых данных. Например, для отслеживания изменений в наборе структурированных данных можно задействовать SQL-базу с дополнительным столбцом для указания версии. Популярной альтернативой служат такие Git-подобные решения, как *DVC* (Data Version Control), отслеживающие каждое изменение в наборе данных. Также существуют решения, основанные на концепции артефактов, которые добавляют к данным виртуальный слой и позволяют отслеживать каждое изменение. Примерами таких платформ являются Comet ML, Weights & Biases (W&B) и ZenML. Во всех этих случаях данные хранятся либо локально, либо в облачных объектных хранилищах (например, AWS S3). Эти инструменты предоставляют функциональные возможности для структурирования наборов данных и их версий, а также для их отслеживания и обращения к ним.

3. Отслеживание экспериментов

Обучение ML-моделей — это итеративный и экспериментальный процесс. В отличие от традиционной разработки ПО, он предполагает запуск множества параллельных экспериментов, сравнение их результатов по заранее определенным метрикам и выбор наилучшей модели для развертывания в продакшене. Инструменты для отслеживания экспериментов позволяют регистрировать ключевую информацию, включая метрики и визуальные представления предсказаний модели, что упрощает сравнение и выбор оптимального варианта. Наиболее популярными инструментами являются Comet ML, W&B, MLflow и Neptune.

4. Тестирование

При тестировании ML-систем данные, модель и код также необходимо проверять по отдельности. Помимо этого, следует убедиться, что пайплайны обработки признаков, обучения и инференса корректно интегрированы с внешними сервисами (например, хранилищем признаков) и функционируют как единое целое. В экосистеме Python для написания тестов чаще всего используется библиотека `pytest`.

Типы тестов

На разных этапах разработки ПО применяются шесть основных типов тестов (рис. П.2).

- *Модульные тесты (unit tests)*. Проверяют отдельные компоненты, выполняющие одну задачу, например функцию сложения двух тензоров или поиска элемента в списке.
- *Интеграционные тесты (integration tests)*. Проверяют взаимодействие компонентов системы, например пайплайна оценки данных и пайплайна генерации признаков, а также их связь с хранилищами данных и признаков.
- *Системные тесты (system tests)*. Играют ключевую роль в цикле разработки, поскольку проверяют работоспособность системы в целом. Такие тесты оценивают весь ее функционал, включая производительность, безопасность и пользовательский опыт. Примером может служить тестирование всех этапов работы ML-пайплайна, от загрузки данных до обучения модели и инференса, чтобы убедиться, что выходные данные корректны.
- *Приемочные тесты (acceptance tests)*. Известны также как UAT (user acceptance testing — «приемочное пользовательское тестирование»), проверяют соответствие системы заданным требованиям и ее готовность к развертыванию.

- *Регрессионные тесты (regression tests)*. Проводятся для выявления ранее устраненных ошибок, чтобы убедиться, что новые изменения не возвращают старые проблемы.
- *Стресс-тесты (stress tests)*. Проверяют устойчивость системы в условиях экстремальной нагрузки или ограничения ресурсов. Они позволяют выявить точки отказа и убедиться в способности системы справляться с неожиданными всплесками пользовательской активности или неблагоприятными ситуациями без сбоев.

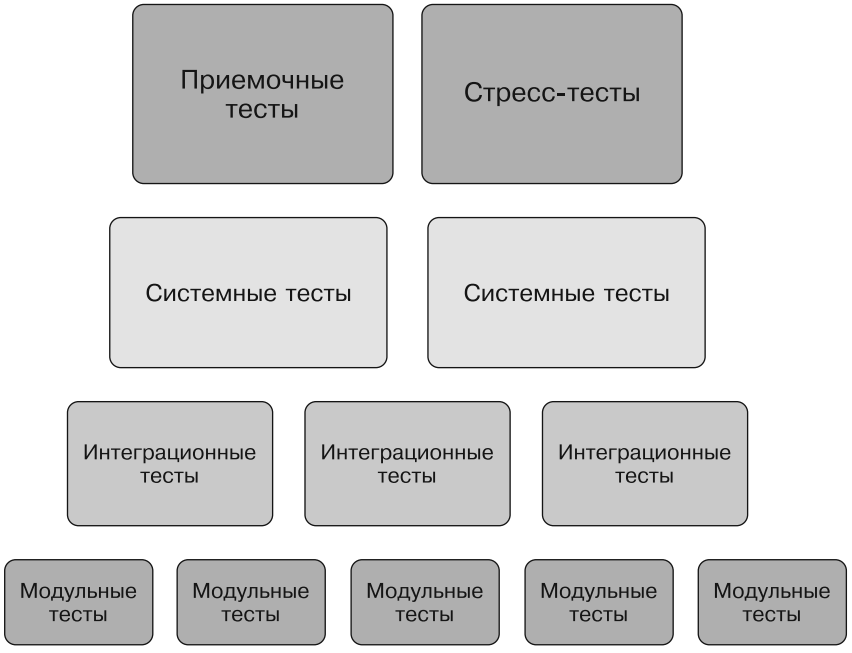


Рис. П.2. Типы тестов

Обратите внимание, что на схеме не указаны регрессионные тесты. Это связано с тем, что они не являются отдельным этапом, а сопровождают проведение всех остальных тестов и позволяют убедиться в том, что вносимые изменения не приводят к повторному возникновению ранее устраненных ошибок.

Предмет тестирования

Как правило, при написании тестов компонент рассматривается в качестве «черного ящика»: мы подаем в него входные данные и проверяем, соответствуют ли выходные данные ожидаемым. Таким образом, предметом тестирования обычно являются:

- *входные данные* — типы, формат, длина, граничные случаи (минимальные, максимальные, слишком короткие или длинные значения и т. д.);
- *выходные данные* — типы, формат, исключения, промежуточные и финальные результаты.

Примеры тестов

При тестировании кода можно опираться на стандарты классической программной инженерии. Ниже приведено несколько примеров тестов, которые стоит включить в рамки модульного тестирования для лучшего понимания того, что именно необходимо проверить на данном этапе. Например, чтобы оценить корректность очистки предложения.

Можно также протестировать алгоритм фрагментации, применив его к различным предложениям и фрагментам разного размера.

Под *тестированием данных*, как правило, подразумевается проверка их корректности. Код, проверяющий корректность данных, обычно запускается либо при загрузке исходных данных из хранилища, либо после вычисления признаков — то есть является частью пайплайна признаков. Таким образом, реализуя интеграционные или системные тесты для этого пайплайна, вы можете убедиться, что система правильно реагирует как на корректные, так и на некорректные данные.

Проверка корректности данных во многом зависит от особенностей вашего приложения и типа данных. Например, при работе с табличными данными можно проверить отсутствие пропущенных значений, соответствие категориальных переменных ожидаемым значениям или что числа с плавающей запятой всегда положительные. Для неструктурированных данных, таких как текст, проверяются длина фрагмента, кодировка, язык, наличие специальных символов, а также грамматические ошибки.

Тестирование моделей представляет наибольшую сложность, поскольку обучение модели — наименее детерминированный процесс в ML-системе. В отличие от традиционного программного обеспечения ML-системы могут завершать свою работу без каких-либо ошибок, но при этом выдавать некорректные результаты, обнаружить которые можно только при проведении оценки или тестирования. Типичными методами тестирования модели являются:

- проверка формы входных и выходных тензоров модели;
- проверка того, что значение функции потерь уменьшается после одной (или нескольких) итерации цикла обучения;
- проверка способности модели переобучаться на небольшой выборке с приближением значения функции потерь к нулю;

- проверка работоспособности пайплайна обучения на всех поддерживаемых устройствах, таких как CPU и GPU;
- проверка корректности логики ранней остановки и сохранения контрольных точек.

Все тесты выполняются внутри CI-пайплайна. Ресурсоемкие тесты, например модельные, можно выполнять только при наступлении определенных событий, в частности при изменении кода модели.

На другом конце спектра находится *поведенческое тестирование*, в рамках которого модель рассматривается как черный ящик. При этом анализируются только входные и ожидаемые выходные данные. Такой подход делает методику поведенческого тестирования независимой от модели. Фундаментальной работой в этой области считается статья *Beyond Accuracy: Behavioral Testing of NLP Models with CheckList*, с которой мы рекомендуем ознакомиться тем, кто хочет глубже погрузиться в тему. Если вкратце, то авторы данной статьи предлагают проверять модель с помощью трех типов тестов. В качестве примера рассмотрим модель, извлекающую подлежащее из предложения.

- *Инвариантность*. Изменения во входных данных не должны влиять на результат. Вот пример с подстановкой синонимов:

```
model(text="Достижения в области ИИ быстро меняют мир.")
# выход: ai

model(text="Прогресс в области ИИ быстро меняет мир.")
# выход: ai
```

- *Направленная изменяемость*. Изменения во входных данных должны влиять на результат. Ниже приведен пример, в котором изменение вывода является ожидаемым следствием изменения входа:

```
model(text="Глубокое обучение, используемое для анализа эмоций.")
# выход: deep-learning

model(text="Глубокое обучение, используемое для обнаружения объектов.")
# выход: deep-learning

model(text="RNN для анализа эмоций.")
# выход: rnn
```

- *Минимальная функциональность*. Простейшие сочетания входных и ожидаемых выходных данных. Ниже приведены примеры, которые модель всегда должна обрабатывать корректно:

```
model(text="Обработка естественного языка — это следующая большая волна
в области машинного обучения.")
# выход: nlp
```

```
model(text="MLOps — это следующая большая волна в области машинного обучения.")
# выход: mlops

model(text="Речь идет о графовых нейронных сетях.")
# выход: gnn
```



Для более подробного ознакомления с темой тестирования рекомендуем статью Гоку Мохандаса Testing Machine Learning Systems: Code, Data and Models (https://madewithml.com/courses/mlops/testing/).

5. Мониторинг

Мониторинг жизненно важен для любой ML-системы, развернутой в продакшене. Традиционное ПО работает по правилам, поэтому его поведение является детерминированным. Однако ML-модели устроены иначе: мы не описываем их поведение напрямую, а обучаем их на данных для получения вероятностного решения. То есть модели всегда подвержены определенной деградации из-за различий между данными, используемыми в продакшене, и теми, которые были задействованы при обучении.

Такие ситуации неизбежны, и вместо того, чтобы пытаться их предотвратить, нам следует выстроить стратегию своевременного обнаружения и устранения ошибок. Система мониторинга отслеживает снижение производительности модели и подает сигнал тревоги, когда модель нуждается в переобучении (ручным, автоматическим или гибридным способом).

Зачем переобучать модель? Поскольку снижение уровня производительности вызвано рассогласованием между обучающими данными и теми, которые используются в продакшене, единственный способ восстановить качество — это адаптировать модель к новым реалиям, переобучив ее на более актуальном наборе.

Поскольку обучение — это ресурсозатратная операция, существуют приемы, позволяющие избежать переобучения модели. Однако прежде, чем перейти к ним, давайте кратко обсудим показатели, которые можно отслеживать для оценки состояния ML-системы.

Логи

Подход к ведению логов довольно прост. В них следует:

- документировать конфигурации системы;
- записывать запросы, результаты и промежуточные выходы;
- фиксировать начало работы компонентов, ее завершение, сбои и т. д.;
- снабжать каждую запись метками, позволяющими отследить ее источник в системе.

Хотя при таком подходе объем логов может быстро расти, для их эффективного управления существуют ИИ-инструменты, выполняющие автоматический анализ и поиск выбросов.

Метрики

Для количественной оценки состояния системы необходимо определить набор метрик, отражающих различные аспекты работы приложения, такие как инфраструктура, данные и модель.

Системные метрики

К системным метрикам относятся показатели, отражающие качество сервиса (задержка, пропускная способность, частота ошибок) и состояние инфраструктуры (загрузка CPU/GPU, объем памяти и т. д.). Эти метрики универсальны и применяются как в традиционном ПО, так и в ML-системах, поскольку позволяют убедиться в том, что инфраструктура функционирует корректно и обеспечивает должный уровень обслуживания пользователей.

Модельные метрики

Одного лишь контроля состояния системы недостаточно, чтобы выявить проблемы в самой модели, поэтому нам необходимо отслеживать такие показатели ее производительности, как точность, F1-мера, а также бизнес-метрики вроде ROI или частоты кликов.

При этом анализ совокупных показателей качества за весь период эксплуатации модели часто оказывается неэффективным. Вместо этого важно оценивать производительность за определенные временные интервалы, релевантные конкретному приложению, например за час. На практике это означает разбиение входных данных на окна, для каждого из которых вычисляются и агрегируются метрики. Такие «скользящие» показатели позволяют получить более точное представление о состоянии системы и быстрее выявлять проблемы на раннем этапе — прежде чем их замаскируют накопленные исторические данные.

Мы не всегда можем получить доступ к эталонным выходным значениям, необходимым для оценки производительности модели на рабочих данных. Особенно сложно это в ситуациях, когда присутствует значительная задержка в получении результатов или когда реальные данные требуют ручной разметки. Чтобы справиться с подобной проблемой, можно использовать приближенный сигнал, позволяющий оценить производительность модели, или разметить для этого небольшую часть потоковых данных. В контексте мониторинга ML-систем такой приближенный сигнал называют *прокси-метрикой*. Как правило, она реализуется с помощью методов обнаружения дрейфа, которые мы рассмотрим далее.

Дрейф

Дрейф — это прокси-метрика, помогающая выявлять потенциальные проблемы с моделью, развернутой в продакшене, без использования эталонных ответов (меток). В табл. П.1 представлены три типа дрейфа.

Таблица П.1. Взаимосвязи между изменениями данных, модели и кода

Что изменяется	Описание	Формулировка дрейфа
X	Входы (признаки)	Дрейф данных $\rightarrow P(X) \neq P_{\text{эталон}}(X)$
y	Выходы (эталонные ответы/метки)	Дрейф целевой переменной $\rightarrow P(y) \neq P_{\text{эталон}}(y)$
$P(y\|X)$	Зависимость y от X	Дрейф концепции $\rightarrow P(y\|X) \neq P_{\text{эталон}}(y\|X)$

Дрейф данных. *Дрейф данных (data drift)*, также называемый дрейфом признаков или ковариационным сдвигом, возникает, когда распределение рабочих данных отклоняется от распределения обучающих данных (рис. П.3). Это означает, что модель неспособна адекватно обрабатывать новые входные признаки, что может привести к снижению качества предсказаний[1]. Причинами такого дрейфа являются как естественные изменения, так и системные сбои, например вызванные отсутствием данных, ошибками в пайплайне их обработки или изменением их схемы[2].

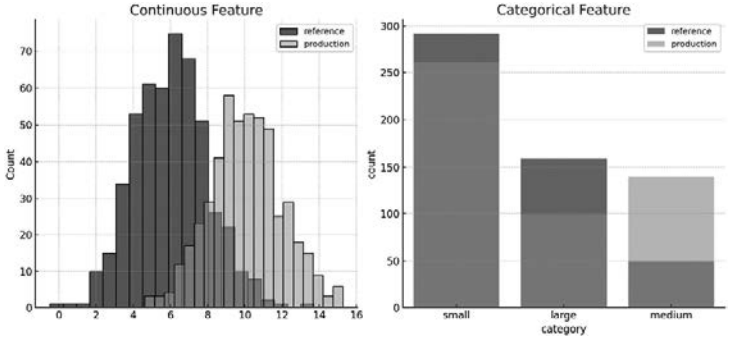


Рис. П.3. Примеры дрейфа данных

Когда в данных возникает дрейф, ухудшение производительности модели может проявиться не сразу, особенно если она успешно справляется с интерполяцией.

---

[1] Например, автопилот учили работать в городе с разметкой, а машина выезжает на проселочную дорогу. — *Примеч. науч. ред.*

[2] Например, поле с возрастом пользователя, влияющее на выдачу кредита, стало null или было удалено. — *Примеч. науч. ред.*

Однако переобучать модель стоит именно на этом этапе, не дожидаясь снижения качества ее работы.

Дрейф целевой переменной. Помимо изменений во входных данных (дрейфа данных), мы можем столкнуться с изменениями в распределении выводов. Обычно это выражается в преобразовании формы распределения или в появлении/исчезновении классов (в задачах классификации). Хотя переобучение модели может смягчить деградацию ее производительности, вызванную *дрейфом целевой переменной* (target drift), во многих случаях этого получится избежать, адаптировав этапы обработки выходных данных и выходной слой модели под новую схему выходных классов.

Например, если ваш классификатор определяет, содержит ли изображение животных или людей, а в новой выборке появляются фотографии зданий, то вы можете либо адаптировать модель для корректной обработки неизвестного класса, либо изменить ее выходной слой, добавив новый класс для будущих предсказаний.

Дрейф концепции. Изменяться могут не только входные и выходные данные, но и зависимость между ними. Это явление, известное как *дрейф концепции* (concept drift), приводит к снижению эффективности модели из-за устаревания закономерностей, лежащих в основе ее предсказаний. Как показано на рис. П.4, дрейф концепции проявляется:

- постепенно — по мере накопления изменений;
- внезапно — из-за наступления некоего внешнего события;
- периодически — в силу сезонных или циклических факторов.

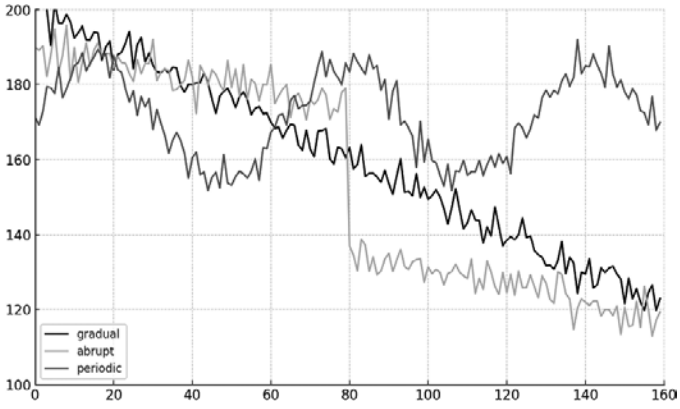


Рис. П.4. Примеры дрейфа концепции

Такой дрейф может возникнуть, например, при использовании модели в другой географической зоне. Допустим, вы разрабатывали модель для предсказания вероятности покупки автомобиля и обучили ее на данных об американском рынке.

Если теперь вы задействуете ее в Европе, жители которой предпочитают более компактные автомобили, между таким признаком, как «размер автомобиля», и вероятностью покупки возникнет дрейф. Разумеется, дрейф концепции может быть гораздо менее выраженным, чем в этом упрощенном примере.



Разные типы дрейфа могут происходить одновременно, что значительно затрудняет выявление их причин.

Обнаружение и измерение дрейфа. Теперь, когда мы разобрались с различными типами дрейфа, давайте обсудим способы его обнаружения и измерения. Для этого необходимо использовать два типа окон (временные интервалы данных).

- *Эталонное окно (reference window)*. Содержит выборку точек данных, используемую в качестве основы для сравнения с распределением рабочих данных при выявлении дрейфа. Обычно формируется из обучающего набора.
- *Тестовое окно (test window)*. Содержит точки данных, собранные во время эксплуатации ML-системы в продакшене. Сравнение этого окна с эталонным позволяет подтвердить наличие дрейфа.

Для измерения дрейфа применяются статистические критерии, проверяющие наличие изменений в распределении между двумя окнами. Например, для отслеживания непрерывного признака можно использовать *критерий Колмогорова — Смирнова* — одномерный тест, который необходимо проводить отдельно для каждого отслеживаемого признака. Для отслеживания категориальных переменных применяется одномерный критерий хи-квадрат, позволяющий проверить соответствие частотных распределений событий в рабочих и эталонных данных:

```
from alibi_detect.cd import KSDrift

cd = KSDrift(X_ref, p_val=.05, preprocess_fn=preprocess_fn, input_shape=(max_len,))
```

При работе с текстовыми данными в виде эмбеддингов необходимо учитывать многомерное распределение, так как LLM работают именно с такими представлениями. Один из популярных подходов заключается в том, чтобы взять эмбеддинги для тестового и эталонного окна, применить к ним метод снижения размерности, а затем — алгоритм измерения различий между распределениями, например MMD (maximum mean discrepancy — «максимальное среднее расхождение»). Этот непараметрический метод измеряет расстояние между распределениями путем вычисления разницы между средними эмбеддингами двух окон:

```
from alibi_detect.cd import MMDDrift

cd = MMDDrift(x_ref, backend='pytorch', p_val=.05)
preds = cd.predict(x)
```

Мониторинг и наблюдаемость

Мониторинг предполагает сбор и визуализацию данных, тогда как наблюдаемость (observability) позволяет оценивать состояние системы через анализ ее входов и выходов. Например, мониторинг дает возможность отслеживать конкретную метрику для выявления потенциальных проблем.

В то же время система считается наблюдаемой, если она генерирует информативные данные о своем внутреннем состоянии, позволяющие находить причины сбоев.

Оповещения

После того как вы определились с метриками, необходимо настроить механизм оповещений. Для этого можно:

- задать пороговые значения метрик, например, чтобы система посылала уведомления при падении уровня точности прогнозов классификатора ниже 0,8;
- настроить *p*-значения в статистических тестах для выявления дрейфа. Чем ниже *p*-значение, тем выше уверенность в том, что распределение рабочих данных отличается от эталонного.

Выбор пороговых и p-значений зависит от конкретного приложения. Важно подобрать их правильно, поскольку слишком чувствительная система оповещений может быть подвержена ложным срабатываниям, из-за которых вы будете либо постоянно реагировать на несущественные сигналы, либо вовсе перестанете обращать на них внимание. Наиболее распространенными каналами для уведомления заинтересованных сторон являются платформы Slack, Discord, PagerDuty или электронная почта. К заинтересованным сторонам в данном случае относятся инженеры, менеджеры и любые другие специалисты, ответственные за работу системы.

Способ реагирования на сигнал зависит от его характера. Однако прежде, чем предпринимать какие-либо шаги, важно проанализировать причину этого сигнала. Следует выяснить, какая именно метрика привела к срабатыванию системы оповещения, каким было ее значение и в какой момент времени это произошло, а также собрать дополнительные сведения, имеющие значение в контексте вашего приложения.

При снижении производительности модели у вас может возникнуть естественное желание переобучить ее. Это довольно ресурсоемкая операция, поэтому прежде, чем ее проводить, необходимо убедиться в корректности данных, проверить, не изменилась ли их схема, а также исключить вероятность того, что тревога вызвана единичным выбросом. Если ни одна из этих причин не подтвердится, вы можете запустить пайплайн обучения и переобучить модель на обновленном наборе данных для устранения выявленного дрейфа.

6. Воспроизводимость

Воспроизводимость означает, что каждый процесс внутри вашей ML-системы должен давать идентичный результат при одинаковых входных данных. Этот принцип включает два основных требования.

Во-первых, вы всегда должны точно знать, какие именно входные данные применялись при выполнении того или иного процесса. Так, при обучении модели задействуется множество гиперпараметров, поэтому вам необходимо отслеживать, какие именно ресурсы использовались для получения результата, например версии наборов данных и конфигурации модели.

Во-вторых, обеспечение воспроизводимости осложняется недетерминированной природой ML-процессов. Так, при обучении модели с нуля ее веса инициализируются случайным образом. Поэтому, даже если вы задействуете один и тот же набор данных и те же гиперпараметры, итоговые модели могут демонстрировать разную производительность. Эта проблема решается путем задания начального значения (seed) генератора случайных чисел, поскольку цифровые системы на самом деле генерируют не подлинно случайные, а псевдослучайные числа. Следовательно, используя затравку, мы гарантируем, что последовательность псевдослучайных чисел всегда будет одинаковой. Аналогичные ситуации могут возникать и на этапе генерации признаков, например при заполнении пропущенных значений случайными числами или при удалении случайно выбранных данных или меток. Поэтому по возможности старайтесь делать свои процессы максимально детерминированными, а в случаях, требующих привнесения элемента случайности, используйте контролируемое начальное значение генератора случайных чисел.

Read IT Club

Комьюнити рецензентов
и переводчиков ИТ-литературы

Миссия участников клуба — обеспечить высокое качество профессиональной переводной литературы в России. «Книжные дебагеры» проверяют корректность терминологии и подписей на схемах и иллюстрациях, чтобы сделать книги более понятными русскоязычному читателю. Стать участником Read IT Club может любой ИТ-специалист, готовый поделиться опытом с сообществом.

присоединиться к нам



Джон Берриман, Альберт Циглер

ПРОМПТ-ИНЖИНИРИНГ ДЛЯ LLM. ИСКУССТВО ПОСТРОЕНИЯ ПРИЛОЖЕНИЙ НА ОСНОВЕ БОЛЬШИХ ЯЗЫКОВЫХ МОДЕЛЕЙ

Большие языковые модели (LLM) кардинально меняют мир, обещая автоматизацию различных задач и решение сложных проблем. Программные приложения последнего поколения используют эти модели в качестве строительных блоков для создания новых возможностей практически во всех областях, но, чтобы пользоваться этими возможностями, нужны навыки. Книга расскажет о науке и искусстве промпт-инжиниринга, и эти знания помогут вам раскрыть истинный потенциал LLM.

Опытнейшие эксперты Джон Берриман и Альберт Циглер поделятся приемами эффективного взаимодействия с ИИ и расскажут, как представлять свои идеи в формате, удобном для языковой модели. Освоив философские основы и практические приемы, вы обретете знания и уверенность, необходимые для создания приложений нового поколения на базе LLM.

SPRINT book

Майкл Альбада

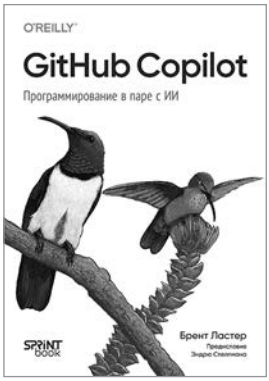
СОЗДАНИЕ ПРИЛОЖЕНИЙ С ИИ-АГЕНТАМИ. ПРОЕКТИРОВАНИЕ И ВНЕДРЕНИЕ МУЛЬТИАГЕНТНЫХ СИСТЕМ



Генеративный ИИ произвел революцию в том, как организации решают проблемы, ускорив переход от концепции к прототипу и готовому решению. По мере роста возможностей моделей наблюдается появление нового паттерна проектирования — ИИ-агентов. Сочетание инструментов, знаний, памяти и механизмов обучения с продвинутыми фундаментальными моделями позволяет выстраивать последовательности из нескольких инференсов моделей для решения неоднозначных и сложных задач. Агенты — от помощников разработчика до исследовательских и аналитических агентов — уже ускоряют работу команд и целых организаций. Хотя эти агенты повышают эффективность, они часто требуют тщательного планирования и многочисленных правок, а их развертывание остается проблемой во многих организациях, особенно при стремительном развитии технологий и исследований.

Автор Майкл Альбада предлагает практический и основанный на исследованиях подход к проектированию и внедрению одно- и мультиагентных систем. Он упрощает сложности и вооружает инструментами для эффективного перехода от концепций к решению.

SPRINT book



Брент Ластер

GITHUB COPILOT. ПРОГРАММИРОВАНИЕ В ПАРЕ С ИИ

Освойте навыки программирования в тесном сотрудничестве с генеративным ИИ, используя GitHub Copilot — пожалуй, самый популярный ИИ-инструмент разработчика. Брент Ластер, опытный девелопер и преподаватель, описал реальные сценарии из ежедневной практики и тем самым показал, что GitHub Copilot — это не просто подсказки в редакторе, а полноценный партнер по программированию, который может в разы ускорить вашу работу, если знать, как с ним взаимодействовать.

Книга идеально подходит разработчикам, тестировщикам, DevOps-инженерам и всем, кто так или иначе причастен к созданию ПО, независимо от уровня опыта. Вы освоите не только базовые функции, но и более мощные возможности: Copilot Edits, Copilot Agent и даже Copilot Vision, узнаете, как создавать собственные расширения и интегрировать их в свой рабочий процесс, независимо от выбранного языка программирования.

SPRINT book