Содержание лабораторных работ 01-12

Лабораторная работа 1-2.1

Подготовка рабочего окружения для MLOps

1.1. Теоретическая часть 1.1.1. Введение в Conda и виртуальные окружения 1.1.2. Введение в Docker 1.2. Практическая часть 1.2.1. Установка и настройка Miniconda 1.2.2. Создание виртуального окружения 1.2.3. Установка Docker и базовые операции 1.2.4. Запуск JupyterLab в контейнере 1.3. Результаты и выводы

Лабораторная работа 1-2.2

Основы трекинга экспериментов с MLflow

2.1. Теоретическая часть 2.1.1. Введение в MLOps и MLflow 2.1.2. Ключевые концепции MLflow Tracking 2.2. Практическая часть 2.2.1. Установка MLflow и запуск сервера 2.2.2. Написание скрипта с трекингом эксперимента 2.2.3. Анализ результатов в MLflow UI 2.3. Результаты и выводы

Лабораторная работа 3-4.1

Знакомство с платформой Hugging Face Hub

3.1. Теоретическая часть 3.1.1. Введение в Hugging Face и экосистему 3.1.2. Ключевые концепции платформы 3.1.3. Задача текстовой классификации 3.2. Практическая часть 3.2.1. Установка необходимых библиотек 3.2.2. Работа с Hugging Face Hub через веб-интерфейс 3.2.3. Программная работа с Hugging Face Hub 3.2.4. Сохранение локальных копий 3.3. Результаты и выводы

Лабораторная работа 3-4.2

Тонкая настройка модели для текстовой классификации

4.1. Теоретическая часть 4.1.1. Тонкая настройка (Fine-tuning) 4.1.2. Архитектура Transformer для классификации 4.1.3. Процесс обучения 4.2. Практическая часть 4.2.1. Подготовка среды и данных 4.2.2. Предобработка данных 4.2.3. Настройка модели и обучения 4.2.4. Обучение модели 4.2.5. Оценка модели 4.3. Результаты и выводы

Лабораторная работа 3-4.3

Интеграция с MLflow для трекинга экспериментов

5.1. Теоретическая часть 5.1.1. Интеграция MLflow с Transformers 5.1.2. Компоненты трекинга для NLP 5.1.3. Стратегии логирования 5.2. Практическая часть 5.2.1. Подготовка среды и конфигурация 5.2.2. Модификация скрипта обучения 5.2.3. Запуск и мониторинг эксперимента 5.2.4. Дополнительные эксперименты 5.2.5. Анализ результатов 5.3. Результаты и выводы

Лабораторная работа 5-6.1

Работа с векторными базами данных (ChromaDB)

6.1. Теоретическая часть 6.1.1. Векторные базы данных 6.1.2. ChromaDB 6.1.3. Применение в NLP 6.2. Практическая часть 6.2.1. Установка и настройка окружения 6.2.2. Подготовка данных и модели 6.2.3. Создание и настройка базы данных 6.2.4. Подготовка и векторизация данных 6.2.5. Семантический поиск 6.2.6. Расширенная функциональность 6.2.7. Сохранение и загрузка БД 6.2.8. Интеграция с реальными данными 6.3. Результаты и выводы

Лабораторная работа 5-6.2

Работа с графовыми базами данных (Neo4j)

7.1. Теоретическая часть 7.1.1. Графовые базы данных 7.1.2. Neo4j 7.1.3. Применение в управлении знаниями 7.2. Практическая часть 7.2.1. Запуск Neo4j в Docker 7.2.2. Подключение к Neo4j Browser 7.2.3. Основы языка Cypher 7.2.4. Создание онтологии предметной области 7.2.5. Выполнение базовых запросов 7.2.6. Сложные запросы и анализ 7.2.7. Работа с реальными данными 7.2.8. Визуализация и экспорт 7.3. Результаты и выводы

Лабораторная работа 7-8.1

Знакомство с онтологиями в Protégé

8.1. Теоретическая часть 8.1.1. Онтологии в компьютерных науках 8.1.2. Язык OWL 8.1.3. Protégé 8.2. Практическая часть 8.2.1. Установка и настройка Protégé 8.2.2. Загрузка и изучение образовательной онтологии 8.2.3. Анализ структуры онтологии 8.2.4. Работа с классами и свойствами 8.2.5. Использование reasoner для логического вывода 8.2.6. Сохранение и экспорт онтологии 8.2.7. Документирование онтологии 8.3. Результаты и выводы

Лабораторная работа 7-8.2

Работа с SPARQL-запросами

9.1. Теоретическая часть 9.1.1. Язык SPARQL 9.1.2. Apache Jena Fuseki 9.1.3. Структура SPARQL-запроса 9.2. Практическая часть 9.2.1. Установка и запуск Apache Jena Fuseki 9.2.2. Загрузка онтологии в Fuseki 9.2.3. Написание базовых SPARQL-запросов 9.2.4. Сложные запросы с фильтрацией 9.2.5. CONSTRUCT-запросы для создания данных 9.2.6. Работа с онтологией через RDFLib 9.2.7. Создание комплексных отчетов 9.2.8. Интеграционные тесты 9.3. Результаты и выводы

Лабораторная работа 7-8.3

Извлечение данных с помощью LLM

10.1. Теоретическая часть 10.1.1. Генерация SPARQL через NL-to-SPARQL 10.1.2. Архитектура решения 10.1.3. Оценка качества 10.2. Практическая часть 10.2.1. Настройка окружения и подключение к LLM 10.2.2. Базовая генерация SPARQL-запросов 10.2.3. Валидация и выполнение запросов 10.2.4. Few-shot обучение через промпты 10.2.5. Интеграция с семантическим валидатором 10.2.6. Оценка качества и метрики 10.2.7. Создание демонстрационного интерфейса 10.3. Результаты и выводы

Лабораторная работа 9-10.1

Развертывание ML-моделей с FastAPI

11.1. Теоретическая часть 11.1.1. FastAPI для ML-сервисов 11.1.2. Архитектура ML-сервиса 11.1.3. Производственные практики 11.2. Практическая часть 11.2.1. Установка и настройка окружения 11.2.2. Создание моделей данных с Pydantic 11.2.3. Реализация ML-модели для обслуживания 11.2.4. Создание эндпоинтов API 11.2.5. Создание основного приложения 11.2.6. Тестирование API 11.2.7. Запуск и использование API 11.3. Результаты и выводы

Лабораторная работа 9-10.2

Контейнеризация ML-сервиса с Docker

12.1. Теоретическая часть 12.1.1. Контейнеризация ML-приложений 12.1.2. Dockerfile для Python-приложений 12.1.3. Многоступенчатая сборка 12.2. Практическая часть 12.2.1. Подготовка приложения к контейнеризации 12.2.2. Создание Dockerfile 12.2.3. Создание .dockerignore 12.2.4. Сборка Docker-образа 12.2.5. Запуск контейнера 12.2.6. Оптимизация образа 12.2.7. Управление контейнером и очистка 12.3. Результаты и выводы

Лабораторная работа 9-10.3

Тестирование API через Swagger UI

13.1. Теоретическая часть 13.1.1. Swagger/OpenAPI для тестирования API 13.1.2. Стратегии тестирования REST API 13.1.3. Критические аспекты тестирования ML-сервисов 13.2. Практическая часть 13.2.1. Подготовка к тестированию 13.2.2. Базовое тестирование эндпоинтов 13.2.3. Комплексное тестирование эндпоинта предсказания 13.2.4. Негативное тестирование и валидация ошибок 13.2.5. Тестирование эндпоинта информации о модели 13.2.6. Сравнение с curl-запросами 13.2.7. Создание тестового отчета 13.2.8. Производительность и нагрузочное тестирование 13.3. Результаты и выводы

Лабораторная работа 11-12.1

Построение прототипа RAG-системы

14.1. Теоретическая часть 14.1.1. Архитектура RAG-систем 14.1.2. Ключевые компоненты RAG-пайплайна 14.1.3. Метрики оценки RAG-систем 14.2. Практическая часть 14.2.1. Подготовка окружения и данных 14.2.2. Реализация модуля поиска (Retriever) 14.2.3. Реализация модуля генерации (Generator) 14.2.4. Интеграция компонентов в RAG-пайплайн 14.2.5. Тестирование системы 14.3. Результаты и выводы

Лабораторная работа 11-12.2

Настройка языковой модели в качестве генератора

15.1. Теоретическая часть 15.1.1. Архитектуры языковых моделей для генерации 15.1.2. Методы оптимизации для production 15.1.3. Критерии выбора модели для RAG 15.2. Практическая часть 15.2.1. Сравнение различных моделей-генераторов 15.2.2. Оптимизация генерации с помощью продвинутых техник 15.2.3. Сравнительный анализ моделей 15.2.4. Интеграция оптимизированного генератора 15.2.5. Создание отчета о настройке генератора 15.3. Результаты и выводы

Лабораторная работа 11-12.3

Создание конвейера RAG-системы

16.1. Теоретическая часть 16.1.1. Архитектура полного RAG-конвейера 16.1.2. Критические аспекты production-систем 16.1.3. Стратегии оптимизации производительности 16.2. Практическая часть 16.2.1. Проектирование архитектуры конвейера 16.2.2. Реализация основного конвейера 16.2.3. Создание FastAPI-сервиса для конвейера 16.2.4. Тестирование и валидация конвейера 16.2.5. Демонстрация работы системы 16.3. Результаты и выводы

Лабораторная работа 11-12.4

Оценка качества работы системы

17.1. Теоретическая часть 17.1.1. Метрики оценки RAG-систем 17.1.2. Метрики для ретривера 17.1.3. Метрики для генератора 17.1.4. Human Evaluation 17.2. Практическая часть 17.2.1. Подготовка тестового датасета 17.2.2. Реализация системы оценки ретривера 17.2.3. Реализация системы оценки генератора 17.2.4. Комплексная оценка RAG-системы 17.2.5. Визуализация и анализ результатов 17.2.6. Запуск комплексной оценки 17.3. Результаты и выводы

Содержание сквозного проекта (Лабораторные работы 13-18)

Лабораторная работа 13

Проектирование системы управления знаниями

18.1. Выбор темы и постановка задачи 18.1.1. Анализ предметной области 18.1.2. Определение целей и задач проекта 18.1.3. Выбор стека технологий 18.2. Проектирование архитектуры 18.2.1. Проектирование схемы данных 18.2.2. Разработка онтологии предметной области 18.2.3. Планирование пайплайна обработки данных 18.3. Подготовка инфраструктуры 18.3.1. Настройка среды разработки 18.3.2. Инициализация репозитория проекта 18.3.3. Создание базовой конфигурации

Лабораторная работа 14

Реализация ядра системы

19.1. Разработка модуля обработки данных 19.1.1. Реализация загрузчиков документов 19.1.2. Создание пайплайна препроцессинга 19.1.3. Реализация чанкинга документов 19.2. Построение поискового движка 19.2.1. Настройка векторной базы данных 19.2.2. Реализация семантического поиска 19.2.3. Добавление гибридного поиска 19.3. Интеграция генеративного компонента 19.3.1. Настройка языковой модели 19.3.2. Разработка системы промптов 19.3.3. Реализация механизма цитирования

Лабораторная работа 15

Создание пользовательских интерфейсов

20.1. Разработка бэкенд API 20.1.1. Создание RESTful API с FastAPI 20.1.2. Реализация эндпоинтов поиска 20.1.3. Добавление аутентификации и авторизации 20.2. Создание фронтенд интерфейса 20.2.1. Разработка поискового интерфейса 20.2.2. Реализация страницы результатов 20.2.3. Создание админ-

Лабораторная работа 16

Оценка и оптимизация качества системы

Лабораторная работа 17

Улучшение функциональности системы

Лабораторная работа 18

Подготовка к production и финальная презентация

Приложения к сквозному проекту

Приложение A: Документация проекта (по желанию)

Приложение B: Метрики и результаты

Лабораторная работа №1-2, Часть 1: Подготовка рабочего окружения для MLOps

Цель работы: Освоить базовые принципы установки и настройки современного рабочего окружения для Data Science и MLOps, основанного на дистрибутиве Anaconda (conda) и технологии Docker. Получить практические навыки управления виртуальными окружениями Python, работы с менеджером пакетов conda и выполнения основных операций с Docker-контейнерами.

Стек технологий:

- **Операционная система:** Ubuntu 24.04 LTS
- **Менеджер пакетов и окружений:** Conda (в составе дистрибутива Anaconda или Miniconda)
- **Технология контейнеризации:** Docker
- **Платформа для интерактивных вычислений:** JupyterLab

Теоретическая часть (краткое содержание)

1. Введение в Conda и виртуальные окружения Современная разработка на Python, особенно в области анализа данных и машинного обучения, требует управления множеством зависимостей (библиотек) и их версий. Виртуальное окружение — это изолированное пространство, позволяющее устанавливать конкретные версии пакетов, не влияя на глобальную установку Python или другие проекты. **Conda** — это кроссплатформенный менеджер пакетов и управления окружениями с открытым исходным кодом. В отличие от стандартного `venv`, conda может управлять не только Python-пакетами, но и бинарными зависимостями (например, библиотеками C/C++), что критически важно для работы таких пакетов, как NumPy, SciPy или TensorFlow. **Anaconda** — это дистрибутив Python и R, который включает в себя conda, множество предустановленных научных пакетов (data science stack) и графические утилиты. **Miniconda** — его минималистичная версия, содержащая только conda, Python и небольшой набор базовых пакетов.

2. Введение в Docker Docker — это платформа для разработки, доставки и запуска приложений в контейнерах. **Контейнер** — это стандартизированная единица программного обеспечения, которая инкапсулирует код и все его зависимости, обеспечивая быстрое и надежное выполнение приложения в любой среде (ноутбук, сервер, облако). Ключевые преимущества:

- **Изоляция:** Приложение в контейнере не зависит от окружения хостовой системы.
- **Переносимость:** Образ, собранный на одной машине, гарантированно запустится на другой.
- **Повторяемость:** Исключается проблема "но на моей машине это работало".
- **Масштабируемость:** Легко запустить несколько экземпляров приложения.

Базовые концепции:

- **Docker Image:** Шаблон только для чтения, используемый для создания контейнеров. Собирается из Dockerfile.
- **Docker Container:** Запущенный экземпляр образа.
- **Docker Hub:** Публичный реестр для хранения и обмена образами.

Задание на практическую реализацию

Этап 1: Установка и настройка Miniconda

1. **Загрузка и установка Miniconda:**
   - Откройте терминал (`Ctrl+Alt+T`).
   - Загрузите последнюю версию установщика Miniconda для Linux (x86_64) с официального сайта с помощью `wget`:

     ```
     wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
     ```

   - Запустите установщик (ответьте `yes` на запрос лицензии и запуска `conda init`):

     ```
     bash Miniconda3-latest-Linux-x86_64.sh
     ```

   - Закройте и снова откройте терминал, чтобы активировать изменения. В начале строки приглашения терминала должно появиться `(base)`, что указывает на активацию базового окружения conda.
2. **Создание и активация виртуального окружения:**
   - Создайте новое виртуальное окружение с именем `mlops-lab` и Python версии 3.10:

     ```
     conda create -n mlops-lab python=3.10
     ```

   - Активируйте созданное окружение:

     ```
     conda activate mlops-lab
     ```

   - Убедитесь, что приглашение терминала изменилось с `(base)` на `(mlops-lab)`.
3. **Установка пакетов в созданное окружение:**
   - Установите основные пакеты для Data Science и последующей работы с MLflow, используя менеджер пакетов `pip`:

     ```
     pip install numpy pandas scikit-learn matplotlib jupyterlab
     ```

   - Проверьте установку, например, попробовав импортировать `pandas`:

```
python -c "import pandas as pd; print(pd.__version__)"
```

Этап 2: Установка Docker и базовые операции

1. **Установка Docker Engine:**
   - Обновите индексы пакетов и установите необходимые зависимости:

     ```
     sudo apt-get update
     sudo apt-get install ca-certificates curl
     ```

   - Добавьте официальный GPG-ключ Docker:

     ```
     sudo install -m 0755 -d /etc/apt/keyrings
     sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
     /etc/apt/keyrings/docker.asc
     sudo chmod a+r /etc/apt/keyrings/docker.asc
     ```

   - Настройте репозиторий:

     ```
     echo \
       "deb [arch=$(dpkg --print-architecture) signed-
     by=/etc/apt/keyrings/docker.asc]
     https://download.docker.com/linux/ubuntu \
       $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
       sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
     ```

   - Установите Docker Engine:

     ```
     sudo apt-get update
     sudo apt-get install docker-ce docker-ce-cli containerd.io docker-
     buildx-plugin docker-compose-plugin
     ```

   - Проверьте установку, запустив тестовый образ:

     ```
     sudo docker run hello-world
     ```

     Эта команда загрузит тестовый образ и запустит его в контейнере, который выведет сообщение об успешной установке.

2. **Настройка прав для не-root пользователя (ОБЯЗАТЕЛЬНО):**

- По умолчанию для управления Docker требуется права `sudo`. Чтобы запускать команды `docker` без `sudo`, добавьте своего пользователя в группу `docker`:

```
sudo usermod -aG docker $USER
```

- **Закройте терминал и войдите в систему заново**, чтобы изменения вступили в силу.

3. **Базовые команды Docker:**

- **Запуск контейнера:** Запустите контейнер из образа `ubuntu:24.04` в интерактивном режиме (`-it`):

```
docker run -it ubuntu:24.04 /bin/bash
```

Вы окажетесь внутри контейнера. Попробуйте выполнить команды `ls /`, `cat /etc/os-release`. Выйдите из контейнера командой `exit`.

- **Просмотр запущенных контейнеров:** Команда `docker ps` покажет работающие контейнеры. Чтобы увидеть все контейнеры (включая остановленные), используйте `docker ps -a`.
- **Просмотр образов:** Команда `docker images` отобразит список образов, скачанных на ваш компьютер.
- **Удаление контейнера:** Найдите ID остановленного контейнера (`docker ps -a`) и удалите его:

```
docker rm <container_id>
```

4. **Запуск JupyterLab в Docker-контейнере:**

- Запустите предсобранный образ JupyterLab, предоставляемый проектом Jupyter Docker Stacks. Эта команда делает порт 8888 контейнера доступным на локальной машине и монтирует текущую рабочую директорию (`$PWD`) в директорию `/home/jovyan/work` внутри контейнера:

```
docker run -p 8888:8888 -v $PWD:/home/jovyan/work jupyter/datascience-notebook:latest
```

- В логах запуска найдите URL-адрес вида `http://127.0.0.1:8888/lab?token=...`. Скопируйте его и откройте в браузере.
- Вы должны увидеть интерфейс JupyterLab. Проверьте, что в левой панели видна папка `work`, содержащая файлы из вашей текущей директории.
- Создайте новый Python-ноутбук (`File` -> `New` -> `Notebook`), выбрав ядро Python 3.
- В ячейке ноутбука выполните код для проверки окружения:

```
import numpy as np
import pandas as pd
print("NumPy version:", np.__version__)
print("Pandas version:", pd.__version__)
```

- Остановите контейнер, нажав `Ctrl+C` в терминале, где он запущен, или найдя его ID (`docker ps`) и выполнив `docker stop <container_id>`.

Критерии оценки для Части 1:

- **Удовлетворительно:** Успешно выполнены Этап 1 (установка Conda, создание окружения) и первые два пункта Этапа 2 (установка Docker, команда `hello-world`). Предоставлены соответствующие скриншоты.
- **Хорошо:** Дополнительно успешно выполнен пункт по запуску JupyterLab в контейнере и проверке работы ноутбука. Предоставлены скриншоты и даны ответы на контрольные вопросы.
- **Отлично:** Все задания выполнены в полном объеме. В ответах на вопросы продемонстрировано понимание принципов работы инструментов.

Рекомендуемая литература (источники)

1. **Официальная документация Conda:** https://docs.conda.io/
2. **Официальная документация Docker:** https://docs.docker.com/
3. **Jupyter Docker Stacks:** https://jupyter-docker-stacks.readthedocs.io/
4. **Ubuntu 24.04 Documentation:** https://help.ubuntu.com/
5. **Python Packaging User Guide:** https://packaging.python.org/
6. **Виртуальные окружения в Python:** https://docs.python.org/3/tutorial/venv.html
7. **Docker Get Started Tutorial:** https://docs.docker.com/get-started/
8. **Conda User Guide:** https://docs.conda.io/projects/conda/en/latest/user-guide/
9. **Статья "Зачем нужен Docker":** https://habr.com/ru/company/ruvds/blog/438796/
10. **Книга "Python for Data Analysis" by Wes McKinney:** (Главы о настройке окружения)

Лабораторная работа №1-2, Часть 2: Основы трекинга экспериментов с использованием MLflow

Цель работы: Освоить базовые принципы работы с платформой MLflow для управления жизненным циклом машинного обучения (MLOps). Получить практические навыки логирования параметров, метрик и артефактов вычислительного эксперимента, а также организации их хранения, визуализации и сравнения.

Стек технологий:

- **Операционная система:** Ubuntu 24.04 LTS
- **Менеджер пакетов и окружений:** Conda (окружение `mlops-lab` из Части 1)
- **Библиотеки:** `mlflow`, `scikit-learn`, `pandas`, `matplotlib`
- **Платформа для трекинга:** MLflow Tracking Server (standalone)

Теоретическая часть (краткое содержание)

1. Введение в MLOps и MLflow MLOps (Machine Learning Operations) — это совокупность практик, направленных на автоматизацию и надежность жизненного цикла машинного обучения (развертывание, мониторинг, управление данными). Ключевая проблема, которую решает MLOps — обеспечение воспроизводимости, отслеживаемости и управляемости ML-экспериментов. **MLflow** — это open-source платформа для управления end-to-end жизненным циклом машинного обучения. Она включает в себя четыре основных компонента:

- **MLflow Tracking:** API и UI для логирования параметров, метрик, кода и артефактов (графики, модели) в процессе выполнения ML-кода.
- **MLflow Projects:** Стандартный формат упаковки ML-кода для обеспечения воспроизводимости на любой платформе.
- **MLflow Models:** Стандартный формат упаковки ML-моделей для упрощения их развертывания с помощью различных инструментов.
- **MLflow Model Registry:** Централизованное хранилище моделей для управления их жизненным циклом (staging, production, archiving).

В данной работе focuses на компоненте **Tracking**.

2. Ключевые концепции MLflow Tracking

- **Эксперимент (Experiment):** Контейнер для группы запусков (например, "Оптимизация гиперпараметров для модели X").
- **Запуск (Run):** Одно выполнение кода, которое логируется в MLflow. Каждый запуск фиксирует:
  - **Параметры (Parameters):** Входные переменные модели (например, `max_depth`, `learning_rate`).
  - **Метрики (Metrics):** Количественные показатели качества модели (например, `accuracy`, `rmse`). Метрики могут обновляться по ходу выполнения (эпохам, итерациям).
  - **Артефакты (Artifacts):** Файлы любого типа, связанные с запуском (например, графики, обученная модель, файл с предсказаниями).

    - **Теги (Tags):** Произвольные ключ-значения для пометки запусков.
- **Backend Store:** Хранилище (файловая система или база данных), где сохраняются метаданные запусков (параметры, метрики).
- **Artifact Store:** Хранилище (например, локальная папка, S3) для артефактов.

Задание на практическую реализацию

Этап 1: Установка MLflow и запуск Tracking Server

1. **Активация окружения и установка пакетов:**
    - Активируйте созданное ранее окружение `mlops-lab`:

```
conda activate mlops-lab
```

    - Установите библиотеку `mlflow`:

```
pip install mlflow
```

2. **Запуск MLflow Tracking Server:**
    - В терминале выполните команду для запуска сервера. Флаги `--backend-store-uri` и `--default-artifact-root` указывают, где хранить метаданные и артефакты соответственно (в данном случае — в локальной директории `./mlruns`).

```
mlflow server --backend-store-uri sqlite:///mlflow.db --default-artifact-root ./mlruns --host 0.0.0.0 --port 5000
```

    - Сервер будет запущен и доступен в браузере по адресу `http://<your_ip_address>:5000` или `http://localhost:5000`. Вы должны увидеть интерфейс MLflow UI с списком экспериментов (пока пустой).

Этап 2: Написание и запуск скрипта с трекингом эксперимента

1. **Создание Python-скрипта:**
    - Создайте файл `mlflow_basic.py`.
    - Скопируйте в него следующий код, который реализует простое обучение модели логистической регрессии на встроенном в `sklearn` наборе данных Iris.

```
import mlflow
import mlflow.sklearn
from sklearn.datasets import load_iris
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score
import matplotlib.pyplot as plt

# Установите URI для отслеживания (указывает на запущенный сервер)
mlflow.set_tracking_uri("http://localhost:5000")

# Создайте или установите активный эксперимент
experiment_name = "Iris_Classification_Baseline"
mlflow.set_experiment(experiment_name)

# Загрузка данных
iris = load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Определите параметры модели для логирования
params = {
    "solver": "lbfgs",
    "max_iter": 1000,
    "multi_class": "auto",
    "random_state": 42
}

# Начало запуска MLflow
with mlflow.start_run():
    # Логирование параметров
    mlflow.log_params(params)

    # Создание и обучение модели
    model = LogisticRegression(**params)
    model.fit(X_train, y_train)

    # Предсказание и расчет метрик
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted')
    recall = recall_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')

    # Логирование метрик
    mlflow.log_metric("accuracy", accuracy)
    mlflow.log_metric("precision", precision)
    mlflow.log_metric("recall", recall)
    mlflow.log_metric("f1_score", f1)

    # Логирование модели
    mlflow.sklearn.log_model(model, "model")

    # Создание и логирование артефакта (графика)
```

```
    fig, ax = plt.subplots()
    ax.bar(['Accuracy', 'Precision', 'Recall', 'F1'], [accuracy, precision,
recall, f1])
    ax.set_ylabel('Score')
    ax.set_title('Model Performance Metrics')
    plt.savefig("metrics_plot.png") # Сохраняем график в файл
    mlflow.log_artifact("metrics_plot.png") # Логируем файл как артефакт

    # Вывод метрик в консоль для удобства
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")

print("Run completed and logged to MLflow!")
```

2. **Запуск скрипта:**
    - Убедитесь, что MLflow Server запущен.
    - В другом окне терминала (с активированным окружением `mlops-lab`) выполните:

      ```
      python mlflow_basic.py
      ```

    - В консоли должны отобразиться рассчитанные метрики и сообщение об успешном завершении.

Этап 3: Анализ результатов в MLflow UI

1. **Откройте UI:** Перейдите по адресу `http://localhost:5000`.
2. **Найдите свой эксперимент:** В боковом меню выберите эксперимент `Iris_Classification_Baseline`. Вы увидите список запусков (в данном случае один).
3. **Изучите детали запуска:**
    - Нажмите на дату/время запуска, чтобы открыть его детальную страницу.
    - Во вкладке **Overview** просмотрите логированные параметры и метрики.
    - Перейдите во вкладку **Artifacts**. Здесь вы должны увидеть:
        - Папку `model` с сохраненной моделью в формате MLflow.
        - Файл `metrics_plot.png` с построенным графиком.
4. **Сравнение запусков (опционально, для будущих работ):**
    - Запустите скрипт еще 1-2 раза, изменив какой-либо параметр (например, `"solver": "liblinear"`).
    - В UI на странице эксперимента можно будет сравнить метрики всех запусков.

Требования к оформлению и отчету (для Части 2)

Критерии оценки для Части 2:

- **Удовлетворительно:** Успешно выполнен Этап 1 (установка MLflow, запуск Tracking Server) и Этап 2 (написание и выполнение скрипта `mlflow_basic.py` без ошибок). В UI MLflow отображается один завершенный запуск.
- **Хорошо:** Дополнительно к критерию "Удовлетворительно" в UI MLflow корректно отображаются все логированные параметры, метрики и артефакты (модель и график). Продемонстрирована работа UI.
- **Отлично:** Все задания выполнены в полном объеме. Проведен дополнительный эксперимент: скрипт был модифицирован для логирования другой метрики (например, `log_loss`) или параметра (например, `test_size`), произведено 2 и более запусков, и в UI продемонстрировано их сравнение.

Рекомендуемая литература (источники)

1. **Официальная документация MLflow:** https://mlflow.org/docs/latest/index.html
2. **Официальный туториал MLflow Tracking:** https://mlflow.org/docs/latest/tracking.html
3. **Scikit-learn User Guide:** https://scikit-learn.org/stable/user_guide.html
4. **Статья "Getting Started with MLflow":** https://towardsdatascience.com/getting-started-with-mlflow-52eff9c57d0d
5. **Книга "Machine Learning Engineering" by Andriy Burkov:** (Главы, касающиеся MLOps и воспроизводимости экспериментов)
6. **MLflow: A Tool for Managing the Machine Learning Lifecycle:** https://databricks.com/blog/2018/06/05/introducing-mlflow-an-open-source-machine-learning-platform.html
7. **Документация по sklearn.metrics:** https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics
8. **Matplotlib Documentation:** https://matplotlib.org/stable/contents.html

Лабораторная работа №3-4, Часть 1: Знакомство с платформой Hugging Face Hub

Цель работы: Освоить базовые принципы работы с платформой Hugging Face Hub - центральным репозиторием моделей, датасетов и приложений машинного обучения. Получить практические навыки поиска, оценки и загрузки моделей и датасетов для задачи текстовой классификации.

Стек технологий:

- **Операционная система:** Ubuntu 24.04 LTS
- **Менеджер пакетов и окружений:** Conda (окружение `mlops-lab`)
- **Библиотеки:** `huggingface_hub`, `datasets`, `transformers`, `pandas`, `numpy`
- **Платформа:** Hugging Face Hub

Теоретическая часть (краткое содержание)

1. Введение в Hugging Face и экосистему Transformers Hugging Face — это компания и сообщество, создавшее самую популярную в мире open-source платформу для машинного обучения. Ключевые продукты:

- **Transformers:** Библиотека, предоставляющая тысячи предобученных моделей для NLP, компьютерного зрения, аудио и других задач.
- **Hugging Face Hub:** Централизованный репозиторий для обмена моделями, датасетами и демо-приложениями (Spaces).
- **Datasets:** Библиотека для простой загрузки и обработки датасетов.

Hugging Face Hub функционирует как "GitHub для ML", где исследователи и инженеры могут:

- **Обнаруживать** предобученные модели и датасеты
- **Совместно работать** над ML-проектами
- **Делиться** своими разработками с сообществом

2. Ключевые концепции платформы

- **Модели (Models):** Предобученные веса архитектур нейронных сетей (BERT, GPT, ResNet и др.) для различных задач.
- **Датасеты (Datasets):** Коллекции данных для обучения и оценки моделей. Могут быть официальными (от создателей) или community-driven.
- **Spaces:** Интерактивные веб-демонстрации моделей с графическим интерфейсом.
- **Tasks:** Стандартизированные типы ML-задач (текстовая классификация, суммирование, перевод и т.д.).

3. Задача текстовой классификации Текстовая классификация — одна из фундаментальных задач NLP, включающая:

- **Классификация тональности** (sentiment analysis)
- **Классификация тем** (topic classification)

- **Определение спама**
- **Категоризация текстов**

Задание на практическую реализацию

Этап 1: Установка необходимых библиотек

1. **Активация окружения и установка пакетов:**

```
conda activate mlops-lab
pip install huggingface_hub datasets transformers pandas numpy
```

Этап 2: Работа с Hugging Face Hub через веб-интерфейс

1. **Знакомство с интерфейсом:**
   - Откройте Hugging Face Hub в браузере.
   - Изучите главную страницу: разделы Models, Datasets, Spaces, Documentation.
2. **Поиск датасета для текстовой классификации:**
   - Перейдите в раздел **Datasets**.
   - В поиске введите "sentiment analysis" или "text classification".
   - Найдите популярные датасеты:
     - **IMDb** - классификация тональности отзывов на фильмы
     - **AG News** - классификация новостей по темам
     - **Emotion** - классификация эмоций в тексте
   - Выберите датасет **emotion** (отметьте количество примеров, лицензию, язык).
3. **Поиск модели для текстовой классификации:**
   - Перейдите в раздел **Models**.
   - В фильтрах выберите:
     - **Task:** Text Classification
     - **Library:** Transformers
     - **Dataset:** emotion (опционально)
   - Изучите доступные модели, обращая внимание на:
     - Количество загрузок
     - Размер модели
     - Язык
     - Метрики качества (если указаны)
   - Выберите модель **bert-base-uncased** или **distilbert-base-uncased** (более легкая версия).

Этап 3: Программная работа с Hugging Face Hub

1. **Создание Python-скрипта для исследования:**

```
touch hf_hub_exploration.py
```

2. **Написание кода для загрузки датасета:**

```
from datasets import load_dataset
from huggingface_hub import list_models, list_datasets
import pandas as pd

# Исследование доступных датасетов
print("Доступные датасеты для текстовой классификации:")
datasets = list_datasets(filter="task_categories:text-classification")
for dataset in datasets:
    print(f"- {dataset.id}")

# Загрузка датасета emotion
print("\nЗагрузка датасета emotion...")
dataset = load_dataset("emotion")

# Исследование структуры датасета
print(f"\nСтруктура датасета: {dataset}")
print(f"\nПримеры из train split:")
train_df = pd.DataFrame(dataset['train'][:5])
print(train_df)

# Анализ распределения классов
print("\nРаспределение классов в тренировочных данных:")
label_counts = pd.DataFrame(dataset['train']['label']).value_counts()
print(label_counts)
```

3. **Написание кода для исследования моделей:**

```
# Исследование доступных моделей
print("\n\nДоступные модели для текстовой классификации:")
models = list_models(
    filter="task:text-classification",
    sort="downloads",
    direction=-1,
    limit=5
)

for model in models:
    print(f"\nМодель: {model.id}")
    print(f"Загрузок: {model.downloads}")
    print(f"Тэги: {model.tags}")
    if model.pipeline_tag:
        print(f"Тип задачи: {model.pipeline_tag}")
```

4. **Загрузка выбранной модели:**

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification

# Загрузка токенизатора и модели
model_name = "distilbert-base-uncased"
print(f"\nЗагрузка модели {model_name}...")

tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(
    model_name,
    num_labels=6  # Количество классов в датасете emotion
)

print("Модель и токенизатор успешно загружены!")
print(f"Размер словаря: {tokenizer.vocab_size}")
print(f"Архитектура модели: {model.__class__.__name__}")
```

5. **Тестирование работы токенизатора:**

```
# Тестирование токенизатора
test_text = "I am feeling very happy today!"
print(f"\nТекст для теста: {test_text}")

tokens = tokenizer(test_text, return_tensors="pt")
print(f"Токены: {tokens}")
print(f"Декодированные токены: {tokenizer.decode(tokens['input_ids'][0])}")
```

6. **Запуск скрипта:**

```
python hf_hub_exploration.py
```

Этап 4: Сохранение локальных копий

1. **Создание директории для проекта:**

```
mkdir text-classification-project
cd text-classification-project
```

2. **Сохранение информации о выбранных ресурсах:**

```
echo "Датасет: emotion" > resources.txt
echo "Модель: distilbert-base-uncased" >> resources.txt
```

```
echo "Количество классов: 6" >> resources.txt
```

Требования к оформлению и отчету

Критерии оценки для Части 1:

- **Удовлетворительно:** Успешно выполнены Этапы 1-2 (установка библиотек, исследование Hub через веб-интерфейс, выбор датасета и модели). Создан файл `resources.txt` с информацией о выбранных ресурсах.
- **Хорошо:** Дополнительно успешно выполнен Этап 3 (написан и запущен скрипт `hf_hub_exploration.py`, который загружает датасет и информацию о моделях). Продемонстрировано понимание структуры датасета.
- **Отлично:** Все задания выполнены в полном объеме. Скрипт дополнен функционалом анализа датасета (статистика по длине текстов, визуализация распределения классов) и тестирования работы модели на примерах из датасета.

Рекомендуемая литература

1. **Официальная документация Hugging Face:** https://huggingface.co/docs
2. **Hugging Face Transformers Documentation:** https://huggingface.co/docs/transformers
3. **Hugging Face Datasets Documentation:** https://huggingface.co/docs/datasets
4. **Статья "Getting Started with Hugging Face":** https://towardsdatascience.com/getting-started-with-hugging-face-transformers-
5. **Книга "Natural Language Processing with Transformers":** Lewis et al. (Главы 1-2)

Лабораторная работа №3-4, Часть 2: Тонкая настройка модели для текстовой классификации

Цель работы: Освоить практические навыки тонкой настройки (fine-tuning) предобученных моделей для задачи текстовой классификации с использованием библиотеки Transformers. Получить опыт подготовки данных, настройки обучения и оценки качества модели.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Окружение:** Conda (`mlops-lab`)
- **Библиотеки:** `transformers`, `datasets`, `torch`, `numpy`, `pandas`, `scikit-learn`
- **Фреймворк:** PyTorch
- **Модель:** DistilBERT (`distilbert-base-uncased`)
- **Датасет:** Emotion

Теоретическая часть

1. Тонкая настройка (Fine-tuning) Тонкая настройка — это процесс дополнительного обучения предобученной модели на специфичном для задачи наборе данных. В отличие от обучения с нуля, fine-tuning:

- Требует меньше данных
- Сходится быстрее
- Достигает лучшего качества на целевой задаче

2. Архитектура Transformer для классификации Модели на основе Transformer (BERT, DistilBERT) для классификации состоят из:

- **Энкодера:** Создает контекстуализированные эмбеддинги токенов
- **Пулинга:** Извлекает представление всего текста (обычно [CLS]-токен)
- **Классификационной головки:** Линейный слой для предсказания класса

3. Процесс обучения

- **Токенизация:** Преобразование текста в токены
- **Пакетная обработка:** Группировка примеров для эффективного обучения
- **Прямое распространение:** Получение предсказаний модели
- **Вычисление потерь:** Сравнение предсказаний с истинными метками
- **Обратное распространение:** Обновление весов модели

Задание на практическую реализацию

Этап 1: Подготовка среды и данных

1. **Активация окружения:**

```
conda activate mlops-lab
cd text-classification-project
```

2. **Создание скрипта для обучения:**

```
touch fine_tuning.py
```

3. **Инициализация и загрузка данных:**

```
from datasets import load_dataset
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    TrainingArguments,
    Trainer,
    DataCollatorWithPadding
)
import numpy as np
from sklearn.metrics import accuracy_score, f1_score
import torch

# Загрузка датасета
dataset = load_dataset("emotion")

# Загрузка токенизатора
model_name = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

Этап 2: Предобработка данных

1. **Токенизация текста:**

```
def tokenize_function(examples):
    return tokenizer(
        examples["text"],
        truncation=True,
        padding=True,
        max_length=128
    )

# Применение токенизации ко всему датасету
tokenized_datasets = dataset.map(tokenize_function, batched=True)

# Форматирование данных для PyTorch
tokenized_datasets = tokenized_datasets.rename_column("label", "labels")
```

```
tokenized_datasets.set_format("torch", columns=["input_ids",
"attention_mask", "labels"])
```

2. **Создание DataCollator:**

```
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Этап 3: Настройка модели и обучения

1. **Загрузка модели:**

```
# Определение количества классов
num_labels = len(set(dataset["train"]["label"]))

# Загрузка модели с правильным количеством классов
model = AutoModelForSequenceClassification.from_pretrained(
    model_name,
    num_labels=num_labels,
    id2label={0: 'sadness', 1: 'joy', 2: 'love', 3: 'anger', 4: 'fear', 5:
'surprise'},
    label2id={'sadness': 0, 'joy': 1, 'love': 2, 'anger': 3, 'fear': 4,
'surprise': 5}
)
```

2. **Определение метрик:**

```
def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    predictions = np.argmax(predictions, axis=1)

    acc = accuracy_score(labels, predictions)
    f1 = f1_score(labels, predictions, average="weighted")

    return {"accuracy": acc, "f1_score": f1}
```

3. **Настройка гиперпараметров:**

```
training_args = TrainingArguments(
    output_dir="./results",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    evaluation_strategy="epoch",
```

```
    save_strategy="epoch",
    load_best_model_at_end=True,
    metric_for_best_model="f1_score",
    logging_dir="./logs",
    logging_steps=100,
    report_to="none"
)
```

Этап 4: Обучение модели

1. **Создание Trainer:**

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)
```

2. **Запуск обучения:**

```
print("Начало обучения...")
train_result = trainer.train()

# Сохранение модели
trainer.save_model("./emotion-classifier")
tokenizer.save_pretrained("./emotion-classifier")

print("Обучение завершено!")
print(f"Результаты обучения: {train_result.metrics}")
```

Этап 5: Оценка модели

1. **Оценка на тестовых данных:**

```
# Оценка на тестовом наборе
test_results = trainer.evaluate(tokenized_datasets["test"])
print(f"Результаты на тестовых данных: {test_results}")

# Сохранение результатов
with open("test_results.txt", "w") as f:
    f.write(f"Accuracy: {test_results['eval_accuracy']:.4f}\n")
    f.write(f"F1 Score: {test_results['eval_f1_score']:.4f}\n")
```

2. **Тестирование на примерах:**

```
# Функция для предсказания
def predict_emotion(text):
    inputs = tokenizer(text, return_tensors="pt", truncation=True,
padding=True)
    with torch.no_grad():
        outputs = model(**inputs)
    predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
    predicted_class = torch.argmax(predictions, dim=1).item()
    return model.config.id2label[predicted_class], predictions[0]
[predicted_class].item()

# Тестовые примеры
test_texts = [
    "I am feeling absolutely wonderful today!",
    "This is making me so angry and frustrated",
    "I'm scared about what might happen tomorrow"
]

print("\nТестирование модели:")
for text in test_texts:
    emotion, confidence = predict_emotion(text)
    print(f"Текст: '{text}'")
    print(f"Предсказание: {emotion} (уверенность: {confidence:.3f})")
    print()
```

3. **Запуск скрипта:**

```
python fine_tuning.py
```

Требования к оформлению и отчету

Критерии оценки для Части 2:

- **Удовлетворительно:** Успешно выполнены Этапы 1-3 (подготовка данных, настройка модели). Скрипт запускается без ошибок, начинается процесс обучения.
- **Хорошо:** Дополнительно успешно выполнен Этап 4 (модель прошла полное обучение, сохранена в директорию `emotion-classifier`). Получены метрики на валидационном наборе.
- **Отлично:** Все задания выполнены в полном объеме. Получены метрики на тестовом наборе (файл `test_results.txt`), реализована функция предсказания и протестирована на примерах. Проанализировано качество модели.

Рекомендуемая литература

1. **Hugging Face Transformers Documentation:** https://huggingface.co/docs/transformers/training
2. **Fine-tuning Tutorial:** https://huggingface.co/docs/transformers/training
3. **PyTorch Documentation:** https://pytorch.org/docs/stable/index.html
4. **Research Paper: DistilBERT:** https://arxiv.org/abs/1910.01108
5. **Practical NLP Book:** https://github.com/practical-nlp/practical-nlp

Лабораторная работа №3-4, Часть 3: Интеграция с MLflow для трекинга экспериментов

Цель работы: Освоить интеграцию процесса тонкой настройки моделей с платформой MLflow для комплексного трекинга экспериментов. Научиться автоматически логировать гиперпараметры, метрики, артефакты и модели в ходе обучения.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Окружение:** Conda (`mlops-lab`)
- **Библиотеки:** `mlflow`, `transformers`, `datasets`, `torch`
- **Платформа:** MLflow Tracking Server
- **Интеграция:** MLflow + Hugging Face Transformers

Теоретическая часть

1. Интеграция MLflow с Transformers MLflow предоставляет нативные интеграции с популярными ML-фреймворками, включая Hugging Face Transformers. Ключевые возможности:

- **Автоматическое логирование:** Автологирование параметров, метрик и артефактов
- **Модельный регистр:** Версионирование и управление моделями
- **Воспроизводимость:** Фиксация всех компонентов эксперимента

2. Компоненты трекинга для NLP

- **Параметры:** learning rate, batch size, архитектура модели
- **Метрики:** accuracy, F1-score, perplexity, loss
- **Артефакты:** модель, токенизатор, графики обучения
- **Тэги:** задача, датасет, версия модели

3. Стратегии логирования

- **Ручное логирование:** Полный контроль над процессом
- **Автологирование:** Автоматическая фиксация метрик
- **Колбэки:** Интеграция через системные хуки

Задание на практическую реализацию

Этап 1: Подготовка среды и конфигурация

1. **Активация окружения и проверка зависимостей:**

```
conda activate mlops-lab
cd text-classification-project
pip install mlflow
```

2. **Запуск MLflow Tracking Server:**

```
mlflow server --backend-store-uri sqlite:///mlflow.db --default-artifact-root ./mlruns --host 0.0.0.0 --port 5000
```

3. **Создание скрипта для интегрированного обучения:**

```
touch mlflow_integration.py
```

Этап 2: Модификация скрипта обучения с интеграцией MLflow

```
import mlflow
import mlflow.transformers
from datasets import load_dataset
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    TrainingArguments,
    Trainer,
    DataCollatorWithPadding
)
import numpy as np
from sklearn.metrics import accuracy_score, f1_score
import torch
import os

# Настройка MLflow
mlflow.set_tracking_uri("http://localhost:5000")
mlflow.set_experiment("Emotion-Classification-FineTuning")

def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    predictions = np.argmax(predictions, axis=1)

    acc = accuracy_score(labels, predictions)
    f1 = f1_score(labels, predictions, average="weighted")

    return {"accuracy": acc, "f1_score": f1}

def tokenize_function(examples):
    tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
    return tokenizer(
        examples["text"],
        truncation=True,
        padding=True,
        max_length=128
```

```
    )

# Начало эксперимента MLflow
with mlflow.start_run():
    # Загрузка и подготовка данных
    dataset = load_dataset("emotion")
    tokenized_datasets = dataset.map(tokenize_function, batched=True)
    tokenized_datasets = tokenized_datasets.rename_column("label", "labels")
    tokenized_datasets.set_format("torch", columns=["input_ids", "attention_mask",
"labels"])

    # Параметры модели и обучения
    model_params = {
        "model_name": "distilbert-base-uncased",
        "num_labels": 6,
        "learning_rate": 2e-5,
        "batch_size": 16,
        "num_epochs": 3,
        "weight_decay": 0.01
    }

    # Логирование параметров
    mlflow.log_params(model_params)

    # Загрузка модели
    model = AutoModelForSequenceClassification.from_pretrained(
        model_params["model_name"],
        num_labels=model_params["num_labels"],
        id2label={0: 'sadness', 1: 'joy', 2: 'love', 3: 'anger', 4: 'fear', 5:
'surprise'},
        label2id={'sadness': 0, 'joy': 1, 'love': 2, 'anger': 3, 'fear': 4,
'surprise': 5}
    )

    # Настройка обучения
    training_args = TrainingArguments(
        output_dir="./results",
        learning_rate=model_params["learning_rate"],
        per_device_train_batch_size=model_params["batch_size"],
        per_device_eval_batch_size=model_params["batch_size"],
        num_train_epochs=model_params["num_epochs"],
        weight_decay=model_params["weight_decay"],
        evaluation_strategy="epoch",
        save_strategy="epoch",
        load_best_model_at_end=True,
        metric_for_best_model="f1_score",
        logging_dir="./logs",
        logging_steps=100,
        report_to="none"
    )

    data_collator =
DataCollatorWithPadding(tokenizer=AutoTokenizer.from_pretrained(model_params["mode
l_name"]))
```

```
    # Создание тренера
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=tokenized_datasets["train"],
        eval_dataset=tokenized_datasets["validation"],
        tokenizer=AutoTokenizer.from_pretrained(model_params["model_name"]),
        data_collator=data_collator,
        compute_metrics=compute_metrics,
    )

    # Обучение с логированием метрик
    print("Начало обучения с трекингом в MLflow...")
    train_result = trainer.train()

    # Логирование метрик обучения
    mlflow.log_metrics({
        "train_loss": train_result.metrics["train_loss"],
        "eval_loss": train_result.metrics["eval_loss"],
        "eval_accuracy": train_result.metrics["eval_accuracy"],
        "eval_f1_score": train_result.metrics["eval_f1_score"]
    })

    # Оценка на тестовых данных
    test_results = trainer.evaluate(tokenized_datasets["test"])
    mlflow.log_metrics({
        "test_accuracy": test_results["eval_accuracy"],
        "test_f1_score": test_results["eval_f1_score"]
    })

    # Сохранение и логирование модели
    model_path = "./emotion-classifier-mlflow"
    trainer.save_model(model_path)

    # Логирование модели в MLflow
    mlflow.transformers.log_model(
        transformers_model={
            "model": model,
            "tokenizer": AutoTokenizer.from_pretrained(model_params["model_name"])
        },
        artifact_path="emotion-classifier",
        registered_model_name="distilbert-emotion-classifier"
    )

    # Логирование дополнительных артефактов
    with open("training_summary.txt", "w") as f:
        f.write(f"Training completed successfully!\n")
        f.write(f"Final training loss:
{train_result.metrics['train_loss']:.4f}\n")
        f.write(f"Validation accuracy:
{train_result.metrics['eval_accuracy']:.4f}\n")
        f.write(f"Test accuracy: {test_results['eval_accuracy']:.4f}\n")
```

```
    mlflow.log_artifact("training_summary.txt")

    print("Эксперимент успешно завершен и записан в MLflow!")
```

Этап 3: Запуск и мониторинг эксперимента

1. **Запуск скрипта:**

```
python mlflow_integration.py
```

2. **Мониторинг в MLflow UI:**
   - Откройте `http://localhost:5000` в браузере
   - Найдите эксперимент "Emotion-Classification-FineTuning"
   - Изучите записанные параметры и метрики
   - Проверьте залогированную модель в разделе Artifacts

Этап 4: Дополнительные эксперименты

1. **Создание скрипта для сравнения гиперпараметров:**

```
touch hyperparameter_tuning.py
```

2. **Код для сравнения разных конфигураций:**

```
import mlflow
from mlflow_integration import train_model

# Эксперимент с разными learning rates
learning_rates = [1e-5, 2e-5, 5e-5]

for lr in learning_rates:
    with mlflow.start_run(nested=True):
        mlflow.log_param("learning_rate", lr)
        results = train_model(learning_rate=lr)
        mlflow.log_metrics(results)

print("Эксперимент по подбору learning rate завершен!")
```

Этап 5: Анализ результатов

1. **Создание скрипта для анализа:**

```
touch analyze_results.py
```

2. **Код для анализа экспериментов:**

```
import mlflow
from mlflow.tracking import MlflowClient

client = MlflowClient()

# Получение всех запусков эксперимента
experiment = client.get_experiment_by_name("Emotion-Classification-
FineTuning")
runs = client.search_runs(experiment.experiment_id)

print("Результаты экспериментов:")
for run in runs:
    print(f"Run ID: {run.info.run_id}")
    print(f"Parameters: {run.data.params}")
    print(f"Metrics: {run.data.metrics}")
    print("-" * 50)

# Нахождение лучшего запуска
best_run = min(runs, key=lambda x: x.data.metrics.get('eval_loss',
float('inf')))
print(f"Лучший запуск: {best_run.info.run_id}")
print(f"Лучшие метрики: {best_run.data.metrics}")
```

Требования к оформлению и отчету

Критерии оценки для Части 3:

- **Удовлетворительно:** Успешно выполнены Этапы 1-2 (настройка MLflow, модификация скрипта). Эксперимент запускается и базовые параметры записываются в MLflow.
- **Хорошо:** Дополнительно успешно выполнен Этап 3 (полное обучение с логированием всех метрик, модель зарегистрирована в MLflow). В UI отображаются все артефакты.
- **Отлично:** Все задания выполнены в полном объеме. Проведены дополнительные эксперименты (Этап 4) и выполнен анализ результатов (Этап 5). Сравнены разные конфигурации гиперпараметров.

Рекомендуемая литература

1. **MLflow Documentation:** https://mlflow.org/docs/latest/index.html
2. **MLflow Transformers Integration:** https://mlflow.org/docs/latest/python_api/mlflow.transformers.html
3. **Hugging Face Transformers Training:** https://huggingface.co/docs/transformers/training
4. **MLflow Tracking API:** https://mlflow.org/docs/latest/tracking.html
5. **Experiment Tracking Best Practices:** https://mlflow.org/docs/latest/tracking.html#organizing-runs-in-experiments

Лабораторная работа №5-6, Часть 1: Работа с векторными базами данных (ChromaDB)

Цель работы: Освоить принципы работы с векторными базами данных на примере ChromaDB. Получить практические навыки создания коллекций, генерации эмбеддингов, выполнения семантического поиска и интеграции с ML-моделями.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Окружение:** Conda (`mlops-lab`)
- **Библиотеки:** `chromadb`, `sentence-transformers`, `pandas`, `numpy`
- **Модели:** `all-MiniLM-L6-v2` (SentenceTransformer)
- **База данных:** ChromaDB (in-memory mode)

Теоретическая часть

1. Векторные базы данных Векторные БД предназначены для хранения и поиска векторных представлений данных (эмбеддингов). Ключевые особенности:

- **Хранение векторов:** Оптимизированы для работы с высокоразмерными векторами
- **Семантический поиск:** Поиск по смыслу, а не точному совпадению
- **Ближайшие соседи:** Эффективный поиск похожих объектов

2. ChromaDB Open-source векторная БД с простым API:

- **Локальный режим:** Работа в памяти или с персистентным хранилищем
- **Клиент-сервер:** Разделение на клиент и сервер
- **Python-first:** Простая интеграция с Python-экосистемой

3. Применение в NLP

- Поиск похожих документов
- Векторизация текстовых данных
- Основа для RAG-систем

Задание на практическую реализацию

Этап 1: Установка и настройка окружения

1. **Активация окружения и установка пакетов:**

```
conda activate mlops-lab
pip install chromadb sentence-transformers pandas numpy
```

2. **Создание рабочей директории:**

```
mkdir vector-db-lab
cd vector-db-lab
```

Этап 2: Подготовка данных и модели

1. **Создание скрипта для работы с ChromaDB:**

```
touch chroma_demo.py
```

2. **Импорт библиотек и загрузка модели:**

```
import chromadb
from sentence_transformers import SentenceTransformer
import pandas as pd
import numpy as np
from typing import List, Dict

# Загрузка модели для создания эмбеддингов
model = SentenceTransformer('all-MiniLM-L6-v2')
print("Модель для эмбеддингов загружена")
```

Этап 3: Создание и настройка базы данных

1. **Инициализация ChromaDB:**

```
# Создание клиента ChromaDB (в памяти)
client = chromadb.Client()

# Создание коллекции
collection = client.create_collection(
    name="documents_collection",
    metadata={"hnsw:space": "cosine"}  # Использование косинусного
расстояния
)

print("Коллекция создана")
```

Этап 4: Подготовка и векторизация данных

1. **Создание тестового набора документов:**

```
# Примеры документов для базы знаний
documents = [
    "Машинное обучение - это область искусственного интеллекта",
    "Глубокое обучение использует нейронные сети с множеством слоев",
    "Трансформеры revolutionized обработку естественного языка",
    "BERT является популярной моделью для понимания текста",
    "GPT модели используются для генерации текста",
    "Векторные базы данных хранят embeddings для семантического поиска",
    "ChromaDB - это open-source векторная база данных",
    "Semantic search позволяет находить документы по смыслу",
    "Neural networks inspired биологическими нейронными сетями",
    "Natural Language Processing обрабатывает человеческий язык"
]

# Создание метаданных и идентификаторов
metadata = [{"category": "AI", "source": "educational"} for _ in
range(len(documents))]
ids = [f"doc_{i}" for i in range(len(documents))]

print(f"Подготовлено {len(documents)} документов")
```

2. **Генерация эмбеддингов и добавление в коллекцию:**

```
# Генерация эмбеддингов для документов
embeddings = model.encode(documents).tolist()

# Добавление документов в коллекцию
collection.add(
    documents=documents,
    embeddings=embeddings,
    metadatas=metadata,
    ids=ids
)

print("Документы добавлены в коллекцию")
```

Этап 5: Семантический поиск

1. **Функция для выполнения запросов:**

```
def semantic_search(query: str, n_results: int = 3):
    # Генерация эмбеддинга для запроса
    query_embedding = model.encode([query]).tolist()

    # Поиск похожих документов
    results = collection.query(
        query_embeddings=query_embedding,
        n_results=n_results,
```

```
        include=["documents", "distances", "metadatas"]
    )

    return results

# Тестовые запросы
test_queries = [
    "искусственный интеллект",
    "нейронные сети",
    "обработка текста",
    "базы данных"
]

print("\nРезультаты семантического поиска:")
for query in test_queries:
    print(f"\nЗапрос: '{query}'")
    results = semantic_search(query)

    for i, (doc, distance) in enumerate(zip(results['documents'][0],
results['distances'][0])):
        print(f"{i+1}. {doc} (расстояние: {distance:.4f})")
```

Этап 6: Расширенная функциональность

1. **Работа с метаданными и фильтрация:**

```
def filtered_search(query: str, filter_dict: Dict, n_results: int = 2):
    query_embedding = model.encode([query]).tolist()

    results = collection.query(
        query_embeddings=query_embedding,
        n_results=n_results,
        where=filter_dict,
        include=["documents", "distances", "metadatas"]
    )

    return results

# Поиск с фильтрацией
print("\n\nПоиск с фильтрацией по категории:")
results = filtered_search(
    "модели машинного обучения",
    {"category": "AI"},
    n_results=2
)

for i, (doc, distance) in enumerate(zip(results['documents'][0],
results['distances'][0])):
    print(f"{i+1}. {doc} (расстояние: {distance:.4f})")
```

2. **Анализ коллекции:**

```
# Получение информации о коллекции
print(f"\nИнформация о коллекции:")
print(f"Количество документов: {collection.count()}")

# Получение нескольких документов
sample_docs = collection.get(ids=["doc_0", "doc_1"])
print("\nПримеры документов:")
for doc in sample_docs['documents']:
    print(f"- {doc}")
```

Этап 7: Сохранение и загрузка базы данных

1. **Персистентное хранение:**

```
# Создание персистентного клиента
persistent_client = chromadb.PersistentClient(path="./chroma_db")

# Создание персистентной коллекции
persistent_collection = persistent_client.create_collection(
    name="persistent_docs",
    metadata={"hnsw:space": "cosine"}
)

# Добавление документов в персистентную коллекцию
persistent_collection.add(
    documents=documents[:5],  # Первые 5 документов
    embeddings=embeddings[:5],
    metadatas=metadata[:5],
    ids=ids[:5]
)

print("Персистентная коллекция создана и заполнена")

# Проверка загрузки
loaded_collection = persistent_client.get_collection("persistent_docs")
print(f"Загружено документов: {loaded_collection.count()}")
```

Этап 8: Интеграция с реальными данными

1. **Загрузка датасета для тестирования:**

```
# Создание коллекции с реальными данными
def create_news_collection():
    # Пример данных (в реальности можно загрузить из файла)
    news_data = [
```

```
        "Рынок акций вырос на 2% сегодня",
        "Новая технология в области искусственного интеллекта представлена",
        "Криптовалюты показывают волатильность на рынке",
        "Ученые сделали breakthrough в квантовых вычислениях",
        "Центральные банки обсуждают monetary policy"
    ]

    news_metadata = [
        {"category": "finance", "date": "2024-01-15"},
        {"category": "technology", "date": "2024-01-15"},
        {"category": "crypto", "date": "2024-01-14"},
        {"category": "science", "date": "2024-01-14"},
        {"category": "economics", "date": "2024-01-13"}
    ]

    news_ids = [f"news_{i}" for i in range(len(news_data))]
    news_embeddings = model.encode(news_data).tolist()

    news_collection = client.create_collection(name="news_collection")
    news_collection.add(
        documents=news_data,
        embeddings=news_embeddings,
        metadatas=news_metadata,
        ids=news_ids
    )

    return news_collection

# Тестирование новостной коллекции
news_coll = create_news_collection()
results = news_coll.query(
    query_embeddings=model.encode(["финансовые новости"]).tolist(),
    n_results=2
)

print("\nПоиск в новостной коллекции:")
for doc in results['documents'][0]:
    print(f"- {doc}")
```

2. **Запуск скрипта:**

```
python chroma_demo.py
```

Требования к оформлению и отчету

Критерии оценки для Части 1:

- **Удовлетворительно:** Успешно выполнены Этапы 1-4 (установка, создание коллекции, добавление документов). Скрипт запускается без ошибок.

- **Хорошо:** Дополнительно успешно выполнен Этап 5 (реализован семантический поиск, протестированы запросы). Получены осмысленные результаты поиска.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы расширенные функции (Этапы 6-8): фильтрация, персистентное хранение, работа с разными типами данных. Проанализирована эффективность поиска.

Рекомендуемая литература

1. **ChromaDB Documentation:** https://docs.trychroma.com/
2. **Sentence Transformers Documentation:** https://www.sbert.net/
3. **Vector Databases Overview:** https://www.pinecone.io/learn/vector-database/
4. **Semantic Search Techniques:** https://www.sbert.net/examples/applications/semantic-search/README.html
5. **Similarity Search Algorithms:** https://en.wikipedia.org/wiki/Nearest_neighbor_search

Лабораторная работа №5-6, Часть 2: Работа с графовыми базами данных (Neo4j)

Цель работы: Освоить принципы работы с графовыми базами данных на примере Neo4j. Получить практические навыки создания узлов и связей, выполнения запросов на языке Cypher и визуализации графовых структур.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Окружение:** Docker, Conda (`mlops-lab`)
- **Библиотеки:** `neo4j`, `pandas`, `matplotlib`
- **База данных:** Neo4j (Docker контейнер)
- **Язык запросов:** Cypher
- **Интерфейс:** Neo4j Browser

Теоретическая часть

1. Графовые базы данных Графовые БД предназначены для хранения и обработки данных в виде графов (узлов и связей). Ключевые особенности:

- **Узлы (Nodes):** Сущности данных (объекты)
- **Связи (Relationships):** Отношения между узлами
- **Свойства (Properties):** Атрибуты узлов и связей
- **Метки (Labels):** Категории узлов

2. Neo4j Ведущая графовая база данных с открытым исходным кодом:

- **Cypher:** Декларативный язык запросов
- **ACID-совместимость:** Поддержка транзакций
- **Визуализация:** Интерактивный просмотр графов

3. Применение в управлении знаниями

- Построение онтологий и знанийых графов
- Анализ социальных сетей
- Рекомендательные системы
- Поиск паттернов и взаимосвязей

Задание на практическую реализацию

Этап 1: Запуск Neo4j в Docker

1. **Запуск Neo4j контейнера:**

```
docker run \
    --name neo4j-graphdb \
    -p 7474:7474 \
    -p 7687:7687 \
    -d \
    -v $(pwd)/neo4j/data:/data \
    -v $(pwd)/neo4j/logs:/logs \
    -v $(pwd)/neo4j/import:/var/lib/neo4j/import \
    --env NEO4J_AUTH=neo4j/password123 \
    neo4j:latest
```

2. **Проверка работы контейнера:**

```
docker ps
docker logs neo4j-graphdb
```

Этап 2: Подключение к Neo4j Browser

1. **Открытие веб-интерфейса:**
   - Откройте браузер и перейдите по адресу: `http://localhost:7474`
   - Логин: `neo4j`
   - Пароль: `password123`
2. **Изменение пароля:**
   - При первом входе измените пароль на `graphdb2024`

Этап 3: Основы языка Cypher

1. **Создание скрипта для работы с Neo4j:**

```
touch neo4j_demo.py
```

2. **Подключение к базе данных:**

```
from neo4j import GraphDatabase
import pandas as pd

# Настройки подключения
URI = "bolt://localhost:7687"
AUTH = ("neo4j", "graphdb2024")

# Создание драйвера
driver = GraphDatabase.driver(URI, auth=AUTH)
```

```
def test_connection():
    with driver.session() as session:
        result = session.run("RETURN 'Connected to Neo4j' AS message")
        return result.single()["message"]

print(test_connection())
```

Этап 4: Создание онтологии предметной области

1. **Создание узлов и связей:**

```
def create_knowledge_graph(tx):
    # Очистка базы данных
    tx.run("MATCH (n) DETACH DELETE n")

    # Создание узлов (сущностей)
    query = """
    CREATE
    (ai:Domain {name: 'Artificial Intelligence'}),
    (ml:Technology {name: 'Machine Learning', type: 'ML'}),
    (dl:Technology {name: 'Deep Learning', type: 'DL'}),
    (nlp:Technology {name: 'NLP', type: 'NLP'}),
    (bert:Model {name: 'BERT', developer: 'Google'}),
    (gpt:Model {name: 'GPT', developer: 'OpenAI'}),
    (python:Language {name: 'Python', paradigm: 'multi-paradigm'}),
    (pytorch:Framework {name: 'PyTorch', language: 'Python'}),
    (tf:Framework {name: 'TensorFlow', language: 'Python'}),

    // Создание связей
    (ai)-[:INCLUDES]->(ml),
    (ai)-[:INCLUDES]->(nlp),
    (ml)-[:CONTAINS]->(dl),
    (nlp)-[:USES]->(bert),
    (nlp)-[:USES]->(gpt),
    (ml)-[:IMPLEMENTED_IN]->(python),
    (dl)-[:FRAMEWORK]->(pytorch),
    (dl)-[:FRAMEWORK]->(tf),
    (bert)-[:BUILT_WITH]->(tf),
    (gpt)-[:BUILT_WITH]->(pytorch),
    (python)-[:HAS_FRAMEWORK]->(pytorch),
    (python)-[:HAS_FRAMEWORK]->(tf)
    """
    tx.run(query)

with driver.session() as session:
    session.execute_write(create_knowledge_graph)
    print("База знаний создана")
```

Этап 5: Выполнение базовых запросов

1. **Поиск всех узлов:**

```
def get_all_nodes(tx):
    result = tx.run("MATCH (n) RETURN n.name AS name, labels(n) AS labels")
    return [{"name": record["name"], "labels": record["labels"]} for record
in result]

print("Все узлы в базе:")
nodes = get_all_nodes(driver.session())
for node in nodes:
    print(f"{node['name']} - {node['labels']}")
```

2. **Поиск связей:**

```
def get_relationships(tx):
    query = """
    MATCH (a)-[r]->(b)
    RETURN a.name AS source, type(r) AS relationship, b.name AS target
    """
    result = tx.run(query)
    return [{"source": record["source"], "relationship":
record["relationship"], "target": record["target"]} for record in result]

print("\nСвязи в графе:")
relationships = get_relationships(driver.session())
for rel in relationships:
    print(f"{rel['source']} --{rel['relationship']}--> {rel['target']}")
```

Этап 6: Сложные запросы и анализ

1. **Поиск путей:**

```
def find_paths(tx, start_node, end_node):
    query = """
    MATCH path = (a {name: $start_node})-[*]->(b {name: $end_node})
    RETURN [node in nodes(path) | node.name] AS path_nodes,
           [rel in relationships(path) | type(rel)] AS relationships
    """
    result = tx.run(query, start_node=start_node, end_node=end_node)
    return [{"nodes": record["path_nodes"], "relationships":
record["relationships"]} for record in result]

print("\nПути от AI до BERT:")
paths = find_paths(driver.session(), "Artificial Intelligence", "BERT")
for path in paths:
    print(f"Путь: {' -> '.join(path['nodes'])}")
```

2. **Анализ степени связности:**

```
def analyze_connectivity(tx):
    query = """
    MATCH (n)
    RETURN n.name AS node,
           size((n)--()) AS degree,
           labels(n)[0] AS type
    ORDER BY degree DESC
    """
    result = tx.run(query)
    return [{"node": record["node"], "degree": record["degree"], "type":
record["type"]} for record in result]

print("\nАнализ связности узлов:")
connectivity = analyze_connectivity(driver.session())
for node in connectivity:
    print(f"{node['node']} ({node['type']}): {node['degree']} связей")
```

Этап 7: Работа с реальными данными

1. **Импорт данных из CSV:**

```
def import_ai_researchers(tx):
    # Создание узлов исследователей
    query = """
    LOAD CSV WITH HEADERS FROM 'file:///ai_researchers.csv' AS row
    CREATE (:Researcher {
        name: row.name,
        affiliation: row.affiliation,
        field: row.field,
        h_index: toInteger(row.h_index)
    })
    """
    tx.run(query)

# Создание CSV файла для импорта
researchers_data = """name,affiliation,field,h_index
Yann LeCun,Facebook AI Research,Computer Vision,180
Andrew Ng,Stanford University,Machine Learning,150
Yoshua Bengio,MILA,Deep Learning,170
Geoffrey Hinton,University of Toronto,Neural Networks,200
Demis Hassabis,Google DeepMind,Reinforcement Learning,80
"""

with open("neo4j/import/ai_researchers.csv", "w") as f:
    f.write(researchers_data)

with driver.session() as session:
```

```
    session.execute_write(import_ai_researchers)
    print("Данные исследователей импортированы")
```

2. **Создание связей с существующей онтологией:**

```
def connect_researchers_to_domains(tx):
    query = """
    MATCH (r:Researcher), (d:Domain {name: 'Artificial Intelligence'})
    CREATE (r)-[:WORKS_IN]->(d)

    MATCH (r:Researcher {name: 'Yann LeCun'}), (dl:Technology {name: 'Deep
Learning'})
    CREATE (r)-[:CONTRIBUTED_TO]->(dl)

    MATCH (r:Researcher {name: 'Andrew Ng'}), (ml:Technology {name: 'Machine
Learning'})
    CREATE (r)-[:CONTRIBUTED_TO]->(ml)
    """
    tx.run(query)

with driver.session() as session:
    session.execute_write(connect_researchers_to_domains)
    print("Связи исследователей созданы")
```

Этап 8: Визуализация и экспорт

1. **Визуализация графа через Neo4j Browser:**
   - Откройте Neo4j Browser (`http://localhost:7474`)
   - Выполните запрос: `MATCH (n) RETURN n LIMIT 25`
   - Нажмите "View in graph" для визуализации

2. **Экспорт данных:**

```
def export_graph_data(tx):
    query = """
    MATCH (n)
    RETURN n.name AS name,
           labels(n) AS labels,
           properties(n) AS properties
    """
    result = tx.run(query)
    df = pd.DataFrame([dict(record) for record in result])
    df.to_csv("graph_export.csv", index=False)
    return df

graph_data = export_graph_data(driver.session())
print("\nЭкспортированные данные:")
print(graph_data.head())
```

3. **Закрытие соединения:**

```
driver.close()
print("Соединение с Neo4j закрыто")
```

4. **Остановка контейнера:**

```
docker stop neo4j-graphdb
docker rm neo4j-graphdb
```

Требования к оформлению и отчету

Критерии оценки для Части 2:

- **Удовлетворительно:** Успешно выполнены Этапы 1-4 (запуск Neo4j, создание базовой онтологии). Скрипт запускается без ошибок.
- **Хорошо:** Дополнительно успешно выполнен Этап 5-6 (реализованы базовые и сложные запросы, анализ связности). Получены осмысленные результаты запросов.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы расширенные функции (Этапы 7-8): импорт реальных данных, визуализация графа, экспорт результатов. Проанализирована структура графа знаний.

Рекомендуемая литература

1. **Neo4j Documentation:** https://neo4j.com/docs/
2. **Cypher Query Language:** https://neo4j.com/developer/cypher/
3. **Graph Databases Book:** https://graphdatabases.com/
4. **Neo4j Python Driver:** https://neo4j.com/docs/python-manual/current/
5. **Knowledge Graphs:** https://www.ontotext.com/knowledgehub/fundamentals/what-is-a-knowledge-graph/

Лабораторная работа №7-8, Часть 1: Знакомство с онтологиями в Protégé

Цель работы: Освоить базовые принципы работы с онтологиями и семантическими технологиями через инструмент Protégé. Получить практические навыки изучения структуры онтологий, работы с классами, свойствами и индивидами.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Инструмент:** Protégé Desktop
- **Форматы:** OWL, RDF, Turtle
- **Языки:** OWL 2, RDFS
- **Датасет:** Pizza ontology (образовательная онтология)

Теоретическая часть

1. Онтологии в компьютерных науках Онтология — формальное представление знаний в виде иерархии понятий и отношений между ними. Ключевые компоненты:

- **Классы (Concepts):** Категории объектов предметной области
- **Свойства (Properties):** Отношения между объектами
- **Индивиды (Individuals):** Конкретные экземпляры классов
- **Аксиомы (Axioms):** Правила и ограничения

2. Язык OWL (Web Ontology Language) Стандарт W3C для создания онтологий:

- **OWL DL:** Поддержка сложных логических конструкций
- **Семантика описательной логики:** Формальная основа для логического вывода
- **Инференс:** Автоматическое выведение новых знаний

3. Protégé Ведущий open-source инструмент для работы с онтологиями:

- **Визуальный редактор:** Графическое создание и редактирование онтологий
- **Поддержка推理:** Интеграция с reasoners (HermiT, Pellet)
- **Расширяемость:** Поддержка плагинов

Задание на практическую реализацию

Этап 1: Установка и настройка Protégé

1. **Установка Java (требование для Protégé):**

```
sudo apt update
sudo apt install openjdk-17-jdk
java -version
```

2. **Скачивание и установка Protégé:**

```
wget https://github.com/protegeproject/protege-
distribution/releases/download/v5.6.2/protege-5.6.2-platform.zip
unzip protege-5.6.2-platform.zip
cd protege-5.6.2
```

3. **Запуск Protégé:**

```
./run.sh
```

Этап 2: Загрузка и изучение образовательной онтологии

1. **Загрузка Pizza Ontology:**
   - В меню Protégé выберите: `File → Open from URL...`
   - Введите URL: `https://protege.stanford.edu/ontologies/pizza/pizza.owl`
   - Сохраните локальную копию: `File → Save As → pizza.owl`

2. **Изучение интерфейса Protégé:**
   - **Active Ontology:** Метаинформация об онтологии
   - **Entities:** Основные сущности (Classes, Properties, Individuals)
   - **Class Description:** Описание выбранного класса
   - **OWL Viz:** Визуализация иерархии классов

Этап 3: Анализ структуры онтологии

1. **Изучение вкладки "Classes":**

```
# Создание файла для заметок
touch ontology_analysis.txt
```

   Задание: Изучите иерархию классов и запишите наблюдения:
   - Корневые классы онтологии
   - Глубина иерархии
   - Основные категории пицц

2. **Анализ свойств объекта:**
   - Перейдите во вкладку **Object Properties**
   - Изучите основные отношения:
     - `hasTopping`

- `hasBase`
- `hasSpiciness`

3. **Изучение индивидов:**
   - Перейдите во вкладку **Individuals**
   - Найдите примеры конкретных пицц
   - Изучите их свойства и связи

Этап 4: Работа с классами и свойствами

1. **Создание нового класса:**
   - В **Classes** нажмите `Add subclass`
   - Создайте класс: `RussianPizza`
   - Добавьте аннотацию: `Comment: Traditional Russian pizza variations`
2. **Определение свойств класса:**
   - В **Class Description** выберите `RussianPizza`
   - Добавьте ограничение: `hasTopping some RedOnion`
   - Добавьте ограничение: `hasTopping some Sausage`
3. **Создание объектаных свойств:**
   - В **Object Properties** создайте свойство: `hasTraditionalTopping`
   - Установите домен: `Pizza`
   - Установите диапазон: `PizzaTopping`

Этап 5: Использование reasoner для логического вывода

1. **Выбор и запуск reasoner:**
   - В меню выберите: `Reasoner → HermiT`
   - Запустите: `Reasoner → Start reasoner`
2. **Анализ результатов:**
   - Проверьте автоматическую классификацию пицц
   - Изучите непротиворечивость онтологии
   - Найдите новые inferred классы
3. **Создание запроса:**
   - Перейдите во вкладку **DL Query**
   - Введите запрос: `Pizza and hasTopping value Mushroom`
   - Проанализируйте результаты

Этап 6: Сохранение и экспорт онтологии

1. **Сохранение в разных форматах:**

```
# Сохранение в формате Turtle
File → Save As → pizza_russian.ttl

# Экспорт в RDF/XML
File → Save As → pizza_russian.rdf
```

2. **Создание отчета о онтологии:**

```
# Скрипт для анализа онтологии
from rdflib import Graph
import pandas as pd

def analyze_ontology(file_path):
    g = Graph()
    g.parse(file_path, format="turtle")

    # Статистика онтологии
    classes = list(g.subjects(predicate="http://www.w3.org/1999/02/22-rdf-
syntax-ns#type",
                              object="http://www.w3.org/2002/07/owl#Class"))

    properties = list(g.subjects(predicate="http://www.w3.org/1999/02/22-
rdf-syntax-ns#type",

object="http://www.w3.org/2002/07/owl#ObjectProperty"))

    individuals = list(g.subjects(predicate="http://www.w3.org/1999/02/22-
rdf-syntax-ns#type",

object="http://www.w3.org/2002/07/owl#NamedIndividual"))

    print(f"Классы: {len(classes)}")
    print(f"Свойства: {len(properties)}")
    print(f"Индивиды: {len(individuals)}")

    return {
        "classes": len(classes),
        "properties": len(properties),
        "individuals": len(individuals)
    }

# Анализ оригинальной и модифицированной онтологии
stats_original = analyze_ontology("pizza.owl")
stats_modified = analyze_ontology("pizza_russian.ttl")

# Создание отчета
report = pd.DataFrame([stats_original, stats_modified],
                      index=["Original", "Modified"])
report.to_csv("ontology_report.csv")
```

Этап 7: Документирование онтологии

1. **Создание документации:**

```
# Анализ онтологии Pizza

## Основные классы
- Pizza: Базовый класс всех пицц
- PizzaTopping: Ингредиенты для пиццы
- PizzaBase: Основа пиццы

## Добавленные элементы
- Класс: RussianPizza
- Свойство: hasTraditionalTopping

## Статистика
| Метрика | Оригинал | Модифицированная |
|---------|----------|------------------|
| Классы | X | Y |
| Свойства | X | Y |
| Индивиды | X | Y |
```

Требования к оформлению и отчету

Критерии оценки для Части 1:

- **Удовлетворительно:** Успешно выполнены Этапы 1-2 (установка Protégé, загрузка онтологии). Создана локальная копия онтологии.
- **Хорошо:** Дополнительно успешно выполнен Этап 3-4 (анализ структуры, создание новых классов и свойств). Модифицированная онтология сохранена.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы Этапы 5-7: использование reasoner, экспорт в разные форматы, создание отчета статистики. Проанализированы изменения в онтологии.

Рекомендуемая литература

1. **Protégé Documentation:** https://protegeproject.github.io/protege/
2. **OWL 2 Primer:** https://www.w3.org/TR/owl2-primer/
3. **Pizza Ontology Tutorial:** https://protege.stanford.edu/ontologies/tutorials/protege-owl-tutorial.pdf
4. **Semantic Web Technologies:** https://www.w3.org/standards/semanticweb/
5. **RDFLib Documentation:** https://rdflib.readthedocs.io/

Лабораторная работа №7-8, Часть 2: Работа с SPARQL-запросами

Цель работы: Освоить язык запросов SPARQL для работы с семантическими данными. Получить практические навыки подключения к семантическому хранилищу, выполнения различных типов запросов и анализа результатов.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Хранилище:** Apache Jena Fuseki
- **Библиотеки:** `SPARQLWrapper`, `rdflib`, `pandas`
- **Язык запросов:** SPARQL 1.1
- **Форматы данных:** RDF, OWL, Turtle

Теоретическая часть

1. Язык SPARQL SPARQL (SPARQL Protocol and RDF Query Language) — стандартный язык запросов для RDF-данных. Основные типы запросов:

- **SELECT:** Возвращает таблицу результатов
- **CONSTRUCT:** Создает новый RDF-граф
- **ASK:** Возвращает boolean-ответ
- **DESCRIBE:** Возвращает RDF-описание ресурса

2. Apache Jena Fuseki Сервер SPARQL с веб-интерфейсом для работы с RDF-данными:

- **Поддержка SPARQL 1.1:** Полная реализация стандарта
- **Веб-интерфейс:** Интерактивное выполнение запросов
- **REST API:** Программный доступ к данным

3. Структура SPARQL-запроса

- **PREFIX:** Определение пространств имен
- **SELECT/CONSTRUCT:** Цель запроса
- **WHERE:** Шаблон для сопоставления
- **FILTER:** Условия фильтрации
- **OPTIONAL:** Необязательные совпадения
- **ORDER BY/LIMIT:** Сортировка и ограничения

Задание на практическую реализацию

Этап 1: Установка и запуск Apache Jena Fuseki

1. **Скачивание и установка:**

```
wget https://archive.apache.org/dist/jena/binaries/apache-jena-fuseki-
4.10.0.tar.gz
tar -xzf apache-jena-fuseki-4.10.0.tar.gz
cd apache-jena-fuseki-4.10.0
```

2. **Запуск Fuseki сервера:**

```
./fuseki-server --mem --update /ds
```

3. **Проверка работы:**
   - Откройте браузер: `http://localhost:3030`
   - Убедитесь, что сервер доступен

Этап 2: Загрузка онтологии в Fuseki

1. **Загрузка данных через веб-интерфейс:**
   - Перейдите в `Manage Datasets` → `Add new dataset`
   - Выберите `in-memory` хранилище
   - Имя: `pizza_ds`
   - Нажмите `Create`
2. **Загрузка онтологии:**
   - Перейдите в `Upload data`
   - Загрузите файл `pizza.owl`
   - Нажмите `Upload`
3. **Проверка загрузки:**
   - Перейдите во вкладку `Query`
   - Выполните тестовый запрос:

```
SELECT (COUNT(*) AS ?count) WHERE { ?s ?p ?o }
```

Этап 3: Написание базовых SPARQL-запросов

1. **Создание скрипта для работы с SPARQL:**

```
touch sparql_queries.py
```

2. **Настройка подключения:**

```
from SPARQLWrapper import SPARQLWrapper, JSON, XML
import pandas as pd

# Настройка SPARQL endpoint
sparql = SPARQLWrapper("http://localhost:3030/ds/sparql")
sparql.setReturnFormat(JSON)

def run_query(query):
    sparql.setQuery(query)
    try:
        results = sparql.query().convert()
        return results
    except Exception as e:
        print(f"Ошибка выполнения запроса: {e}")
        return None
```

3. **Запрос 1: Получение всех классов онтологии**

```
query1 = """
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?class ?label
WHERE {
    ?class a owl:Class .
    OPTIONAL { ?class rdfs:label ?label }
}
ORDER BY ?class
"""

results1 = run_query(query1)
print("Классы онтологии:")
for result in results1["results"]["bindings"]:
    print(f"{result['class']['value']} - {result.get('label',
{}).get('value', 'No label')}")
```

4. **Запрос 2: Поиск всех пицц**

```
query2 = """
PREFIX pizza: <http://www.co-ode.org/ontologies/pizza/pizza.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?pizza ?name
WHERE {
    ?pizza a pizza:Pizza .
    ?pizza rdfs:label ?name .
}
ORDER BY ?name
"""
```

```
results2 = run_query(query2)
print("\nВсе пиццы:")
for result in results2["results"]["bindings"]:
    print(result['name']['value'])
```

Этап 4: Сложные запросы с фильтрацией

1. **Запрос 3: Пиццы с определенной начинкой**

```
query3 = """
PREFIX pizza: <http://www.co-ode.org/ontologies/pizza/pizza.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?pizza ?name ?topping
WHERE {
    ?pizza a pizza:Pizza .
    ?pizza rdfs:label ?name .
    ?pizza pizza:hasTopping ?toppingObj .
    ?toppingObj rdfs:label ?topping .
    FILTER (CONTAINS(LCASE(?topping), "mushroom"))
}
"""

results3 = run_query(query3)
print("\nПиццы с грибами:")
for result in results3["results"]["bindings"]:
    print(f"{result['name']['value']} - {result['topping']['value']}")
```

2. **Запрос 4: Статистика по начинкам**

```
query4 = """
PREFIX pizza: <http://www.co-ode.org/ontologies/pizza/pizza.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?topping (COUNT(?pizza) AS ?count)
WHERE {
    ?pizza a pizza:Pizza .
    ?pizza pizza:hasTopping ?toppingObj .
    ?toppingObj rdfs:label ?topping .
}
GROUP BY ?topping
ORDER BY DESC(?count)
LIMIT 10
"""

results4 = run_query(query4)
print("\nПопулярные начинки:")
```

```
for result in results4["results"]["bindings"]:
    print(f"{result['topping']['value']}: {result['count']['value']}")
```

Этап 5: CONSTRUCT-запросы для создания новых данных

1. **Запрос 5: Создание RDF-графа вегетарианских пицц**

```
query5 = """
PREFIX pizza: <http://www.co-ode.org/ontologies/pizza/pizza.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ex: <http://example.org/vegetarian#>

CONSTRUCT {
    ?pizza ex:isVegetarian true .
    ?pizza ex:hasTopping ?topping .
}
WHERE {
    ?pizza a pizza:Pizza .
    ?pizza rdfs:label ?name .
    ?pizza pizza:hasTopping ?toppingObj .
    ?toppingObj rdfs:label ?topping .
    FILTER NOT EXISTS {
        ?toppingObj a pizza:MeatTopping .
    }
}
"""

# Для CONSTRUCT запросов меняем формат вывода
sparql.setReturnFormat(XML)
results5 = run_query(query5)
print("CONSTRUCT запрос выполнен")

# Сохранение результатов
with open("vegetarian_pizzas.rdf", "w") as f:
    f.write(results5.toxml())
```

Этап 6: Работа с онтологией через RDFLib

1. **Альтернативный способ работы с данными:**

```
from rdflib import Graph, Namespace
from rdflib.plugins.stores import sparqlstore

# Создание графа с SPARQL endpoint
store = sparqlstore.SPARQLUpdateStore()
store.open(('http://localhost:3030/ds/sparql',
'http://localhost:3030/ds/update'))

g = Graph(store)
```

```
# Определение namespace
PIZZA = Namespace("http://www.co-ode.org/ontologies/pizza/pizza.owl#")
RDFS = Namespace("http://www.w3.org/2000/01/rdf-schema#")

# Запрос через RDFLib
query6 = """
SELECT ?pizza ?name
WHERE {
    ?pizza a pizza:Pizza .
    ?pizza rdfs:label ?name .
}
LIMIT 5
"""

results6 = g.query(query6, initNs={"pizza": PIZZA, "rdfs": RDFS})
print("\nРезультаты через RDFLib:")
for row in results6:
    print(f"{row.pizza} - {row.name}")
```

Этап 7: Создание комплексных отчетов

1. **Генерация отчета по онтологии:**

```
def generate_ontology_report():
    queries = {
        "total_classes": """
            SELECT (COUNT(DISTINCT ?class) AS ?count)
            WHERE { ?class a owl:Class }
        """,
        "total_properties": """
            SELECT (COUNT(DISTINCT ?prop) AS ?count)
            WHERE { ?prop a owl:ObjectProperty }
        """,
        "total_individuals": """
            SELECT (COUNT(DISTINCT ?ind) AS ?count)
            WHERE { ?ind a owl:NamedIndividual }
        """
    }

    report = {}
    for name, query in queries.items():
        results = run_query(query)
        if results:
            count = results["results"]["bindings"][0]["count"]["value"]
            report[name] = count

    # Сохранение отчета
    df = pd.DataFrame([report])
    df.to_csv("ontology_report.csv", index=False)
    return report
```

```
ontology_stats = generate_ontology_report()
print("\nСтатистика онтологии:")
for key, value in ontology_stats.items():
    print(f"{key}: {value}")
```

Этап 8: Интеграционные тесты

1. **Тестирование различных endpoint:**

```
def test_endpoints():
    endpoints = [
        "http://localhost:3030/ds/sparql",
        "http://dbpedia.org/sparql",
        "http://query.wikidata.org/sparql"
    ]

    test_query = "SELECT (COUNT(*) AS ?count) WHERE { ?s ?p ?o } LIMIT 1"

    for endpoint in endpoints:
        try:
            sparql = SPARQLWrapper(endpoint)
            sparql.setQuery(test_query)
            sparql.setReturnFormat(JSON)
            results = sparql.query().convert()
            print(f"{endpoint}: Работает ({results['results']['bindings'][0]
['count']['value']} triplets)")
        except:
            print(f"{endpoint}: Не доступен")

test_endpoints()
```

Требования к оформлению и отчету

Критерии оценки для Части 2:

- **Удовлетворительно:** Успешно выполнены Этапы 1-3 (запуск Fuseki, базовые SELECT-запросы). Получены списки классов и пицц.
- **Хорошо:** Дополнительно успешно выполнен Этап 4-5 (сложные запросы с фильтрацией, CONSTRUCT-запросы). Создан RDF-граф вегетарианских пицц.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы Этапы 6-8: работа через RDFLib, генерация отчетов, интеграционные тесты. Проанализирована структура онтологии через SPARQL.

Рекомендуемая литература

1. **SPARQL 1.1 Specification:** https://www.w3.org/TR/sparql11-query/

2. **Apache Jena Documentation:** https://jena.apache.org/documentation/fuseki2/
3. **SPARQLWrapper Documentation:** https://sparqlwrapper.readthedocs.io/
4. **RDFLib Documentation:** https://rdflib.readthedocs.io/
5. **Learning SPARQL:** https://www.learnsparql.com/

Лабораторная работа №7-8, Часть 3: Извлечение данных с помощью LLM

Цель работы: Исследовать возможности использования языковых моделей для генерации SPARQL-запросов по текстовым описаниям на естественном языке. Получить практические навыки интеграции LLM с семантическими технологиями.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Окружение:** Conda (`mlops-lab`)
- **Библиотеки:** `transformers`, `sparqlwrapper`, `rdflib`, `openai` (опционально)
- **Модели:** GPT-3.5/4, Llama 2, Mistral (через Hugging Face)
- **Инструменты:** Jena Fuseki, Protégé

Теоретическая часть

1. Генерация SPARQL через NL-to-SPARQL Преобразование естественного языка (Natural Language) в SPARQL:

- **Zero-shot подход:** Генерация без примеров
- **Few-shot подход:** Генерация с несколькими примерами
- **Fine-tuning:** Специализированное обучение на парах (вопрос-SPARQL)

2. Архитектура решения

- **Вход:** Текстовый запрос на естественном языке
- **Обработка:** LLM генерирует SPARQL-запрос
- **Валидация:** Проверка синтаксиса и выполнение запроса
- **Итерация:** Исправление ошибок через feedback loop

3. Оценка качества

- **Синтаксическая корректность:** Правильность SPARQL-синтаксиса
- **Семантическая корректность:** Соответствие intent пользователя
- **Эффективность:** Оптимальность выполнения запроса

Задание на практическую реализацию

Этап 1: Настройка окружения и подключение к LLM

1. **Установка необходимых пакетов:**

```
conda activate mlops-lab
pip install transformers sparqlwrapper rdflib openai
```

2. **Создание скрипта для работы с LLM:**

```
touch llm_sparql_generation.py
```

3. **Настройка подключения к LLM (Hugging Face):**

```
from transformers import pipeline, AutoTokenizer, AutoModelForCausalLM
import torch

class SPARQLGenerator:
    def __init__(self, model_name="mistralai/Mistral-7B-Instruct-v0.2"):
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModelForCausalLM.from_pretrained(
            model_name,
            torch_dtype=torch.float16,
            device_map="auto"
        )
        self.tokenizer.pad_token = self.tokenizer.eos_token

    def generate_sparql(self, natural_language_query):
        prompt = f"""
        Convert the following natural language query to SPARQL for the Pizza
ontology.
        Use prefixes: PREFIX pizza: <http://www.co-
ode.org/ontologies/pizza/pizza.owl#>
        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

        Natural language: {natural_language_query}
        SPARQL:
        """

        inputs = self.tokenizer(prompt, return_tensors="pt",
truncation=True, max_length=512)
        with torch.no_grad():
            outputs = self.model.generate(
                **inputs,
                max_new_tokens=200,
                temperature=0.7,
                do_sample=True,
                pad_token_id=self.tokenizer.eos_token_id
            )

        generated_text = self.tokenizer.decode(outputs[0],
skip_special_tokens=True)
        sparql_query = generated_text.split("SPARQL:")[-1].strip()

        return sparql_query
```

Этап 2: Базовая генерация SPARQL-запросов

1. **Функция для тестирования генерации:**

```
def test_basic_generation():
    generator = SPARQLGenerator()

    test_queries = [
        "Find all pizzas that have mushroom as topping",
        "Show me vegetarian pizzas",
        "List pizzas with spicy toppings",
        "Find pizzas that are not too spicy",
        "Show me pizzas with cheese and tomato"
    ]

    for query in test_queries:
        print(f"\nNatural language: {query}")
        sparql = generator.generate_sparql(query)
        print(f"Generated SPARQL: {sparql}")
        print("-" * 50)

test_basic_generation()
```

Этап 3: Валидация и выполнение сгенерированных запросов

1. **SPARQL валидатор и исполнитель:**

```
from SPARQLWrapper import SPARQLWrapper, JSON, SPARQLExceptions
import re

class SPARQLValidator:
    def __init__(self, endpoint="http://localhost:3030/ds/sparql"):
        self.endpoint = endpoint
        self.sparql = SPARQLWrapper(endpoint)
        self.sparql.setReturnFormat(JSON)

    def validate_syntax(self, query):
        """Проверка синтаксиса SPARQL"""
        try:
            # Базовая проверка структуры
            if not query.strip().upper().startswith(('SELECT', 'CONSTRUCT',
'ASK', 'DESCRIBE')):
                return False, "Query must start with SELECT, CONSTRUCT, ASK
or DESCRIBE"

            # Проверка наличия WHERE clause
            if "WHERE" not in query.upper():
                return False, "Missing WHERE clause"

            return True, "Syntax appears valid"
        except Exception as e:
            return False, f"Syntax validation error: {e}"
```

```
    def execute_query(self, query):
        """Выполнение SPARQL-запроса"""
        try:
            self.sparql.setQuery(query)
            results = self.sparql.query().convert()
            return True, results
        except SPARQLExceptions.QueryBadFormed as e:
            return False, f"Malformed query: {e}"
        except Exception as e:
            return False, f"Execution error: {e}"

def test_generated_queries():
    generator = SPARQLGenerator()
    validator = SPARQLValidator()

    test_cases = [
        "Find all pizzas with mushroom",
        "Show me non-vegetarian pizzas",
        "List pizzas with exactly two toppings"
    ]

    for query in test_cases:
        print(f"\n{'='*60}")
        print(f"Testing: {query}")

        # Генерация SPARQL
        sparql = generator.generate_sparql(query)
        print(f"Generated: {sparql}")

        # Валидация синтаксиса
        is_valid, syntax_msg = validator.validate_syntax(sparql)
        print(f"Syntax valid: {is_valid} - {syntax_msg}")

        # Выполнение запроса
        if is_valid:
            success, result = validator.execute_query(sparql)
            if success:
                print("Query executed successfully!")
                if "results" in result and "bindings" in result["results"]:
                    bindings = result["results"]["bindings"]
                    print(f"Results: {len(bindings)} found")
                    for i, binding in enumerate(bindings[:3]):
                        print(f"  {i+1}. {binding}")
            else:
                print(f"Execution failed: {result}")
```

Этап 4: Few-shot обучение через промпты

1. **Улучшенный генератор с примерами:**

```
class ImprovedSPARQLGenerator(SPARQLGenerator):
    def __init__(self, model_name="mistralai/Mistral-7B-Instruct-v0.2"):
        super().__init__(model_name)
        self.examples = [
            {
                "nl": "Find all pizzas with mushroom topping",
                "sparql": """
                PREFIX pizza: <http://www.co-
ode.org/ontologies/pizza/pizza.owl#>
                PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

                SELECT ?pizza ?name
                WHERE {
                    ?pizza a pizza:Pizza .
                    ?pizza rdfs:label ?name .
                    ?pizza pizza:hasTopping ?topping .
                    ?topping rdfs:label ?toppingName .
                    FILTER (CONTAINS(LCASE(?toppingName), "mushroom"))
                }
                """
            },
            {
                "nl": "Show me vegetarian pizzas",
                "sparql": """
                PREFIX pizza: <http://www.co-
ode.org/ontologies/pizza/pizza.owl#>
                PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

                SELECT ?pizza ?name
                WHERE {
                    ?pizza a pizza:Pizza .
                    ?pizza rdfs:label ?name .
                    FILTER NOT EXISTS {
                        ?pizza pizza:hasTopping ?topping .
                        ?topping a pizza:MeatTopping .
                    }
                }
                """
            }
        ]

    def generate_with_examples(self, natural_language_query):
        prompt = "Convert natural language queries to SPARQL for Pizza
ontology.\n\n"

        # Добавление примеров
        for example in self.examples:
            prompt += f"NL: {example['nl']}\nSPARQL:
{example['sparql']}\n\n"

        prompt += f"NL: {natural_language_query}\nSPARQL:"

        inputs = self.tokenizer(prompt, return_tensors="pt",
```

```
truncation=True, max_length=1024)
        with torch.no_grad():
            outputs = self.model.generate(
                **inputs,
                max_new_tokens=300,
                temperature=0.3,  # Более детерминированное поколение
                do_sample=True
            )

        generated_text = self.tokenizer.decode(outputs[0],
skip_special_tokens=True)
        sparql_query = generated_text.split("SPARQL:")[-1].strip()

        return sparql_query
```

Этап 5: Интеграция с семантическим валидатором

1. **Расширенная валидация с семантической проверкой:**

```
class SemanticValidator(SPARQLValidator):
    def validate_ontology_compatibility(self, query):
        """Проверка совместимости с онтологией"""
        try:
            # Проверка использования правильных префиксов
            if "pizza:" not in query:
                return False, "Missing pizza prefix usage"

            # Проверка существующих классов/свойств
            class_check = self.execute_query("""
                PREFIX pizza: <http://www.co-
ode.org/ontologies/pizza/pizza.owl#>
                SELECT DISTINCT ?class WHERE { ?class a owl:Class }
            """)

            if class_check[0]:
                valid_classes = [str(r['class']['value']) for r in
class_check[1]['results']['bindings']]
                # Простая проверка на наличие pizza:Pizza в запросе
                if "pizza:Pizza" in query and "http://www.co-
ode.org/ontologies/pizza/pizza.owl#Pizza" in valid_classes:
                    return True, "Ontology compatibility check passed"

            return False, "May reference non-existent ontology elements"

        except Exception as e:
            return False, f"Ontology validation error: {e}"

def comprehensive_test():
    generator = ImprovedSPARQLGenerator()
    validator = SemanticValidator()
```

```
    complex_queries = [
        "Find pizzas that have both cheese and tomato toppings",
        "Show me spicy pizzas that are not too expensive",
        "List vegetarian pizzas with exactly three toppings"
    ]

    results = []
    for query in complex_queries:
        print(f"\nTesting complex query: {query}")

        # Генерация с примерами
        sparql = generator.generate_with_examples(query)

        # Многоуровневая валидация
        syntax_ok, syntax_msg = validator.validate_syntax(sparql)
        ontology_ok, ontology_msg =
validator.validate_ontology_compatibility(sparql)

        result = {
            "query": query,
            "generated_sparql": sparql,
            "syntax_valid": syntax_ok,
            "ontology_compatible": ontology_ok,
            "execution_result": None
        }

        if syntax_ok and ontology_ok:
            success, exec_result = validator.execute_query(sparql)
            result["execution_success"] = success
            result["execution_result"] = exec_result if success else
str(exec_result)

        results.append(result)

        # Вывод результатов
        print(f"Syntax: {syntax_ok} ({syntax_msg})")
        print(f"Ontology: {ontology_ok} ({ontology_msg})")
        if result.get("execution_success"):
            print("Execution: Successful")

    return results
```

Этап 6: Оценка качества и метрики

1. **Система оценки сгенерированных запросов:**

```
def evaluate_sparql_generation():
    test_dataset = [
        {
            "nl": "Find pizzas with mushroom",
            "expected_patterns": ["pizza:hasTopping", "mushroom", "FILTER"],
```

```
            "min_results": 1
        },
        {
            "nl": "Show vegetarian pizzas",
            "expected_patterns": ["FILTER NOT EXISTS", "pizza:MeatTopping"],
            "min_results": 3
        }
    ]

    generator = ImprovedSPARQLGenerator()
    validator = SemanticValidator()

    evaluation_results = []

    for test_case in test_dataset:
        nl_query = test_case["nl"]
        print(f"\nEvaluating: {nl_query}")

        # Генерация
        sparql = generator.generate_with_examples(nl_query)

        # Проверка ожидаемых паттернов
        pattern_matches = sum(1 for pattern in
test_case["expected_patterns"] if pattern in sparql)
        pattern_score = pattern_matches /
len(test_case["expected_patterns"])

        # Выполнение и проверка результатов
        exec_success, exec_result = validator.execute_query(sparql)
        result_count = len(exec_result["results"]["bindings"]) if
exec_success else 0
        result_score = 1.0 if result_count >= test_case["min_results"] else
result_count / test_case["min_results"]

        # Общая оценка
        total_score = (pattern_score * 0.6) + (result_score * 0.4)

        evaluation_results.append({
            "query": nl_query,
            "generated_sparql": sparql,
            "pattern_score": pattern_score,
            "result_score": result_score,
            "total_score": total_score,
            "status": "PASS" if total_score >= 0.7 else "FAIL"
        })

        print(f"Score: {total_score:.2f} (Patterns: {pattern_score:.2f},
Results: {result_score:.2f})")
        print(f"Status: {evaluation_results[-1]['status']}")

    # Сохранение результатов оценки
    import json
    with open("sparql_generation_evaluation.json", "w") as f:
        json.dump(evaluation_results, f, indent=2)
```

```
    return evaluation_results
```

Этап 7: Создание демонстрационного интерфейса

1. **Простой веб-интерфейс с Flask:**

```
from flask import Flask, request, jsonify
import threading

app = Flask(__name__)
generator = ImprovedSPARQLGenerator()
validator = SemanticValidator()

@app.route('/generate-sparql', methods=['POST'])
def generate_sparql_endpoint():
    data = request.json
    nl_query = data.get('query', '')

    if not nl_query:
        return jsonify({"error": "No query provided"}), 400

    try:
        # Генерация SPARQL
        sparql = generator.generate_with_examples(nl_query)

        # Валидация
        syntax_ok, syntax_msg = validator.validate_syntax(sparql)
        ontology_ok, ontology_msg =
validator.validate_ontology_compatibility(sparql)

        response = {
            "natural_language_query": nl_query,
            "generated_sparql": sparql,
            "validation": {
                "syntax": {"valid": syntax_ok, "message": syntax_msg},
                "ontology": {"valid": ontology_ok, "message": ontology_msg}
            }
        }

        return jsonify(response)

    except Exception as e:
        return jsonify({"error": str(e)}), 500

def run_flask_app():
    app.run(host='0.0.0.0', port=5001, debug=False)

# Запуск в отдельном потоке
# flask_thread = threading.Thread(target=run_flask_app)
# flask_thread.start()
```

Требования к оформлению и отчету

Критерии оценки для Части 3:

- **Удовлетворительно:** Успешно выполнены Этапы 1-2 (настройка LLM, базовая генерация). Сгенерированы SPARQL-запросы для простых вопросов.
- **Хорошо:** Дополнительно успешно выполнен Этап 3-4 (валидация, few-shot обучение). Реализована система проверки синтаксиса и семантики.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы Этапы 5-7: семантическая валидация, система оценки, демонстрационный интерфейс. Проведен комплексный анализ качества генерации.

Рекомендуемая литература

1. **NL-to-SPARQL Survey:** https://arxiv.org/abs/2104.07281
2. **GPT-3 for SPARQL Generation:** https://arxiv.org/abs/2210.07812
3. **Hugging Face Transformers:** https://huggingface.co/docs/transformers
4. **SPARQL 1.1 Specification:** https://www.w3.org/TR/sparql11-query/
5. **Prompt Engineering Guide:** https://www.promptingguide.ai/

Лабораторная работа №9-10, Часть 1: Развертывание ML-моделей с FastAPI

Цель работы: Освоить создание production-ready веб-сервисов для обслуживания ML-моделей с использованием FastAPI. Получить практические навыки разработки RESTful API, валидации данных и документирования эндпоинтов.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Окружение:** Conda (`mlops-lab`)
- **Фреймворк:** FastAPI, Uvicorn
- **Библиотеки:** `pydantic`, `scikit-learn`, `transformers`, `numpy`
- **Модель:** Классификатор эмоций (из предыдущих работ)
- **Документация:** Swagger/OpenAPI

Теоретическая часть

1. FastAPI для ML-сервисов FastAPI — современный фреймворк для создания API на Python:

- **Высокая производительность:** На основе Starlette и Pydantic
- **Автодокументирование:** Генерация OpenAPI-спецификации
- **Валидация данных:** Использование Pydantic моделей
- **Асинхронность:** Поддержка async/await

2. Архитектура ML-сервиса

- **Загрузка модели:** Инициализация при запуске приложения
- **Предобработка:** Валидация и преобразование входных данных
- **Инференс:** Выполнение предсказания моделью
- **Постобработка:** Форматирование результатов
- **Логирование:** Мониторинг работы сервиса

3. Производственные практики

- **Health checks:** Проверка работоспособности сервиса
- **Валидация входных данных:** Защита от некорректных запросов
- **Обработка ошибок:** Graceful degradation
- **Метрики:** Мониторинг производительности

Задание на практическую реализацию

Этап 1: Установка и настройка окружения

1. **Установка необходимых пакетов:**

```
conda activate mlops-lab
pip install fastapi uvicorn pydantic scikit-learn numpy
```

2. **Создание структуры проекта:**

```
mkdir emotion-api
cd emotion-api
mkdir models routers utils
touch main.py models/emotion_model.py routers/predict.py utils/validation.py
```

Этап 2: Создание моделей данных с Pydantic

1. **Модели для валидации входных/выходных данных:**

```
# utils/validation.py
from pydantic import BaseModel, Field
from typing import List, Dict, Optional

class PredictionRequest(BaseModel):
    text: str = Field(..., min_length=1, max_length=1000,
                      description="Текст для анализа эмоций")
    model_version: Optional[str] = Field("default",
                                         description="Версия модели для
использования")

class EmotionPrediction(BaseModel):
    emotion: str = Field(..., description="Предсказанная эмоция")
    confidence: float = Field(..., ge=0.0, le=1.0,
                              description="Уверенность предсказания")

class PredictionResponse(BaseModel):
    request_id: str = Field(..., description="Уникальный ID запроса")
    predictions: List[EmotionPrediction] = Field(...,
                                                 description="Список
предсказаний")
    model_version: str = Field(..., description="Использованная версия
модели")
    processing_time: float = Field(..., description="Время обработки в
секундах")

class HealthResponse(BaseModel):
    status: str = Field(..., description="Статус сервиса")
    model_loaded: bool = Field(..., description="Модель загружена")
    timestamp: str = Field(..., description="Время проверки")
```

Этап 3: Реализация ML-модели для обслуживания

1. **Класс для работы с моделью:**

```
# models/emotion_model.py
import numpy as np
import pickle
import time
from typing import List, Dict, Tuple
import logging

logger = logging.getLogger(__name__)

class EmotionClassifier:
    def __init__(self, model_path: str = None):
        self.model = None
        self.vectorizer = None
        self.label_encoder = None
        self.model_version = "v1.0"
        self.is_loaded = False

        if model_path:
            self.load_model(model_path)

    def load_model(self, model_path: str):
        """Загрузка обученной модели"""
        try:
            # В реальном сценарии здесь была бы загрузка вашей модели
            # Для демонстрации создадим простой классификатор
            from sklearn.ensemble import RandomForestClassifier
            from sklearn.feature_extraction.text import TfidfVectorizer
            from sklearn.preprocessing import LabelEncoder

            # Создание демонстрационной модели
            self.vectorizer = TfidfVectorizer(max_features=1000)
            self.label_encoder = LabelEncoder()

            # Пример тренировочных данных
            texts = [
                "I am so happy today", "This is wonderful news",
                "I feel angry about this", "This makes me furious",
                "I am scared of what might happen", "This is terrifying",
                "I love this so much", "This is amazing",
                "I am sad about this", "This is disappointing"
            ]
            labels = ["joy", "joy", "anger", "anger", "fear", "fear",
                      "love", "love", "sadness", "sadness"]

            # Обучение компонентов
            X = self.vectorizer.fit_transform(texts)
            y = self.label_encoder.fit_transform(labels)

            self.model = RandomForestClassifier(n_estimators=10,
random_state=42)
            self.model.fit(X, y)
```

```
            self.is_loaded = True
            logger.info(f"Model loaded successfully. Version:
{self.model_version}")

        except Exception as e:
            logger.error(f"Error loading model: {e}")
            self.is_loaded = False
            raise

    def predict(self, text: str) -> Tuple[str, float]:
        """Выполнение предсказания для одного текста"""
        if not self.is_loaded:
            raise RuntimeError("Model is not loaded")

        start_time = time.time()

        try:
            # Преобразование текста в фичи
            X = self.vectorizer.transform([text])

            # Предсказание
            probabilities = self.model.predict_proba(X)[0]
            predicted_class_idx = np.argmax(probabilities)
            confidence = probabilities[predicted_class_idx]

            # Декодирование класса
            emotion =
self.label_encoder.inverse_transform([predicted_class_idx])[0]

            processing_time = time.time() - start_time
            logger.info(f"Prediction completed in {processing_time:.4f}s")

            return emotion, float(confidence)

        except Exception as e:
            logger.error(f"Prediction error: {e}")
            raise

    def predict_batch(self, texts: List[str]) -> List[Tuple[str, float]]:
        """Пакетное предсказание для нескольких текстов"""
        results = []
        for text in texts:
            try:
                emotion, confidence = self.predict(text)
                results.append((emotion, confidence))
            except Exception as e:
                logger.error(f"Error processing text: {text}, error: {e}")
                results.append(("error", 0.0))
        return results

# Создание глобального экземпляра модели
emotion_model = EmotionClassifier()
```

Этап 4: Создание эндпоинтов API

1. **Роутер для предсказаний:**

```
# routers/predict.py
from fastapi import APIRouter, HTTPException, BackgroundTasks
import uuid
import time
import logging
from typing import List

from utils.validation import PredictionRequest, PredictionResponse,
EmotionPrediction
from models.emotion_model import emotion_model

router = APIRouter(prefix="/predict", tags=["prediction"])
logger = logging.getLogger(__name__)

@router.post("/emotion", response_model=PredictionResponse)
async def predict_emotion(request: PredictionRequest, background_tasks:
BackgroundTasks):
    """
    Предсказание эмоции для текста

    - **text**: Текст для анализа (1-1000 символов)
    - **model_version**: Версия модели (опционально)
    """
    try:
        start_time = time.time()
        request_id = str(uuid.uuid4())

        # Проверка загрузки модели
        if not emotion_model.is_loaded:
            raise HTTPException(status_code=503, detail="Model not loaded")

        # Выполнение предсказания
        emotion, confidence = emotion_model.predict(request.text)

        # Формирование ответа
        processing_time = time.time() - start_time

        prediction = EmotionPrediction(
            emotion=emotion,
            confidence=confidence
        )

        response = PredictionResponse(
            request_id=request_id,
            predictions=[prediction],
            model_version=emotion_model.model_version,
            processing_time=processing_time
```

```
        )

        # Логирование в фоне
        background_tasks.add_task(
            logger.info,
            f"Request {request_id} processed in {processing_time:.4f}s"
        )

        return response

    except Exception as e:
        logger.error(f"Prediction failed: {e}")
        raise HTTPException(status_code=500, detail=str(e))

@router.post("/emotion/batch", response_model=PredictionResponse)
async def predict_emotion_batch(texts: List[str], background_tasks:
BackgroundTasks):
    """
    Пакетное предсказание эмоций для нескольких текстов

    - **texts**: Список текстов для анализа
    """
    try:
        start_time = time.time()
        request_id = str(uuid.uuid4())

        if not emotion_model.is_loaded:
            raise HTTPException(status_code=503, detail="Model not loaded")

        if len(texts) > 100:  # Ограничение на размер батча
            raise HTTPException(status_code=400, detail="Too many texts in
batch")

        # Пакетное предсказание
        results = emotion_model.predict_batch(texts)

        # Формирование ответа
        predictions = []
        for emotion, confidence in results:
            predictions.append(EmotionPrediction(
                emotion=emotion,
                confidence=confidence
            ))

        processing_time = time.time() - start_time

        response = PredictionResponse(
            request_id=request_id,
            predictions=predictions,
            model_version=emotion_model.model_version,
            processing_time=processing_time
        )

        background_tasks.add_task(
```

```
            logger.info,
            f"Batch request {request_id} processed {len(texts)} texts in
{processing_time:.4f}s"
        )

        return response

    except Exception as e:
        logger.error(f"Batch prediction failed: {e}")
        raise HTTPException(status_code=500, detail=str(e))
```

Этап 5: Создание основного приложения

1. **Главный файл приложения:**

```
# main.py
from fastapi import FastAPI, HTTPException
from contextlib import asynccontextmanager
import logging
import time

from routers.predict import router as predict_router
from utils.validation import HealthResponse
from models.emotion_model import emotion_model

# Настройка логирования
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Startup: загрузка модели
    startup_time = time.time()
    try:
        emotion_model.load_model("demo_model.pkl")  # Загрузка демо-модели
        load_time = time.time() - startup_time
        logger.info(f"Application started successfully. Model loaded in
{load_time:.2f}s")
    except Exception as e:
        logger.error(f"Failed to load model: {e}")

    yield  # Приложение работает

    # Shutdown: очистка ресурсов
    logger.info("Application shutting down")

# Создание приложения FastAPI
app = FastAPI(
```

```
    title="Emotion Classification API",
    description="API для классификации эмоций в тексте с использованием ML",
    version="1.0.0",
    lifespan=lifespan
)

# Подключение роутеров
app.include_router(predict_router)

@app.get("/", tags=["root"])
async def root():
    """Корневой эндпоинт с информацией о API"""
    return {
        "message": "Emotion Classification API",
        "version": "1.0.0",
        "docs": "/docs",
        "health": "/health"
    }

@app.get("/health", response_model=HealthResponse, tags=["monitoring"])
async def health_check():
    """Проверка здоровья сервиса"""
    return HealthResponse(
        status="healthy" if emotion_model.is_loaded else "degraded",
        model_loaded=emotion_model.is_loaded,
        timestamp=time.strftime("%Y-%m-%d %H:%M:%S")
    )

@app.get("/model/info", tags=["model"])
async def model_info():
    """Информация о загруженной модели"""
    if not emotion_model.is_loaded:
        raise HTTPException(status_code=503, detail="Model not loaded")

    return {
        "version": emotion_model.model_version,
        "status": "loaded",
        "type": "RandomForestClassifier"
    }

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(
        "main:app",
        host="0.0.0.0",
        port=8000,
        reload=True,  # Автоперезагрузка для разработки
        log_level="info"
    )
```

Этап 6: Тестирование API

1. **Создание тестового клиента:**

```
# test_client.py
import requests
import json

BASE_URL = "http://localhost:8000"

def test_health():
    response = requests.get(f"{BASE_URL}/health")
    print("Health Check:")
    print(json.dumps(response.json(), indent=2))

def test_single_prediction():
    data = {
        "text": "I am feeling absolutely wonderful today!",
        "model_version": "default"
    }

    response = requests.post(f"{BASE_URL}/predict/emotion", json=data)
    print("\nSingle Prediction:")
    print(json.dumps(response.json(), indent=2))

def test_batch_prediction():
    texts = [
        "This is amazing news!",
        "I am very angry about this situation",
        "I feel scared and anxious",
        "This makes me so happy"
    ]

    response = requests.post(f"{BASE_URL}/predict/emotion/batch",
json=texts)
    print("\nBatch Prediction:")
    print(json.dumps(response.json(), indent=2))

def test_invalid_request():
    data = {
        "text": ""  # Пустой текст
    }

    response = requests.post(f"{BASE_URL}/predict/emotion", json=data)
    print("\nInvalid Request:")
    print(f"Status: {response.status_code}")
    print(json.dumps(response.json(), indent=2))

if __name__ == "__main__":
    print("Testing Emotion Classification API")
    print("=" * 50)

    test_health()
    test_single_prediction()
```

```
    test_batch_prediction()
    test_invalid_request()
```

Этап 7: Запуск и использование API

1. **Запуск сервера:**

```
python main.py
```

2. **Проверка документации:**
   - Откройте браузер и перейдите по адресу: `http://localhost:8000/docs`
   - Изучите автоматически сгенерированную документацию Swagger
   - Протестируйте эндпоинты через UI

3. **Пример использования через curl:**

```
# Проверка здоровья
curl -X GET "http://localhost:8000/health"

# Одиночное предсказание
curl -X POST "http://localhost:8000/predict/emotion" \
     -H "Content-Type: application/json" \
     -d '{"text": "I am so happy today!", "model_version": "default"}'

# Пакетное предсказание
curl -X POST "http://localhost:8000/predict/emotion/batch" \
     -H "Content-Type: application/json" \
     -d '["Great news!", "This is terrible", "I am excited"]'
```

Требования к оформлению и отчету

Критерии оценки для Части 1:

- **Удовлетворительно:** Успешно выполнены Этапы 1-3 (создание структуры проекта, моделей данных). Приложение запускается без ошибок.
- **Хорошо:** Дополнительно успешно выполнен Этап 4-5 (реализация эндпоинтов, основного приложения). API отвечает на запросы, документация доступна.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы Этапы 6-7: тестовый клиент, обработка ошибок, валидация данных. API полностью функционально и готово к использованию.

Рекомендуемая литература

1. **FastAPI Documentation:** https://fastapi.tiangolo.com/
2. **Pydantic Documentation:** https://docs.pydantic.dev/

3. **Uvicorn Documentation:** https://www.uvicorn.org/
4. **REST API Best Practices:** https://restfulapi.net/
5. **ML Model Deployment Patterns:** https://mlops.githubapp.com/

Лабораторная работа №9-10, Часть 2: Контейнеризация ML-сервиса с Docker

Цель работы: Освоить процесс упаковки ML-приложения и модели в Docker-контейнер для обеспечения переносимости, воспроизводимости и развертывания в production-средах. Получить практические навыки создания Dockerfile, сборки образов и управления контейнерами.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Окружение:** Docker
- **Исходный код:** FastAPI-приложение из Части 1
- **Реестр образов:** Docker Hub (опционально)

Теоретическая часть

1. Контейнеризация ML-приложений Контейнеризация решает ключевые проблемы развертывания ML-моделей:

- **Воспроизводимость:** Гарантия, что приложение будет работать одинаково на любой системе.
- **Изоляция:** Все зависимости (библиотеки, версии Python, системные библиотеки) упакованы вместе с приложением.
- **Масштабируемость:** Легко запустить несколько экземпляров сервиса.
- **Упрощение деплоя:** Образ — это самодостаточная единица развертывания.

2. Dockerfile для Python-приложений Dockerfile — это инструкция по сборке образа. Ключевые этапы для ML-сервиса:

- **Базовый образ:** Выбор официального Python-образа с нужной версией.
- **Копирование кода:** Перенос файлов приложения в контейнер.
- **Установка зависимостей:** Установка Python-пакетов из `requirements.txt`.
- **Настройка окружения:** Установка переменных окружения.
- **Экспорт портов:** Определение порта, который слушает приложение.
- **Команда запуска:** Команда для запуска приложения при старте контейнера.

3. Многоступенчатая сборка (Multi-stage build) Продвинутая техника, позволяющая:

- Уменьшить итоговый размер образа.
- Отделить этапы сборки (например, компиляции) от этапа выполнения.
- Повысить безопасность (исключить из финального образа инструменты разработки).

Задание на практическую реализацию

Этап 1: Подготовка приложения к контейнеризации

1. **Создание `requirements.txt`:**

- В корне проекта `emotion-api` создайте файл `requirements.txt`.
- Добавьте в него все зависимости вашего приложения.

```
# requirements.txt
fastapi==0.104.1
uvicorn[standard]==0.24.0
pydantic==2.5.0
scikit-learn==1.3.2
numpy==1.26.2
# Добавьте другие зависимости, если они используются
```

2. **Модификация кода для production:**

- Убедитесь, что в `main.py` используется правильный способ запуска для production.
- Удалите или закомментируйте блок `if __name__ == "__main__":`, так как запуск будет осуществляться через `uvicorn` напрямую из CMD.

```
# main.py (исправления в конце файла)
# Удаляем или комментируем блок прямого запуска
# if __name__ == "__main__":
#     import uvicorn
#     uvicorn.run(...)
```

Этап 2: Создание Dockerfile

1. **Создание `Dockerfile`:**

- В корне проекта создайте файл с именем `Dockerfile` (без расширения).

```
# Dockerfile
# Используем официальный Python образ с нужной версией
FROM python:3.10-slim

# Устанавливаем рабочую директорию внутри контейнера
WORKDIR /app

# Копируем файл с зависимостями в первую очередь (для кэширования слоя)
COPY requirements.txt .

# Обновляем pip и устанавливаем зависимости
RUN pip install --no-cache-dir --upgrade pip && \
    pip install --no-cache-dir -r requirements.txt

# Копируем весь исходный код проекта в контейнер
COPY . .

# Создаем непривилегированного пользователя для безопасности
RUN useradd -m -u 1000 user
```

```
USER user

# Сообщаем Docker, что контейнер слушает на порту 8000
EXPOSE 8000

# Команда для запуска приложения с Uvicorn
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Этап 3: Создание `.dockerignore`

1. **Создание `.dockerignore`:**
   - Создайте файл `.dockerignore`, чтобы исключить из образа ненужные файлы (кеш Python, виртуальные окружения, файлы IDE), что уменьшит размер образа и ускорит сборку.

```
# .dockerignore
__pycache__
*.pyc
*.pyo
*.pyd
.Python
env/
venv/
.venv
.vscode
.idea
*.log
.git
.dockerignore
Dockerfile
README.md
```

Этап 4: Сборка Docker-образа

1. **Сборка образа:**
   - Откройте терминал в корневой директории проекта (`emotion-api`).
   - Выполните команду для сборки образа. Флаг `-t` задает имя и тег образа.

```
docker build -t emotion-classifier-api:1.0 .
```

2. **Проверка собранного образа:**
   - Убедитесь, что образ появился в списке локальных образов.

```
docker images
```

- Вы должны увидеть что-то похожее:

```
REPOSITORY                TAG       IMAGE ID       CREATED         SIZE
emotion-classifier-api    1.0       abc123def456   2 minutes ago   1.2GB
```

Этап 5: Запуск контейнера

1. **Запуск контейнера:**
   - Запустите контейнер из собранного образа. Флаг `-p` пробрасывает порт из контейнера на хост (`порт_хоста:порт_контейнера`). Флаг `-d` запускает контейнер в фоновом режиме (detached).

```
docker run -d -p 8000:8000 --name emotion-api-container emotion-classifier-
api:1.0
```

2. **Проверка работы контейнера:**
   - Проверьте, что контейнер запущен.

```
docker ps
```

   - Проверьте логи контейнера на предмет ошибок.

```
docker logs emotion-api-container
```

   - В логах должна быть строка о успешном запуске Uvicorn, например: `"Application startup complete."`.

3. **Тестирование API:**
   - Откройте браузер и перейдите по адресу: `http://localhost:8000/docs`
   - Убедитесь, что Swagger UI загрузился и эндпоинты доступны.
   - Протестируйте эндпоинт `/health` и `/predict/emotion`, как это делалось в Части 1.

Этап 6: Оптимизация образа (опционально, для "Отлично")

1. **Использование многоступенчатой сборки:**
   - Модифицируйте `Dockerfile` для уменьшения размера итогового образа.

```

```

```
# Dockerfile (оптимизированный)
# Этап 1: сборка (builder)
FROM python:3.10-slim as builder

WORKDIR /app

# Копируем и устанавливаем зависимости
COPY requirements.txt .
RUN pip install --no-cache-dir --user -r requirements.txt

# Этап 2: финальный образ
FROM python:3.10-slim

WORKDIR /app

# Копируем установленные зависимости из этапа сборки
COPY --from=builder /root/.local /root/.local

# Копируем исходный код
COPY . .

# Создаем пользователя
RUN useradd -m -u 1000 user
USER user

# Убедимся, что скрипты в ~/.local доступны
ENV PATH=/root/.local/bin:$PATH

EXPOSE 8000

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

2. **Пересборка и сравнение образов:**
    - Соберите образ с новым именем тега.

```
docker build -t emotion-classifier-api:1.0-optimized .
```

    - Сравните размеры двух образов с помощью `docker images`. Оптимизированный образ должен быть меньше.

Этап 7: Управление контейнером и очистка

1. **Основные команды управления:**
    - Остановка контейнера:

```
docker stop emotion-api-container
```

- Запуск остановленного контейнера:

```
docker start emotion-api-container
```

- Перезагрузка контейнера:

```
docker restart emotion-api-container
```

- Удаление контейнера (остановите его перед удалением):

```
docker rm emotion-api-container
```

- Удаление образа:

```
docker rmi emotion-classifier-api:1.0
```

2. **Очистка системы Docker:**
   - Удаление всех остановленных контейнеров, неиспользуемых сетей и образов (опционально, для экономии места).

```
docker system prune -f
```

Требования к оформлению и отчету (для Части 2)

Критерии оценки для Части 2:

- **Удовлетворительно:** Успешно выполнены Этапы 1-4 (подготовка `requirements.txt`, создание `Dockerfile`, сборка образа). Образ собран без ошибок.
- **Хорошо:** Дополнительно успешно выполнен Этап 5 (контейнер запущен, API доступно и отвечает на запросы через Swagger UI). Предоставлены скриншоты работающего API.
- **Отлично:** Все задания выполнены в полном объеме. Дополнительно реализован Этап 6 (многоступенчатая сборка, размер образа уменьшен). В отчете приведено сравнение размеров образов и анализ проведенной оптимизации.

Рекомендуемая литература

1. **Docker Documentation:** https://docs.docker.com/

2. **Dockerfile Reference:** https://docs.docker.com/engine/reference/builder/
3. **Best practices for writing Dockerfiles:** https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
4. **Docker для Python-разработчиков:** https://docs.docker.com/language/python/
5. **Статья "Dockerizing a Python FastAPI App":** https://fastapi.tiangolo.com/deployment/docker/

Лабораторная работа №9-10, Часть 3: Тестирование работоспособности API через Swagger UI

Цель работы: Освоить методы тестирования и валидации RESTful API с использованием автоматически генерируемой документации Swagger UI. Получить практические навыки комплексного тестирования эндпоинтов, включая позитивные и негативные сценарии, проверку валидации данных и анализ ответов.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Окружение:** Docker (контейнер с API из Части 2)
- **Инструменты тестирования:** Swagger UI, curl (для сравнения)
- **Методологии:** Тестирование REST API, валидация JSON Schema

---

Теоретическая часть

1. Swagger/OpenAPI для тестирования API Swagger UI — это интерактивная документация, которая автоматически генерируется из OpenAPI-спецификации FastAPI:

- **Визуализация эндпоинтов:** Древовидное представление всех доступных методов API
- **Интерактивное тестирование:** Возможность отправки запросов прямо из браузера
- **Валидация схемы:** Автоматическая проверка структуры запросов и ответов
- **Автодокументирование:** Актуальная документация, синхронизированная с кодом

2. Стратегии тестирования REST API

- **Позитивное тестирование:** Проверка API с корректными данными
- **Негативное тестирование:** Проверка обработки ошибок с некорректными данными
- **Валидация данных:** Проверка соответствия входных и выходных данных схемам
- **Проверка статус-кодов:** Убедиться, что API возвращает правильные HTTP-коды

3. Критические аспекты тестирования ML-сервисов

- **Предсказуемость ответов:** Стабильность формата выходных данных
- **Обработка edge-cases:** Корректная работа с граничными значениями
- **Производительность:** Время отклика при различных нагрузках
- **Надежность:** Стабильность работы при длительной эксплуатации

Задание на практическую реализацию

Этап 1: Подготовка к тестированию

1. **Запуск API-сервиса:**
    - Убедитесь, что ваш контейнер с FastAPI-приложением запущен:

```
docker ps
```

- Если контейнер не запущен, выполните:

```
docker start emotion-api-container
```

2. **Открытие Swagger UI:**
   - В браузере перейдите по адресу: `http://localhost:8000/docs`
   - Дождитесь загрузки интерфейса Swagger UI
3. **Изучение структуры документации:**
   - Обратите внимание на разделение эндпоинтов по тегам (root, monitoring, prediction, model)
   - Изучите схемы запросов и ответов для каждого эндпоинта

Этап 2: Базовое тестирование эндпоинтов

1. **Тестирование корневого эндпоинта:**
   - Найдите эндпоинт `GET /` в разделе "root"
   - Нажмите "Try it out", затем "Execute"
   - Проанализируйте ответ:
     - **Status Code:** Должен быть `200`
     - **Response Body:** Должен содержать информацию о API
2. **Тестирование health-check:**
   - Найдите эндпоинт `GET /health` в разделе "monitoring"
   - Выполните запрос и проверьте:
     - **Status Code:** `200`
     - **model_loaded:** `true`
     - **status:** `"healthy"`

Этап 3: Комплексное тестирование эндпоинта предсказания

1. **Позитивный тест с корректными данными:**

```
{
  "text": "I am feeling absolutely wonderful today!",
  "model_version": "default"
}
```

   - В эндпоинте `POST /predict/emotion` нажмите "Try it out"
   - Вставьте JSON выше в поле "Request body"
   - Нажмите "Execute"

- **Проверки:**
  - **Status Code:** `200`
  - **Response Schema:** Соответствует `PredictionResponse`
  - **request_id:** Не пустой UUID
  - **predictions:** Массив с одним элементом
  - **emotion:** Одна из ожидаемых эмоций (joy, anger, fear, etc.)
  - **confidence:** Число между 0 и 1

2. **Тестирование пакетного предсказания:**

   - Перейдите к эндпоинту `POST /predict/emotion/batch`
   - Используйте следующий тестовый массив:

```
[
  "This is amazing news!",
  "I am very angry about this situation",
  "I feel scared and anxious",
  "This makes me so happy"
]
```

   - **Проверки:**
     - **Status Code:** `200`
     - **predictions:** Массив из 4 элементов
     - Каждый элемент имеет структуру `EmotionPrediction`

Этап 4: Негативное тестирование и валидация ошибок

1. **Тестирование пустого текста:**

```
{
  "text": "",
  "model_version": "default"
}
```

   - **Ожидаемый результат:**
     - **Status Code:** `422` (Validation Error)
     - **Response Body:** Детали ошибки валидации

2. **Тестирование слишком длинного текста:**

   - Сгенерируйте строку длиной более 1000 символов

```
{
  "text": "very long text...", // >1000 символов
  "model_version": "default"
}
```

- **Ожидаемый результат:** `422` (Validation Error)

3. **Тестирование некорректного JSON:**
   - Попробуйте отправить некорректный JSON:

```
{
  "text": "test",
  "model_version": "default"
```

   - **Ожидаемый результат:** `422` или `400` (Parse Error)

4. **Тестирование большого батча:**
   - Создайте массив со 101 элементом для батч-эндпоинта
   - **Ожидаемый результат:** `400` (Too many texts in batch)

Этап 5: Тестирование эндпоинта информации о модели

1. **Запрос информации о модели:**
   - Найдите эндпоинт `GET /model/info`
   - Выполните запрос и проверьте:
     - **Status Code:** `200`
     - **version:** Версия модели
     - **status:** `"loaded"`
     - **type:** Тип модели

Этап 6: Сравнение с curl-запросами

1. **Тестирование через командную строку:**
   - Выполните те же запросы через curl для сравнения:

```
# Health check
curl -X GET "http://localhost:8000/health"

# Одиночное предсказание
curl -X POST "http://localhost:8000/predict/emotion" \
     -H "Content-Type: application/json" \
     -d '{"text": "I am so happy today!", "model_version": "default"}'

# Пакетное предсказание
curl -X POST "http://localhost:8000/predict/emotion/batch" \
     -H "Content-Type: application/json" \
     -d '["Great news!", "This is terrible", "I am excited"]'
```

Этап 7: Создание тестового отчета

1. **Документирование результатов тестирования:**

   - Создайте файл `api_test_report.md` со следующей структурой:

```
# Отчет о тестировании Emotion Classification API

## Методология тестирования
- Инструмент: Swagger UI
- Версия API: 1.0.0
- Дата тестирования: [дата]

## Результаты тестирования эндпоинтов

### GET /
- Статус: ✅ Успешно
- Код ответа: 200
- Комментарии: Корректная информация о API

### GET /health
- Статус: ✅ Успешно
- Код ответа: 200
- Комментарии: Модель загружена, статус "healthy"

### POST /predict/emotion
- Позитивные тесты: ✅ Успешно
- Негативные тесты: ✅ Успешно
- Валидация данных: ✅ Работает корректно

### POST /predict/emotion/batch
- Позитивные тесты: ✅ Успешно
- Проверка лимитов: ✅ Успешно
- Формат ответа: ✅ Корректный

### GET /model/info
- Статус: ✅ Успешно
- Информация: ✅ Полная и корректная

## Общие выводы
- ✅ Все эндпоинты работают корректно
- ✅ Валидация данных функционирует properly
- ✅ Обработка ошибок реализована adequately
- ✅ API готово к использованию в production
```

2. **Сбор доказательств:**

   - Сделайте скриншоты успешных запросов в Swagger UI
   - Сохраните логи curl-запросов с временем ответа
   - Зафиксируйте примеры корректных и ошибочных ответов

Этап 8: Производительность и нагрузочное тестирование (опционально)

1. **Измерение времени ответа:**
    - Используйте Swagger UI для измерения времени выполнения запросов
    - Зафиксируйте среднее время ответа для разных эндпоинтов
2. **Базовое нагрузочное тестирование:**
    - Выполните серию последовательных запросов к API
    - Проверьте, не возникает ли утечек памяти или ошибок под нагрузкой
    - Используйте инструмент like `wrk` или `ab` для более сложного тестирования

Требования к оформлению и отчету (для Части 3)

Критерии оценки для Части 3:

- **Удовлетворительно:** Успешно выполнены Этапы 1-2 (базовое тестирование эндпоинтов). API отвечает на основные запросы.
- **Хорошо:** Дополнительно успешно выполнен Этап 3-4 (комплексное тестирование предсказаний, негативные сценарии). Создан базовый отчет о тестировании.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы Этапы 5-7: полное тестирование всех эндпоинтов, сравнение с curl, создание детализированного отчета. Дополнительно выполнено нагрузочное тестирование (Этап 8).

Рекомендуемая литература

1. **FastAPI Testing Guide:** https://fastapi.tiangolo.com/tutorial/testing/
2. **OpenAPI Specification:** https://swagger.io/specification/
3. **REST API Testing Methodology:** https://smartbear.com/learn/performance-monitoring/api-testing/
4. **Python Testing with Pytest:** https://docs.pytest.org/
5. **API Testing Best Practices:** https://blog.postman.com/api-testing-strategy/

Лабораторная работа №11-12, Часть 1: Построение прототипа RAG-системы

Цель работы: Освоить принципы построения RAG (Retrieval-Augmented Generation) систем путем интеграции векторного поиска и языковых моделей. Создать работающий прототип системы, способной находить релевантную информацию в базе знаний и генерировать осмысленные ответы на основе извлеченного контекста.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Окружение:** Conda (`mlops-lab`)
- **Векторная БД:** ChromaDB (из ЛР 5-6)
- **Языковая модель:** Hugging Face Transformers
- **Фреймворк:** LangChain (опционально)
- **Эмбеддинги:** sentence-transformers
- **Интерфейс:** FastAPI (из ЛР 9-10)

Теоретическая часть

1. Архитектура RAG-систем RAG (Retrieval-Augmented Generation) — это гибридный подход, сочетающий поиск информации и генерацию текста:

- **Retriever (Поисковый модуль):** Находит релевантные документы в базе знаний по запросу пользователя
- **Generator (Генеративный модуль):** Создает осмысленный ответ на основе найденных документов и исходного запроса

Преимущества RAG:

- **Актуальность:** Может работать с самыми свежими данными
- **Точность:** Снижает вероятность "галлюцинаций" модели
- **Объяснимость:** Можно отследить источники информации
- **Специализация:** Легко адаптируется к конкретной предметной области

2. Ключевые компоненты RAG-пайплайна

- **Загрузка данных:** Подготовка и обработка документов
- **Чанкинг:** Разбиение документов на семантически осмысленные фрагменты
- **Векторизация:** Преобразование текста в векторные представления
- **Поиск:** Нахождение наиболее релевантных чанков
- **Конструирование промта:** Формирование контекста для языковой модели
- **Генерация:** Создание финального ответа

3. Метрики оценки RAG-систем

- **Retrieval metrics:** Precision@K, Recall@K, NDCG

- **Generation metrics:** BLEU, ROUGE, Semantic similarity
- **End-to-end metrics:** Человеческая оценка, Task completion rate

---

Задание на практическую реализацию

Этап 1: Подготовка окружения и данных

1. **Установка необходимых пакетов:**

```
conda activate mlops-lab
pip install chromadb sentence-transformers transformers torch langchain
tiktoken
```

2. **Создание структуры проекта:**

```
mkdir rag-system
cd rag-system
mkdir data documents retriever generator api
touch main.py config.py
```

3. **Подготовка тестовых данных:**

   - Создайте файл с документами для базы знаний:

```
# documents/tech_docs.py
DOCUMENTS = [
    {
        "id": "doc_001",
        "title": "Машинное обучение",
        "content": "Машинное обучение — это область искусственного
интеллекта, которая использует статистические методы для создания моделей,
способных обучаться на данных и делать предсказания. Основные типы машинного
обучения включают обучение с учителем, без учителя и с подкреплением.",
        "category": "AI"
    },
    {
        "id": "doc_002",
        "title": "Глубокое обучение",
        "content": "Глубокое обучение использует нейронные сети с множеством
слоев для извлечения иерархических признаков из данных. Популярные
архитектуры включают сверточные нейронные сети для компьютерного зрения и
трансформеры для обработки естественного языка.",
        "category": "AI"
    },
    {
        "id": "doc_003",
        "title": "Трансформеры в NLP",
```

```
        "content": "Архитектура трансформеров revolutionized обработку
естественного языка. Модели типа BERT и GPT используют механизм внимания для
учета контекста во всем входном последовательности. BERT предназначен для
понимания текста, а GPT — для генерации.",
        "category": "NLP"
    },
    {
        "id": "doc_004",
        "title": "Векторные базы данных",
        "content": "Векторные базы данных оптимизированы для хранения и
поиска векторных представлений данных. Они используют алгоритмы
приближенного поиска ближайших соседей для эффективного семантического
поиска. ChromaDB — популярная open-source векторная БД.",
        "category": "Databases"
    },
    {
        "id": "doc_005",
        "title": "RAG-архитектура",
        "content": "RAG (Retrieval-Augmented Generation) сочетает поиск
информации в векторной базе данных с генерацией текста языковой моделью. Это
позволяет моделям работать с актуальными данными и снижает вероятность
галлюцинаций.",
        "category": "Architecture"
    }
]
```

Этап 2: Реализация модуля поиска (Retriever)

1. **Создание класса для работы с векторной БД:**

```python
# retriever/vector_store.py
import chromadb
from sentence_transformers import SentenceTransformer
from typing import List, Dict, Any
import logging

logger = logging.getLogger(__name__)

class VectorStore:
    def __init__(self, collection_name: str = "rag_documents"):
        self.client = chromadb.Client()
        self.collection_name = collection_name
        self.model = SentenceTransformer('all-MiniLM-L6-v2')

        try:
            self.collection = self.client.get_collection(collection_name)
            logger.info(f"Loaded existing collection: {collection_name}")
        except:
            self.collection = self.client.create_collection(
                name=collection_name,
                metadata={"hnsw:space": "cosine"}
```

```
            )
            logger.info(f"Created new collection: {collection_name}")

    def add_documents(self, documents: List[Dict[str, Any]]):
        """Добавление документов в векторное хранилище"""
        ids = [doc["id"] for doc in documents]
        texts = [doc["content"] for doc in documents]
        metadatas = [{
            "title": doc["title"],
            "category": doc["category"],
            "source": "tech_docs"
        } for doc in documents]

        # Генерация эмбеддингов
        embeddings = self.model.encode(texts).tolist()

        # Добавление в коллекцию
        self.collection.add(
            documents=texts,
            embeddings=embeddings,
            metadatas=metadatas,
            ids=ids
        )
        logger.info(f"Added {len(documents)} documents to collection")

    def search(self, query: str, n_results: int = 3) -> List[Dict[str,
Any]]:
        """Поиск релевантных документов"""
        query_embedding = self.model.encode([query]).tolist()

        results = self.collection.query(
            query_embeddings=query_embedding,
            n_results=n_results,
            include=["documents", "metadatas", "distances"]
        )

        # Форматирование результатов
        formatted_results = []
        for i, (doc, metadata, distance) in enumerate(zip(
            results['documents'][0],
            results['metadatas'][0],
            results['distances'][0]
        )):
            formatted_results.append({
                "content": doc,
                "metadata": metadata,
                "similarity_score": 1 - distance,  # Конвертируем расстояние
в схожесть
                "rank": i + 1
            })

        return formatted_results

    def get_collection_info(self) -> Dict[str, Any]:
```

```
        """Получение информации о коллекции"""
        return {
            "name": self.collection_name,
            "document_count": self.collection.count()
        }
```

Этап 3: Реализация модуля генерации (Generator)

1. **Создание класса для работы с языковой моделью:**

```
# generator/llm_client.py
from transformers import pipeline, AutoTokenizer, AutoModelForCausalLM
import torch
from typing import List, Dict, Any
import logging

logger = logging.getLogger(__name__)

class LLMGenerator:
    def __init__(self, model_name: str = "microsoft/DialoGPT-medium"):
        self.model_name = model_name
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModelForCausalLM.from_pretrained(model_name)
        self.tokenizer.pad_token = self.tokenizer.eos_token

        # Альтернативно, можно использовать pipeline
        self.generator = pipeline(
            "text-generation",
            model=model_name,
            tokenizer=model_name,
            torch_dtype=torch.float16,
            device_map="auto"
        )
        logger.info(f"Loaded language model: {model_name}")

    def generate_response(self, query: str, context: List[Dict[str, Any]]) -
> str:
        """Генерация ответа на основе запроса и контекста"""

        # Формирование промта с контекстом
        context_text = self._build_context_string(context)
        prompt = self._construct_prompt(query, context_text)

        try:
            # Генерация ответа
            response = self.generator(
                prompt,
                max_length=512,
                num_return_sequences=1,
                temperature=0.7,
                do_sample=True,
```

```
                pad_token_id=self.tokenizer.eos_token_id
            )

            generated_text = response[0]['generated_text']
            # Извлекаем только сгенерированную часть (после промта)
            answer = generated_text[len(prompt):].strip()

            return answer

        except Exception as e:
            logger.error(f"Generation error: {e}")
            return "Извините, произошла ошибка при генерации ответа."

    def _build_context_string(self, context: List[Dict[str, Any]]) -> str:
        """Построение строки контекста из найденных документов"""
        context_parts = []
        for i, doc in enumerate(context):
            content = doc['content']
            title = doc['metadata']['title']
            score = doc['similarity_score']
            context_parts.append(f"[Документ {i+1}] {title} (схожесть:
{score:.3f}): {content}")

        return "\n\n".join(context_parts)

    def _construct_prompt(self, query: str, context: str) -> str:
        """Конструирование промта для языковой модели"""
        prompt = f"""На основе предоставленного контекста, ответь на вопрос
пользователя. Если в контексте нет достаточной информации, скажи об этом.
```

Контекст: {context}

Вопрос: {query}

Ответ: """ return prompt ```

Этап 4: Интеграция компонентов в RAG-пайплайн

1. **Создание основного класса RAG-системы:**

```
# main.py
from retriever.vector_store import VectorStore
from generator.llm_client import LLMGenerator
from documents.tech_docs import DOCUMENTS
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class RAGSystem:
```

```
    def __init__(self):
        self.retriever = VectorStore()
        self.generator = LLMGenerator()
        self._initialize_database()

    def _initialize_database(self):
        """Инициализация базы данных с документами"""
        if self.retriever.get_collection_info()["document_count"] == 0:
            logger.info("Initializing database with documents...")
            self.retriever.add_documents(DOCUMENTS)
        else:
            logger.info("Database already initialized")

    def ask(self, question: str, n_documents: int = 3) -> Dict[str, Any]:
        """Основной метод для вопросов к RAG-системе"""
        logger.info(f"Processing question: {question}")

        # Шаг 1: Поиск релевантных документов
        retrieved_docs = self.retriever.search(question,
n_results=n_documents)
        logger.info(f"Retrieved {len(retrieved_docs)} documents")

        # Шаг 2: Генерация ответа на основе контекста
        answer = self.generator.generate_response(question, retrieved_docs)

        # Формирование полного ответа
        response = {
            "question": question,
            "answer": answer,
            "retrieved_documents": retrieved_docs,
            "document_count": len(retrieved_docs)
        }

        return response

    def get_system_info(self) -> Dict[str, Any]:
        """Получение информации о системе"""
        return {
            "retriever": self.retriever.get_collection_info(),
            "generator": {"model": self.generator.model_name},
            "status": "ready"
        }

# Пример использования
if __name__ == "__main__":
    rag_system = RAGSystem()

    # Тестовые вопросы
    test_questions = [
        "Что такое машинное обучение?",
        "Какие бывают типы машинного обучения?",
        "Как работают трансформеры в NLP?",
        "Что такое RAG-архитектура?"
    ]
```

```
    for question in test_questions:
        print(f"\n{'='*60}")
        print(f"Вопрос: {question}")
        response = rag_system.ask(question)
        print(f"Ответ: {response['answer']}")
        print(f"Найдено документов: {response['document_count']}")
        print(f"Лучший документ: {response['retrieved_documents'][0]
['metadata']['title']}")
```

Этап 5: Тестирование системы

1. **Запуск и тестирование RAG-системы:**

```
python main.py
```

2. **Анализ результатов:**
   - Проверьте, что система находит релевантные документы
   - Убедитесь, что генерируются осмысленные ответы
   - Проанализируйте качество поиска и генерации

3. **Создание тестового отчета:**

```
# test_rag_system.py
from main import RAGSystem

def test_rag_system():
    rag = RAGSystem()

    test_cases = [
        {
            "question": "Что такое машинное обучение?",
            "expected_keywords": ["искусственный интеллект", "статистические
методы", "предсказания"]
        },
        {
            "question": "Какие нейронные сети используются в глубоком
обучении?",
            "expected_keywords": ["сверточные", "трансформеры", "слои"]
        }
    ]

    print("Тестирование RAG-системы:")
    print("=" * 50)

    for i, test_case in enumerate(test_cases, 1):
        print(f"\nТест {i}: {test_case['question']}")
        response = rag.ask(test_case['question'])
```

```
        print(f"Ответ: {response['answer']}")
        print(f"Найдено документов: {response['document_count']}")

        # Проверка ключевых слов
        answer_lower = response['answer'].lower()
        found_keywords = [kw for kw in test_case['expected_keywords'] if kw
in answer_lower]
        print(f"Найдено ключевых слов:
{len(found_keywords)}/{len(test_case['expected_keywords'])}")
        print(f"Ключевые слова: {found_keywords}")

if __name__ == "__main__":
    test_rag_system()
```

Требования к оформлению и отчету (для Части 1)

Критерии оценки для Части 1:

- **Удовлетворительно:** Успешно выполнены Этапы 1-2 (подготовка данных, реализация модуля поиска). Система находит релевантные документы по запросам.
- **Хорошо:** Дополнительно успешно выполнен Этап 3 (реализация модуля генерации). Система генерирует ответы на основе найденных документов.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы Этапы 4-5: полная интеграция системы, тестирование на различных запросах, анализ качества работы. Создан детализированный отчет о работе системы.

Рекомендуемая литература

1. **RAG Original Paper:** https://arxiv.org/abs/2005.11401
2. **LangChain Documentation:** https://python.langchain.com/
3. **Hugging Face Transformers:** https://huggingface.co/docs/transformers
4. **ChromaDB Documentation:** https://docs.trychroma.com/
5. **Sentence Transformers:** https://www.sbert.net/

Лабораторная работа №11-12, Часть 2: Настройка языковой модели в качестве генератора

Цель работы: Освоить продвинутые техники настройки и оптимизации языковых моделей для использования в качестве генеративного компонента RAG-системы. Получить практические навыки работы с различными архитектурами моделей, оптимизации промптинга и оценки качества генерации.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Окружение:** Conda (`mlops-lab`)
- **Языковые модели:** Hugging Face Transformers (GPT-2, T5, BART, Falcon)
- **Оптимизация:** Quantization, LoRA, PEFT
- **Фреймворки:** Transformers, Accelerate, BitsAndBytes
- **Метрики:** ROUGE, BLEU, Perplexity

Теоретическая часть

1. Архитектуры языковых моделей для генерации Различные архитектуры подходят для разных сценариев использования в RAG:

- **Авторегрессивные модели (GPT-2, GPT-Neo):** Идеальны для генерации последовательного текста
- **Seq2Seq модели (T5, BART):** Хороши для задач переформулирования и суммирования
- **Инструктивные модели (Alpaca, Vicuna):** Специализированы для следования инструкциям
- **Многомодальные модели:** Могут работать с различными типами данных

2. Методы оптимизации для production

- **Квантование (Quantization):** Уменьшение точности весов для экономии памяти
- **Дистилляция (Knowledge Distillation):** Обучение меньшей модели на выходах большой
- **PEFT/LoRA:** Эффективная тонкая настройка с минимальными параметрами
- **Градиентный чекпоинтинг:** Экономия памяти при обучении

3. Критерии выбора модели для RAG

- **Качество генерации:** Способность создавать связные и релевантные ответы
- **Скорость инференса:** Время отклика для реального использования
- **Потребление памяти:** Совместимость с доступными ресурсами
- **Контекстное окно:** Способность обрабатывать длинные промпты с контекстом

Задание на практическую реализацию

Этап 1: Сравнение различных моделей-генераторов

1. **Создание тестового стенда для сравнения моделей:**

```python
# generator/model_comparison.py
from transformers import (
    pipeline,
    AutoTokenizer,
    AutoModelForCausalLM,
    AutoModelForSeq2SeqLM,
    GenerationConfig
)
import torch
from typing import List, Dict, Any
import time
import logging
from datetime import datetime

logger = logging.getLogger(__name__)

class ModelComparator:
    def __init__(self):
        self.models_config = {
            "gpt2-medium": {
                "type": "causal",
                "description": "Авторегрессивная модель среднего размера"
            },
            "t5-small": {
                "type": "seq2seq",
                "description": "Seq2Seq модель для переформулирования"
            },
            "facebook/bart-base": {
                "type": "seq2seq",
                "description": "BART модель для текстовых задач"
            },
            "microsoft/DialoGPT-medium": {
                "type": "causal",
                "description": "Диалоговая модель на основе GPT-2"
            }
        }

        self.loaded_models = {}

    def load_model(self, model_name: str):
        """Загрузка модели с обработкой ошибок"""
        try:
            logger.info(f"Loading model: {model_name}")
            start_time = time.time()

            if self.models_config[model_name]["type"] == "causal":
                tokenizer = AutoTokenizer.from_pretrained(model_name)
                model = AutoModelForCausalLM.from_pretrained(
                    model_name,
                    torch_dtype=torch.float16,
                    device_map="auto"
                )
```

```
                tokenizer.pad_token = tokenizer.eos_token

            else:  # seq2seq
                tokenizer = AutoTokenizer.from_pretrained(model_name)
                model = AutoModelForSeq2SeqLM.from_pretrained(
                    model_name,
                    torch_dtype=torch.float16,
                    device_map="auto"
                )

            load_time = time.time() - start_time

            self.loaded_models[model_name] = {
                "model": model,
                "tokenizer": tokenizer,
                "type": self.models_config[model_name]["type"],
                "load_time": load_time
            }

            logger.info(f"Successfully loaded {model_name} in
{load_time:.2f}s")

        except Exception as e:
            logger.error(f"Failed to load {model_name}: {e}")

    def generate_with_model(self, model_name: str, prompt: str,
                            max_length: int = 200) -> Dict[str, Any]:
        """Генерация текста с указанной моделью"""
        if model_name not in self.loaded_models:
            self.load_model(model_name)

        model_info = self.loaded_models[model_name]
        tokenizer = model_info["tokenizer"]
        model = model_info["model"]

        start_time = time.time()

        try:
            if model_info["type"] == "causal":
                inputs = tokenizer(prompt, return_tensors="pt",
truncation=True, max_length=512)

                with torch.no_grad():
                    outputs = model.generate(
                        **inputs,
                        max_length=max_length,
                        num_return_sequences=1,
                        temperature=0.7,
                        do_sample=True,
                        pad_token_id=tokenizer.eos_token_id,
                        repetition_penalty=1.1
                    )

                generated_text = tokenizer.decode(outputs[0],
```

```
skip_special_tokens=True)
                # Удаляем промт из сгенерированного текста
                answer = generated_text[len(prompt):].strip()

            else:  # seq2seq
                inputs = tokenizer(prompt, return_tensors="pt",
truncation=True, max_length=512)

                with torch.no_grad():
                    outputs = model.generate(
                        **inputs,
                        max_length=max_length,
                        num_return_sequences=1,
                        temperature=0.7,
                        do_sample=True
                    )

                answer = tokenizer.decode(outputs[0],
skip_special_tokens=True)

            generation_time = time.time() - start_time

            return {
                "model": model_name,
                "answer": answer,
                "generation_time": generation_time,
                "answer_length": len(answer),
                "success": True
            }

        except Exception as e:
            logger.error(f"Generation failed for {model_name}: {e}")
            return {
                "model": model_name,
                "answer": f"Error: {str(e)}",
                "generation_time": 0,
                "answer_length": 0,
                "success": False
            }
```

Этап 2: Оптимизация генерации с помощью продвинутых техник

1. **Реализация оптимизированного генератора:**

```
# generator/optimized_generator.py
from transformers import (
    AutoTokenizer,
    AutoModelForCausalLM,
    BitsAndBytesConfig,
    GenerationConfig
)
```

```python
import torch
from typing import List, Dict, Any
import time
import logging

logger = logging.getLogger(__name__)

class OptimizedLLMGenerator:
    def __init__(self, model_name: str = "microsoft/DialoGPT-medium"):
        self.model_name = model_name
        self.quantization_config = None
        self.generation_config = None
        self._setup_quantization()
        self._setup_generation_config()
        self._load_model()

    def _setup_quantization(self):
        """Настройка квантования для экономии памяти"""
        self.quantization_config = BitsAndBytesConfig(
            load_in_4bit=True,
            bnb_4bit_use_double_quant=True,
            bnb_4bit_quant_type="nf4",
            bnb_4bit_compute_dtype=torch.float16
        )

    def _setup_generation_config(self):
        """Настройка параметров генерации"""
        self.generation_config = GenerationConfig(
            max_new_tokens=150,
            temperature=0.7,
            do_sample=True,
            top_p=0.9,
            top_k=50,
            repetition_penalty=1.1,
            pad_token_id=50256  # EOS token for most models
        )

    def _load_model(self):
        """Загрузка модели с оптимизациями"""
        logger.info(f"Loading optimized model: {self.model_name}")

        self.tokenizer = AutoTokenizer.from_pretrained(self.model_name)
        self.tokenizer.pad_token = self.tokenizer.eos_token

        self.model = AutoModelForCausalLM.from_pretrained(
            self.model_name,
            quantization_config=self.quantization_config,
            device_map="auto",
            torch_dtype=torch.float16,
            trust_remote_code=True
        )

        logger.info("Model loaded successfully with optimizations")
```

```
    def generate_optimized_response(self, query: str, context:
List[Dict[str, Any]]) -> Dict[str, Any]:
        """Оптимизированная генерация ответа"""
        start_time = time.time()

        # Построение улучшенного промта
        prompt = self._construct_enhanced_prompt(query, context)

        try:
            # Токенизация с оптимизацией
            inputs = self.tokenizer(
                prompt,
                return_tensors="pt",
                truncation=True,
                max_length=1024
            )

            # Генерация с кастомизированными параметрами
            with torch.no_grad():
                outputs = self.model.generate(
                    **inputs,
                    generation_config=self.generation_config
                )

            # Декодирование с пропуском специальных токенов
            generated_text = self.tokenizer.decode(
                outputs[0],
                skip_special_tokens=True
            )

            # Извлечение ответа
            answer = generated_text[len(prompt):].strip()

            generation_time = time.time() - start_time

            return {
                "answer": answer,
                "generation_time": generation_time,
                "prompt_length": len(prompt),
                "answer_length": len(answer),
                "model": self.model_name,
                "optimized": True
            }

        except Exception as e:
            logger.error(f"Optimized generation failed: {e}")
            return {
                "answer": f"Generation error: {str(e)}",
                "generation_time": 0,
                "prompt_length": 0,
                "answer_length": 0,
                "model": self.model_name,
                "optimized": False
            }
```

```
    def _construct_enhanced_prompt(self, query: str, context: List[Dict[str,
Any]]) -> str:
        """Улучшенное конструирование промта"""
        context_text = self._build_structured_context(context)

        enhanced_prompt = f"""Ты - AI-ассистент, который отвечает на вопросы
на основе предоставленного контекста.
```

ИНСТРУКЦИИ:

1. Используй только информацию из предоставленного контекста
2. Если в контексте нет ответа, честно скажи об этом
3. Будь точным и информативным
4. Отвечай на русском языке

КОНТЕКСТ: {context_text}

ВОПРОС: {query}

ОТВЕТ: """

```
        return enhanced_prompt

    def _build_structured_context(self, context: List[Dict[str, Any]]) -> str:
        """Структурированное построение контекста"""
        context_parts = []

        for i, doc in enumerate(context, 1):
            content = doc['content']
            title = doc['metadata']['title']
            category = doc['metadata']['category']
            score = doc['similarity_score']

            context_parts.append(
                f"Документ {i}:\n"
                f"Заголовок: {title}\n"
                f"Категория: {category}\n"
                f"Релевантность: {score:.3f}\n"
                f"Содержание: {content}\n"
            )

        return "\n" + "="*50 + "\n".join(context_parts) + "="*50
```

Этап 3: Сравнительный анализ моделей

1. **Создание системы бенчмаркинга:**

```
# generator/benchmark_system.py
import pandas as pd
import time
from typing import List, Dict, Any
from .model_comparison import ModelComparator
from .optimized_generator import OptimizedLLMGenerator

class ModelBenchmark:
    def __init__(self):
        self.comparator = ModelComparator()
        self.test_questions = [
            {
                "question": "Что такое машинное обучение?",
                "context": [{
                    "content": "Машинное обучение — это область
искусственного интеллекта, которая использует статистические методы для
создания моделей, способных обучаться на данных и делать предсказания.",
                    "metadata": {"title": "Машинное обучение", "category":
"AI"},
                    "similarity_score": 0.95
                }]
            },
            {
                "question": "Какие типы нейронных сетей вы знаете?",
                "context": [{
                    "content": "Популярные архитектуры нейронных сетей
включают сверточные нейронные сети для компьютерного зрения и трансформеры
для обработки естественного языка.",
                    "metadata": {"title": "Глубокое обучение", "category":
"AI"},
                    "similarity_score": 0.88
                }]
            }
        ]

    def run_benchmark(self, model_names: List[str]) -> pd.DataFrame:
        """Запуск сравнительного тестирования моделей"""
        results = []

        for model_name in model_names:
            logger.info(f"Benchmarking model: {model_name}")

            # Загрузка модели
            self.comparator.load_model(model_name)

            for test_case in self.test_questions:
                question = test_case["question"]
                context = test_case["context"]

                # Генерация ответа
                result = self.comparator.generate_with_model(model_name,
question)
```

```
                # Оценка качества
                evaluation = self._evaluate_response(
                    result["answer"],
                    question,
                    context
                )

                benchmark_result = {
                    "model": model_name,
                    "question": question,
                    "answer": result["answer"],
                    "generation_time": result["generation_time"],
                    "answer_length": result["answer_length"],
                    "success": result["success"]
                }
                benchmark_result.update(evaluation)

                results.append(benchmark_result)

                # Пауза между запросами
                time.sleep(1)

        return pd.DataFrame(results)

    def _evaluate_response(self, answer: str, question: str, context:
List[Dict]) -> Dict[str, Any]:
        """Базовая оценка качества ответа"""
        # Простые метрики для демонстрации
        context_keywords = self._extract_keywords_from_context(context)
        answer_keywords = self._extract_keywords(answer)

        # Вычисление покрытия ключевых слов
        matched_keywords = set(context_keywords) & set(answer_keywords)
        keyword_coverage = len(matched_keywords) / len(context_keywords) if
context_keywords else 0

        return {
            "keyword_coverage": keyword_coverage,
            "matched_keywords_count": len(matched_keywords),
            "answer_has_content": len(answer.strip()) > 10,
            "contains_uncertainty": "не знаю" in answer.lower() or "нет
информации" in answer.lower()
        }

    def _extract_keywords_from_context(self, context: List[Dict]) ->
List[str]:
        """Извлечение ключевых слов из контекста"""
        all_text = " ".join([doc["content"] for doc in context])
        # Простая токенизация для демонстрации
        words = all_text.lower().split()
        # Фильтрация стоп-слов и коротких слов
        stop_words = {"и", "в", "на", "с", "по", "для", "это", "что", "как"}
        keywords = [word for word in words if len(word) > 3 and word not in
```

```
stop_words]
        return list(set(keywords))[:10]  # Возвращаем топ-10 уникальных
ключевых слов
```

Этап 4: Интеграция оптимизированного генератора в RAG-систему

1. **Модификация основной RAG-системы:**

```
# main_optimized.py
from retriever.vector_store import VectorStore
from generator.optimized_generator import OptimizedLLMGenerator
from documents.tech_docs import DOCUMENTS
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class OptimizedRAGSystem:
    def __init__(self, model_name: str = "microsoft/DialoGPT-medium"):
        self.retriever = VectorStore()
        self.generator = OptimizedLLMGenerator(model_name)
        self._initialize_database()

    def _initialize_database(self):
        """Инициализация базы данных"""
        if self.retriever.get_collection_info()["document_count"] == 0:
            logger.info("Initializing database with documents...")
            self.retriever.add_documents(DOCUMENTS)

    def ask(self, question: str, n_documents: int = 3) -> Dict[str, Any]:
        """Оптимизированный метод для вопросов"""
        logger.info(f"Processing question: {question}")

        # Поиск документов
        retrieved_docs = self.retriever.search(question,
n_results=n_documents)

        # Генерация ответа с оптимизированной моделью
        generation_result = self.generator.generate_optimized_response(
            question,
            retrieved_docs
        )

        response = {
            "question": question,
            "answer": generation_result["answer"],
            "retrieved_documents": retrieved_docs,
            "generation_info": {
                "model": generation_result["model"],
                "generation_time": generation_result["generation_time"],
                "optimized": generation_result["optimized"]
```

```
            },
            "document_count": len(retrieved_docs)
        }

        return response

# Демонстрация работы оптимизированной системы
if __name__ == "__main__":
    # Сравнение производительности
    import time

    rag_standard = OptimizedRAGSystem("microsoft/DialoGPT-medium")

    test_questions = [
        "Объясни что такое машинное обучение",
        "Какие архитектуры нейронных сетей используются в NLP?",
        "Что такое векторные базы данных и для чего они нужны?"
    ]

    print("Тестирование оптимизированной RAG-системы:")
    print("=" * 60)

    for question in test_questions:
        start_time = time.time()
        response = rag_standard.ask(question)
        total_time = time.time() - start_time

        print(f"\nВопрос: {question}")
        print(f"Ответ: {response['answer']}")
        print(f"Общее время: {total_time:.2f}c")
        print(f"Время генерации: {response['generation_info']
['generation_time']:.2f}c")
        print(f"Модель: {response['generation_info']['model']}")
        print(f"Найдено документов: {response['document_count']}")
```

Этап 5: Создание отчета о настройке генератора

1. **Генерация сравнительного отчета:**

```
# generate_report.py
import pandas as pd
from generator.benchmark_system import ModelBenchmark

def create_model_comparison_report():
    """Создание отчета о сравнении моделей"""
    benchmark = ModelBenchmark()

    models_to_test = [
        "gpt2-medium",
        "t5-small",
        "facebook/bart-base",
```

```
            "microsoft/DialoGPT-medium"
        ]

        results_df = benchmark.run_benchmark(models_to_test)

        # Агрегация результатов
        summary = results_df.groupby('model').agg({
            'generation_time': 'mean',
            'keyword_coverage': 'mean',
            'success': 'mean',
            'answer_length': 'mean'
        }).round(3)

        summary = summary.rename(columns={
            'generation_time': 'avg_generation_time',
            'keyword_coverage': 'avg_keyword_coverage',
            'success': 'success_rate',
            'answer_length': 'avg_answer_length'
        })

        # Сохранение отчетов
        results_df.to_csv("model_comparison_detailed.csv", index=False)
        summary.to_csv("model_comparison_summary.csv")

        print("Детальный отчет сохранен в: model_comparison_detailed.csv")
        print("Сводный отчет сохранен в: model_comparison_summary.csv")

        return summary

    if __name__ == "__main__":
        report = create_model_comparison_report()
        print("\nСводный отчет по моделям:")
        print(report)
```

Требования к оформлению и отчету (для Части 2)

Критерии оценки для Части 2:

- **Удовлетворительно:** Успешно выполнены Этапы 1-2 (сравнение моделей, базовая оптимизация). Система генерирует ответы с разными моделями.
- **Хорошо:** Дополнительно успешно выполнен Этап 3 (сравнительный анализ). Создан бенчмаркинг систем и оценка качества генерации.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы Этапы 4-5: полная интеграция оптимизированного генератора, создание детализированных отчетов сравнения. Проведен анализ производительности и качества разных моделей.

Рекомендуемая литература

1. **Hugging Face Generation:** https://huggingface.co/docs/transformers/generation_strategies

2. **Model Quantization:** https://huggingface.co/docs/transformers/quantization
3. **Text Generation Metrics:** https://huggingface.co/spaces/evaluate-metric/rouge
4. **Optimization Techniques:** https://huggingface.co/docs/optimum/index
5. **Prompt Engineering Guide:** https://www.promptingguide.ai/

Лабораторная работа №11-12, Часть 3: Создание конвейера RAG-системы

Цель работы: Реализовать полный конвейер RAG-системы, интегрирующий семантический поиск и генерацию ответов. Освоить проектирование надежных пайплайнов, обработку ошибок, мониторинг производительности и оптимизацию взаимодействия между компонентами системы.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Окружение:** Conda (`mlops-lab`)
- **Компоненты:** ChromaDB, Hugging Face Transformers, FastAPI
- **Мониторинг:** Prometheus (опционально), логирование
- **Оптимизация:** Кэширование, батчинг, асинхронная обработка

Теоретическая часть

1. Архитектура полного RAG-конвейера Производственный RAG-пайплайн состоит из взаимосвязанных этапов:

- **Прием запроса:** Валидация и предобработка входного вопроса
- **Семантический поиск:** Поиск релевантных документов в векторной БД
- **Ранжирование и фильтрация:** Отбор наиболее релевантных фрагментов
- **Конструирование промта:** Формирование оптимального контекста для модели
- **Генерация ответа:** Создание финального ответа языковой моделью
- **Постобработка:** Форматирование и валидация выходных данных

2. Критические аспекты production-систем

- **Обработка ошибок:** Graceful degradation при сбоях компонентов
- **Таймауты:** Контроль времени выполнения каждого этапа
- **Кэширование:** Уменьшение нагрузки на повторяющиеся запросы
- **Мониторинг:** Отслеживание метрик производительности и качества
- **Масштабируемость:** Поддержка увеличения нагрузки

3. Стратегии оптимизации производительности

- **Асинхронная обработка:** Параллельное выполнение независимых операций
- **Пакетная обработка:** Группировка запросов для эффективного использования ресурсов
- **Предварительная загрузка:** Кэширование часто используемых данных
- **Балансировка нагрузки:** Распределение запросов между несколькими экземплярами

Задание на практическую реализацию

Этап 1: Проектирование архитектуры конвейера

1. **Создание конфигурации системы:**

```
# config/pipeline_config.py
from typing import Dict, Any
from dataclasses import dataclass

@dataclass
class RetrieverConfig:
    collection_name: str = "rag_documents"
    top_k: int = 5
    similarity_threshold: float = 0.6
    max_context_length: int = 2000

@dataclass
class GeneratorConfig:
    model_name: str = "microsoft/DialoGPT-medium"
    max_tokens: int = 250
    temperature: float = 0.7
    timeout_seconds: int = 30

@dataclass
class PipelineConfig:
    retriever: RetrieverConfig = RetrieverConfig()
    generator: GeneratorConfig = GeneratorConfig()
    enable_caching: bool = True
    cache_ttl: int = 3600  # 1 hour
    max_retries: int = 3
    request_timeout: int = 60

# Конфигурация по умолчанию
DEFAULT_CONFIG = PipelineConfig()
```

Этап 2: Реализация основного конвейера

1. **Создание класса RAG-конвейера:**

```
# pipeline/rag_pipeline.py
import logging
import time
import asyncio
from typing import Dict, Any, List, Optional
from datetime import datetime
import hashlib
import json

from retriever.vector_store import VectorStore
from generator.optimized_generator import OptimizedLLMGenerator
from config.pipeline_config import PipelineConfig

logger = logging.getLogger(__name__)
```

```
class RAGPipeline:
    def __init__(self, config: PipelineConfig = None):
        self.config = config or PipelineConfig()
        self.retriever = VectorStore(self.config.retriever.collection_name)
        self.generator =
OptimizedLLMGenerator(self.config.generator.model_name)
        self.cache = {}  # Простой in-memory кэш для демонстрации
        self.metrics = {
            "total_requests": 0,
            "successful_requests": 0,
            "average_processing_time": 0,
            "cache_hits": 0
        }

    async def process_question(self, question: str, user_context: Dict[str,
Any] = None) -> Dict[str, Any]:
        """Основной метод обработки вопроса через RAG-конвейер"""
        start_time = time.time()
        request_id = self._generate_request_id(question, user_context)

        logger.info(f"Processing request {request_id}: {question}")
        self.metrics["total_requests"] += 1

        try:
            # Шаг 1: Проверка кэша
            cached_result = self._get_cached_result(request_id)
            if cached_result:
                logger.info(f"Cache hit for request {request_id}")
                self.metrics["cache_hits"] += 1
                cached_result["cached"] = True
                return cached_result

            # Шаг 2: Семантический поиск
            retrieval_start = time.time()
            retrieved_docs = await self._retrieve_documents(question)
            retrieval_time = time.time() - retrieval_start

            # Шаг 3: Фильтрация и ранжирование
            filtered_docs = self._filter_documents(retrieved_docs)

            # Шаг 4: Генерация ответа
            generation_start = time.time()
            answer = await self._generate_answer(question, filtered_docs,
user_context)
            generation_time = time.time() - generation_start

            # Шаг 5: Постобработка
            final_answer = self._postprocess_answer(answer, filtered_docs)

            # Формирование результата
            processing_time = time.time() - start_time
            result = self._build_response(
                question=question,
```

```
                answer=final_answer,
                documents=filtered_docs,
                processing_time=processing_time,
                retrieval_time=retrieval_time,
                generation_time=generation_time,
                request_id=request_id
            )

            # Кэширование результата
            self._cache_result(request_id, result)

            self.metrics["successful_requests"] += 1
            self._update_metrics(processing_time)

            logger.info(f"Request {request_id} completed in
{processing_time:.2f}s")
            return result

        except Exception as e:
            processing_time = time.time() - start_time
            error_result = self._build_error_response(question, str(e),
processing_time, request_id)
            logger.error(f"Request {request_id} failed: {e}")
            return error_result

    async def _retrieve_documents(self, question: str) -> List[Dict[str,
Any]]:
        """Асинхронный поиск документов с таймаутом"""
        try:
            # Используем asyncio для неблокирующего выполнения
            loop = asyncio.get_event_loop()
            documents = await asyncio.wait_for(
                loop.run_in_executor(None, self.retriever.search, question,
self.config.retriever.top_k),
                timeout=self.config.generator.timeout_seconds
            )
            return documents
        except asyncio.TimeoutError:
            logger.warning("Document retrieval timeout")
            return []
        except Exception as e:
            logger.error(f"Retrieval error: {e}")
            return []

    async def _generate_answer(self, question: str, documents:
List[Dict[str, Any]],
                               user_context: Dict[str, Any]) -> str:
        """Асинхронная генерация ответа с обработкой ошибок"""
        try:
            if not documents:
                return "К сожалению, я не нашел достаточно информации для
ответа на этот вопрос."

            loop = asyncio.get_event_loop()
```

```
            generation_result = await asyncio.wait_for(
                loop.run_in_executor(
                    None,
                    self.generator.generate_optimized_response,
                    question, documents
                ),
                timeout=self.config.generator.timeout_seconds
            )

            return generation_result["answer"]

        except asyncio.TimeoutError:
            logger.warning("Answer generation timeout")
            return "Извините, генерация ответа заняла слишком много времени.
Попробуйте переформулировать вопрос."
        except Exception as e:
            logger.error(f"Generation error: {e}")
            return f"Произошла ошибка при генерации ответа: {str(e)}"

    def _filter_documents(self, documents: List[Dict[str, Any]]) ->
List[Dict[str, Any]]:
        """Фильтрация и ранжирование документов"""
        # Фильтрация по порогу схожести
        filtered = [
            doc for doc in documents
            if doc.get('similarity_score', 0) >=
self.config.retriever.similarity_threshold
        ]

        # Сортировка по релевантности
        filtered.sort(key=lambda x: x.get('similarity_score', 0),
reverse=True)

        # Ограничение длины контекста
        total_length = 0
        final_documents = []

        for doc in filtered:
            doc_length = len(doc.get('content', ''))
            if total_length + doc_length <=
self.config.retriever.max_context_length:
                final_documents.append(doc)
                total_length += doc_length
            else:
                break

        return final_documents

    def _postprocess_answer(self, answer: str, documents: List[Dict[str,
Any]]) -> str:
        """Постобработка сгенерированного ответа"""
        # Удаление лишних пробелов и переносов
        answer = ' '.join(answer.split())
```

```
        # Проверка минимальной длины ответа
        if len(answer.strip()) < 10:
            return "Извините, не удалось сгенерировать содержательный ответ
на основе найденной информации."

        return answer

    def _build_response(self, **kwargs) -> Dict[str, Any]:
        """Формирование структурированного ответа"""
        return {
            "success": True,
            "timestamp": datetime.now().isoformat(),
            **kwargs
        }

    def _build_error_response(self, question: str, error: str,
                              processing_time: float, request_id: str) ->
Dict[str, Any]:
        """Формирование ответа об ошибке"""
        return {
            "success": False,
            "question": question,
            "answer": f"Произошла ошибка: {error}",
            "processing_time": processing_time,
            "request_id": request_id,
            "timestamp": datetime.now().isoformat(),
            "documents": []
        }

    def _generate_request_id(self, question: str, user_context: Dict[str,
Any]) -> str:
        """Генерация уникального ID запроса"""
        content = question + json.dumps(user_context or {}, sort_keys=True)
        return hashlib.md5(content.encode()).hexdigest()[:10]

    def _get_cached_result(self, request_id: str) -> Optional[Dict[str,
Any]]:
        """Получение результата из кэша"""
        if not self.config.enable_caching:
            return None

        cached = self.cache.get(request_id)
        if cached and time.time() - cached['timestamp'] <
self.config.cache_ttl:
            return cached['result']
        return None

    def _cache_result(self, request_id: str, result: Dict[str, Any]):
        """Сохранение результата в кэш"""
        if self.config.enable_caching:
            self.cache[request_id] = {
                'result': result,
                'timestamp': time.time()
            }
```

```
            # Очистка устаревших записей (простая реализация)
            if len(self.cache) > 1000:  # Максимум 1000 записей в кэше
                oldest_key = min(self.cache.keys(),
                                 key=lambda k: self.cache[k]['timestamp'])
                del self.cache[oldest_key]

    def _update_metrics(self, processing_time: float):
        """Обновление метрик производительности"""
        total_time = self.metrics["average_processing_time"] *
(self.metrics["successful_requests"] - 1)
        self.metrics["average_processing_time"] = (total_time +
processing_time) / self.metrics["successful_requests"]

    def get_metrics(self) -> Dict[str, Any]:
        """Получение текущих метрик системы"""
        return self.metrics.copy()
```

Этап 3: Создание FastAPI-сервиса для конвейера

1. **Реализация API-эндпоинтов:**

```
# api/pipeline_service.py
from fastapi import FastAPI, HTTPException, BackgroundTasks
from pydantic import BaseModel, Field
from typing import Optional, Dict, Any, List
import logging
import uvicorn

from pipeline.rag_pipeline import RAGPipeline
from config.pipeline_config import PipelineConfig

# Модели данных для API
class QuestionRequest(BaseModel):
    question: str = Field(..., min_length=1, max_length=1000,
description="Вопрос для системы")
    user_context: Optional[Dict[str, Any]] = Field(None,
description="Дополнительный контекст пользователя")
    use_cache: bool = Field(True, description="Использовать кэширование")

class PipelineResponse(BaseModel):
    success: bool = Field(..., description="Успешность выполнения запроса")
    question: str = Field(..., description="Исходный вопрос")
    answer: str = Field(..., description="Сгенерированный ответ")
    documents: List[Dict[str, Any]] = Field(..., description="Найденные
документы")
    processing_time: float = Field(..., description="Общее время обработки")
    retrieval_time: float = Field(..., description="Время поиска
документов")
    generation_time: float = Field(..., description="Время генерации
ответа")
    request_id: str = Field(..., description="ID запроса")
```

```
    timestamp: str = Field(..., description="Временная метка")
    cached: bool = Field(False, description="Результат из кэша")

class MetricsResponse(BaseModel):
    total_requests: int = Field(..., description="Общее количество
запросов")
    successful_requests: int = Field(..., description="Успешные запросы")
    average_processing_time: float = Field(..., description="Среднее время
обработки")
    cache_hits: int = Field(..., description="Попадания в кэш")
    cache_hit_rate: float = Field(..., description="Процент попаданий в
кэш")

# Создание приложения FastAPI
app = FastAPI(
    title="RAG Pipeline API",
    description="API для интеллектуальной системы вопрос-ответ на основе
RAG",
    version="1.0.0"
)

# Глобальные объекты
rag_pipeline = RAGPipeline()

@app.post("/ask", response_model=PipelineResponse, tags=["RAG Pipeline"])
async def ask_question(request: QuestionRequest, background_tasks:
BackgroundTasks):
    """
    Основной эндпоинт для вопросов к RAG-системе

    - **question**: Текст вопроса (1-1000 символов)
    - **user_context**: Дополнительный контекст (опционально)
    - **use_cache**: Использовать кэширование
    """
    try:
        # Временное отключение кэширования если нужно
        original_cache_setting = rag_pipeline.config.enable_caching
        if not request.use_cache:
            rag_pipeline.config.enable_caching = False

        result = await rag_pipeline.process_question(
            question=request.question,
            user_context=request.user_context
        )

        # Восстановление настроек кэширования
        rag_pipeline.config.enable_caching = original_cache_setting

        return result

    except Exception as e:
        logging.error(f"API error: {e}")
        raise HTTPException(status_code=500, detail=str(e))
```

```python
@app.get("/metrics", response_model=MetricsResponse, tags=["Monitoring"])
async def get_metrics():
    """Получение метрик производительности системы"""
    metrics = rag_pipeline.get_metrics()

    # Расчет процента попаданий в кэш
    cache_hit_rate = 0
    if metrics["total_requests"] > 0:
        cache_hit_rate = metrics["cache_hits"] / metrics["total_requests"]

    return MetricsResponse(
        total_requests=metrics["total_requests"],
        successful_requests=metrics["successful_requests"],
        average_processing_time=metrics["average_processing_time"],
        cache_hits=metrics["cache_hits"],
        cache_hit_rate=cache_hit_rate
    )

@app.get("/health", tags=["Monitoring"])
async def health_check():
    """Проверка здоровья системы"""
    return {
        "status": "healthy",
        "timestamp": datetime.now().isoformat(),
        "pipeline_ready": True
    }

@app.get("/config", tags=["System"])
async def get_config():
    """Получение текущей конфигурации системы"""
    return {
        "retriever": {
            "collection_name":
rag_pipeline.config.retriever.collection_name,
            "top_k": rag_pipeline.config.retriever.top_k,
            "similarity_threshold":
rag_pipeline.config.retriever.similarity_threshold
        },
        "generator": {
            "model_name": rag_pipeline.config.generator.model_name,
            "max_tokens": rag_pipeline.config.generator.max_tokens
        },
        "pipeline": {
            "enable_caching": rag_pipeline.config.enable_caching,
            "cache_ttl": rag_pipeline.config.cache_ttl
        }
    }

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Этап 4: Тестирование и валидация конвейера

1. **Создание комплексного теста:**

```
# tests/test_pipeline.py
import asyncio
import pytest
from pipeline.rag_pipeline import RAGPipeline
from config.pipeline_config import PipelineConfig

class TestRAGPipeline:
    def setup_method(self):
        self.config = PipelineConfig(
            retriever=RetrieverConfig(top_k=3, similarity_threshold=0.3),
            generator=GeneratorConfig(timeout_seconds=10),
            enable_caching=False
        )
        self.pipeline = RAGPipeline(self.config)

    @pytest.mark.asyncio
    async def test_pipeline_success(self):
        """Тест успешного выполнения конвейера"""
        result = await self.pipeline.process_question("Что такое машинное
обучение?")

        assert result["success"] == True
        assert "answer" in result
        assert "documents" in result
        assert result["processing_time"] > 0
        assert len(result["documents"]) > 0

    @pytest.mark.asyncio
    async def test_pipeline_empty_question(self):
        """Тест обработки пустого вопроса"""
        result = await self.pipeline.process_question("")

        assert result["success"] == False
        assert "error" in result["answer"].lower()

    @pytest.mark.asyncio
    async def test_pipeline_unknown_topic(self):
        """Тест вопроса по неизвестной теме"""
        result = await self.pipeline.process_question("Что такое квантовая
гравитация?")

        # Система должна корректно обработать отсутствие информации
        assert result["success"] == True
        assert len(result["documents"]) == 0 or "не знаю" in
result["answer"].lower()

    @pytest.mark.asyncio
    async def test_pipeline_caching(self):
        """Тест работы кэширования"""
        self.pipeline.config.enable_caching = True
```

```
        # Первый запрос
        result1 = await self.pipeline.process_question("Что такое ИИ?")
        assert result1["cached"] == False

        # Второй идентичный запрос
        result2 = await self.pipeline.process_question("Что такое ИИ?")
        assert result2["cached"] == True
        assert result1["answer"] == result2["answer"]

    def test_metrics_collection(self):
        """Тест сбора метрик"""
        initial_metrics = self.pipeline.get_metrics()
        assert initial_metrics["total_requests"] == 0

        # После выполнения запросов метрики должны обновиться
        asyncio.run(self.pipeline.process_question("Тестовый вопрос"))

        updated_metrics = self.pipeline.get_metrics()
        assert updated_metrics["total_requests"] > 0
        assert updated_metrics["average_processing_time"] > 0

if __name__ == "__main__":
    # Запуск тестов
    pytest.main([__file__, "-v"])
```

Этап 5: Демонстрация работы системы

1. **Создание демонстрационного скрипта:**

```
# demo/demo_pipeline.py
import asyncio
import time
from pipeline.rag_pipeline import RAGPipeline
from config.pipeline_config import PipelineConfig

async def demonstrate_pipeline():
    """Демонстрация работы полного RAG-конвейера"""
    print("🚀 Запуск демонстрации RAG-конвейера")
    print("=" * 50)

    pipeline = RAGPipeline()

    # Тестовые вопросы разной сложности
    test_cases = [
        "Что такое машинное обучение?",
        "Объясни разницу между AI и ML",
        "Какие типы нейронных сетей используются в компьютерном зрении?",
        "Что такое RAG архитектура и как она работает?",
        "Расскажи о квантовых вычислениях"  # Тема, которой нет в базе
знаний
    ]
```

```
    for i, question in enumerate(test_cases, 1):
        print(f"\n📝 Тест {i}: {question}")
        print("-" * 40)

        start_time = time.time()
        result = await pipeline.process_question(question)
        end_time = time.time()

        print(f"✅ Успех: {result['success']}")
        print(f"⏱️  Время обработки: {result['processing_time']:.2f}с")
        print(f"🔍 Найдено документов: {len(result['documents'])}")
        print(f"🤖 Ответ: {result['answer']}")
        print(f"📊 Из кэша: {result.get('cached', False)}")

        # Показ топ-документа если есть
        if result['documents']:
            best_doc = result['documents'][0]
            print(f"📄 Лучший документ: {best_doc['metadata']['title']}")
            print(f"🎯 Схожесть: {best_doc['similarity_score']:.3f}")

        print("-" * 40)

    # Показ метрик системы
    metrics = pipeline.get_metrics()
    print(f"\n📈 Метрики системы:")
    print(f"   Всего запросов: {metrics['total_requests']}")
    print(f"   Успешных: {metrics['successful_requests']}")
    print(f"   Среднее время: {metrics['average_processing_time']:.2f}с")
    print(f"   Попадания в кэш: {metrics['cache_hits']}")

if __name__ == "__main__":
    asyncio.run(demonstrate_pipeline())
```

Требования к оформлению и отчету (для Части 3)

Критерии оценки для Части 3:

- **Удовлетворительно:** Успешно выполнены Этапы 1-2 (проектирование и реализация базового конвейера). Система обрабатывает вопросы через полный пайплайн.
- **Хорошо:** Дополнительно успешно выполнен Этап 3 (создание API-сервиса). Конвейер доступен через REST API, реализованы основные эндпоинты.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы Этапы 4-5: комплексное тестирование, демонстрация работы, обработка ошибок, кэширование и мониторинг метрик. Система готова к production-использованию.

Рекомендуемая литература

1. **Production ML Systems:** https://martin.zinkevich.org/rules_of_ml/

2. **FastAPI Best Practices:** https://fastapi.tiangolo.com/deployment/
3. **RAG Production Patterns:** https://www.pinecone.io/learn/advanced-rag/
4. **Async Python:** https://docs.python.org/3/library/asyncio.html
5. **System Design for ML:** https://www.oreilly.com/library/view/designing-machine-learning/9781098107956/

Лабораторная работа №11-12, Часть 4: Оценка качества работы системы

Цель работы: Освоить методы и метрики для комплексной оценки качества RAG-системы. Научиться проводить объективное тестирование ретривера и генератора, анализировать результаты и выявлять направления для улучшения системы.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Окружение:** Conda (`mlops-lab`)
- **Метрики:** ROUGE, BLEU, Precision@K, Recall@K, NDCG
- **Фреймворки:** evaluate, sklearn, pandas
- **Визуализация:** matplotlib, seaborn
- **Анализ:** Jupyter Notebook, pandas profiling

Теоретическая часть

1. Метрики оценки RAG-систем Оценка RAG-системы проводится на трех уровнях:

- **Retrieval Quality:** Оценка качества поиска релевантных документов
- **Generation Quality:** Оценка качества сгенерированных ответов
- **End-to-End Quality:** Общая оценка работы системы

2. Метрики для ретривера

- **Precision@K:** Доля релевантных документов среди top-K результатов
- **Recall@K:** Доля найденных релевантных документов от общего числа релевантных
- **NDCG@K:** Учет порядка релевантных документов в результатах
- **MRR (Mean Reciprocal Rank):** Среднее обратное рангов релевантных документов

3. Метрики для генератора

- **ROUGE:** Сравнение с reference-ответами по n-граммам
- **BLEU:** Оценка качества машинного перевода (адаптируется для QA)
- **Semantic Similarity:** Косинусная схожесть эмбеддингов ответов
- **Answer Relevance:** Оценка релевантности ответа вопросу

4. Human Evaluation

- **Correctness:** Фактическая точность ответа
- **Completeness:** Полнота охвата темы
- **Coherence:** Логическая связность ответа
- **Helpfulness:** Практическая полезность ответа

Задание на практическую реализацию

Этап 1: Подготовка тестового датасета

1. **Создание эталонного датасета для оценки:**

```
# evaluation/test_dataset.py
from typing import List, Dict, Any
from dataclasses import dataclass

@dataclass
class TestCase:
    question: str
    ground_truth_answer: str
    relevant_document_ids: List[str]  # ID документов, которые должны быть
найдены
    category: str
    difficulty: str  # easy, medium, hard

# Эталонный датасет для оценки
EVALUATION_DATASET = [
    TestCase(
        question="Что такое машинное обучение?",
        ground_truth_answer="Машинное обучение — это область искусственного
интеллекта, которая использует статистические методы для создания моделей,
способных обучаться на данных и делать предсказания. Основные типы включают
обучение с учителем, без учителя и с подкреплением.",
        relevant_document_ids=["doc_001"],
        category="AI",
        difficulty="easy"
    ),
    TestCase(
        question="Какие архитектуры нейронных сетей используются в NLP?",
        ground_truth_answer="В обработке естественного языка используются
архитектуры трансформеров, такие как BERT для понимания текста и GPT для
генерации. Эти модели используют механизм внимания для учета контекста во
всей входной последовательности.",
        relevant_document_ids=["doc_003"],
        category="NLP",
        difficulty="medium"
    ),
    TestCase(
        question="Что такое RAG-архитектура и какие преимущества она дает?",
        ground_truth_answer="RAG (Retrieval-Augmented Generation) — это
архитектура, которая сочетает поиск информации в векторной базе данных с
генерацией текста языковой моделью. Это позволяет моделям работать с
актуальными данными, снижает вероятность галлюцинаций и повышает точность
ответов.",
        relevant_document_ids=["doc_005"],
        category="Architecture",
        difficulty="hard"
    ),
    TestCase(
        question="Какие бывают типы машинного обучения?",
```

```
        ground_truth_answer="Основные типы машинного обучения: обучение с
учителем (supervised learning), обучение без учителя (unsupervised learning)
и обучение с подкреплением (reinforcement learning).",
        relevant_document_ids=["doc_001"],
        category="AI",
        difficulty="easy"
    )
]

class EvaluationDataset:
    def __init__(self, test_cases: List[TestCase] = None):
        self.test_cases = test_cases or EVALUATION_DATASET

    def get_cases_by_category(self, category: str) -> List[TestCase]:
        return [case for case in self.test_cases if case.category ==
category]

    def get_cases_by_difficulty(self, difficulty: str) -> List[TestCase]:
        return [case for case in self.test_cases if case.difficulty ==
difficulty]

    def get_statistics(self) -> Dict[str, Any]:
        stats = {
            "total_cases": len(self.test_cases),
            "by_category": {},
            "by_difficulty": {}
        }

        for case in self.test_cases:
            stats["by_category"][case.category] =
stats["by_category"].get(case.category, 0) + 1
            stats["by_difficulty"][case.difficulty] =
stats["by_difficulty"].get(case.difficulty, 0) + 1

        return stats
```

Этап 2: Реализация системы оценки ретривера

1. **Создание модуля оценки поиска:**

```
# evaluation/retrieval_evaluator.py
import numpy as np
from typing import List, Dict, Any
from sklearn.metrics import precision_score, recall_score
from .test_dataset import TestCase

class RetrievalEvaluator:
    def __init__(self, vector_store):
        self.vector_store = vector_store

    def evaluate_retrieval(self, test_case: TestCase, top_k: int = 5) ->
```

```
Dict[str, float]:
        """Оценка качества поиска для одного тестового случая"""
        # Выполнение поиска
        retrieved_docs = self.vector_store.search(test_case.question,
n_results=top_k)
        retrieved_ids = [doc['metadata'].get('id', '') for doc in
retrieved_docs]

        # Бинарные метки релевантности
        true_relevant = set(test_case.relevant_document_ids)
        retrieved_set = set(retrieved_ids)

        # Расчет метрик
        precision = self._calculate_precision(retrieved_set, true_relevant)
        recall = self._calculate_recall(retrieved_set, true_relevant)
        f1 = self._calculate_f1(precision, recall)
        mrr = self._calculate_mrr(retrieved_ids, true_relevant)

        return {
            "precision": precision,
            "recall": recall,
            "f1_score": f1,
            "mrr": mrr,
            "retrieved_count": len(retrieved_set),
            "relevant_count": len(true_relevant),
            "retrieved_ids": retrieved_ids
        }

    def evaluate_dataset(self, test_cases: List[TestCase], top_k: int = 5) -
> Dict[str, Any]:
        """Оценка на всем датасете"""
        results = []

        for test_case in test_cases:
            result = self.evaluate_retrieval(test_case, top_k)
            result["question"] = test_case.question
            result["category"] = test_case.category
            result["difficulty"] = test_case.difficulty
            results.append(result)

        # Агрегированные метрики
        aggregated = self._aggregate_metrics(results)
        aggregated["total_cases"] = len(test_cases)

        return {
            "detailed_results": results,
            "aggregated_metrics": aggregated
        }

    def _calculate_precision(self, retrieved: set, relevant: set) -> float:
        if len(retrieved) == 0:
            return 0.0
        return len(retrieved & relevant) / len(retrieved)
```

```
    def _calculate_recall(self, retrieved: set, relevant: set) -> float:
        if len(relevant) == 0:
            return 0.0
        return len(retrieved & relevant) / len(relevant)

    def _calculate_f1(self, precision: float, recall: float) -> float:
        if precision + recall == 0:
            return 0.0
        return 2 * (precision * recall) / (precision + recall)

    def _calculate_mrr(self, retrieved_ids: List[str], relevant: set) ->
float:
        for rank, doc_id in enumerate(retrieved_ids, 1):
            if doc_id in relevant:
                return 1.0 / rank
        return 0.0

    def _aggregate_metrics(self, results: List[Dict]) -> Dict[str, float]:
        metrics = ["precision", "recall", "f1_score", "mrr"]
        aggregated = {}

        for metric in metrics:
            values = [r[metric] for r in results if r[metric] is not None]
            aggregated[f"mean_{metric}"] = np.mean(values) if values else
0.0
            aggregated[f"std_{metric}"] = np.std(values) if values else 0.0

        return aggregated
```

Этап 3: Реализация системы оценки генератора

1. **Создание модуля оценки генерации:**

```
# evaluation/generation_evaluator.py
import evaluate
import numpy as np
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity
from typing import List, Dict, Any
from .test_dataset import TestCase

class GenerationEvaluator:
    def __init__(self):
        self.rouge = evaluate.load('rouge')
        self.bleu = evaluate.load('bleu')
        self.similarity_model = SentenceTransformer('all-MiniLM-L6-v2')

    def evaluate_generation(self, generated_answer: str,
                            ground_truth: str) -> Dict[str, float]:
        """Оценка качества сгенерированного ответа"""
```

```
        # ROUGE метрики
        rouge_results = self.rouge.compute(
            predictions=[generated_answer],
            references=[ground_truth]
        )

        # BLEU метрик
        bleu_results = self.bleu.compute(
            predictions=[generated_answer],
            references=[[ground_truth]]
        )

        # Семантическая схожесть
        semantic_similarity = self._calculate_semantic_similarity(
            generated_answer, ground_truth
        )

        # Длина ответа (простейшая метрика качества)
        answer_length = len(generated_answer.split())

        return {
            "rouge1": rouge_results["rouge1"],
            "rouge2": rouge_results["rouge2"],
            "rougeL": rouge_results["rougeL"],
            "bleu": bleu_results["bleu"],
            "semantic_similarity": semantic_similarity,
            "answer_length": answer_length
        }

    def evaluate_answers(self, test_cases: List[TestCase],
                         generated_answers: List[str]) -> Dict[str, Any]:
        """Оценка набора сгенерированных ответов"""
        results = []

        for test_case, generated_answer in zip(test_cases,
generated_answers):
            evaluation = self.evaluate_generation(
                generated_answer,
                test_case.ground_truth_answer
            )

            result = {
                "question": test_case.question,
                "generated_answer": generated_answer,
                "ground_truth": test_case.ground_truth_answer,
                "category": test_case.category,
                "difficulty": test_case.difficulty
            }
            result.update(evaluation)
            results.append(result)

        # Агрегированные метрики
        aggregated = self._aggregate_generation_metrics(results)
```

```
        return {
            "detailed_results": results,
            "aggregated_metrics": aggregated
        }

    def _calculate_semantic_similarity(self, text1: str, text2: str) ->
float:
        """Вычисление семантической схожести через эмбеддинги"""
        embeddings = self.similarity_model.encode([text1, text2])
        similarity = cosine_similarity(
            [embeddings[0]],
            [embeddings[1]]
        )[0][0]
        return float(similarity)

    def _aggregate_generation_metrics(self, results: List[Dict]) ->
Dict[str, float]:
        metrics = ["rouge1", "rouge2", "rougeL", "bleu",
"semantic_similarity"]
        aggregated = {}

        for metric in metrics:
            values = [r[metric] for r in results]
            aggregated[f"mean_{metric}"] = np.mean(values)
            aggregated[f"std_{metric}"] = np.std(values)
            aggregated[f"min_{metric}"] = np.min(values)
            aggregated[f"max_{metric}"] = np.max(values)

        # Дополнительные метрики
        lengths = [r["answer_length"] for r in results]
        aggregated["mean_answer_length"] = np.mean(lengths)

        return aggregated
```

Этап 4: Комплексная оценка RAG-системы

1. **Создание комплексного оценщика:**

```
# evaluation/rag_evaluator.py
import pandas as pd
import json
from datetime import datetime
from typing import List, Dict, Any
import logging

from .test_dataset import TestCase, EvaluationDataset
from .retrieval_evaluator import RetrievalEvaluator
from .generation_evaluator import GenerationEvaluator
from pipeline.rag_pipeline import RAGPipeline

logger = logging.getLogger(__name__)
```

```
class RAGEvaluator:
    def __init__(self, rag_pipeline: RAGPipeline):
        self.pipeline = rag_pipeline
        self.retrieval_evaluator =
RetrievalEvaluator(rag_pipeline.retriever)
        self.generation_evaluator = GenerationEvaluator()
        self.dataset = EvaluationDataset()

    def run_comprehensive_evaluation(self, test_cases: List[TestCase] =
None) -> Dict[str, Any]:
        """Запуск комплексной оценки RAG-системы"""
        if test_cases is None:
            test_cases = self.dataset.test_cases

        logger.info(f"Starting comprehensive evaluation with
{len(test_cases)} test cases")

        # Этап 1: Оценка ретривера
        retrieval_results =
self.retrieval_evaluator.evaluate_dataset(test_cases)

        # Этап 2: Получение ответов от полной системы
        generated_answers = []
        for test_case in test_cases:
            try:
                result = await
self.pipeline.process_question(test_case.question)
                generated_answers.append(result["answer"])
            except Exception as e:
                logger.error(f"Error processing question:
{test_case.question}, error: {e}")
                generated_answers.append("")  # Пустой ответ в случае ошибки

        # Этап 3: Оценка генерации
        generation_results = self.generation_evaluator.evaluate_answers(
            test_cases, generated_answers
        )

        # Этап 4: Агрегация результатов
        final_report = self._compile_final_report(
            retrieval_results,
            generation_results,
            test_cases
        )

        return final_report

    def evaluate_by_category(self) -> Dict[str, Any]:
        """Оценка по категориям вопросов"""
        categories = set(case.category for case in self.dataset.test_cases)
        category_results = {}

        for category in categories:
```

```
            category_cases = self.dataset.get_cases_by_category(category)
            if category_cases:
                results = self.run_comprehensive_evaluation(category_cases)
                category_results[category] = results["aggregated_metrics"]

        return category_results

    def evaluate_by_difficulty(self) -> Dict[str, Any]:
        """Оценка по сложности вопросов"""
        difficulties = ["easy", "medium", "hard"]
        difficulty_results = {}

        for difficulty in difficulties:
            difficulty_cases =
self.dataset.get_cases_by_difficulty(difficulty)
            if difficulty_cases:
                results =
self.run_comprehensive_evaluation(difficulty_cases)
                difficulty_results[difficulty] =
results["aggregated_metrics"]

        return difficulty_results

    def _compile_final_report(self, retrieval_results: Dict,
                              generation_results: Dict,
                              test_cases: List[TestCase]) -> Dict[str, Any]:
        """Компиляция финального отчета"""

        # Объединение метрик
        aggregated_metrics = {
            "retrieval": retrieval_results["aggregated_metrics"],
            "generation": generation_results["aggregated_metrics"],
            "overall_score": self._calculate_overall_score(
                retrieval_results["aggregated_metrics"],
                generation_results["aggregated_metrics"]
            )
        }

        # Детальные результаты
        detailed_results = []
        for retrieval, generation in zip(
            retrieval_results["detailed_results"],
            generation_results["detailed_results"]
        ):
            combined = {**retrieval, **generation}
            detailed_results.append(combined)

        report = {
            "evaluation_timestamp": datetime.now().isoformat(),
            "test_cases_count": len(test_cases),
            "aggregated_metrics": aggregated_metrics,
            "detailed_results": detailed_results,
            "dataset_statistics": self.dataset.get_statistics()
        }
```

```
        return report

    def _calculate_overall_score(self, retrieval_metrics: Dict,
                                 generation_metrics: Dict) -> float:
        """Расчет общего скора системы"""
        # Веса для разных компонентов
        weights = {
            "retrieval_precision": 0.3,
            "retrieval_recall": 0.2,
            "generation_semantic_similarity": 0.3,
            "generation_rougeL": 0.2
        }

        score = 0
        score += retrieval_metrics.get("mean_precision", 0) *
weights["retrieval_precision"]
        score += retrieval_metrics.get("mean_recall", 0) *
weights["retrieval_recall"]
        score += generation_metrics.get("mean_semantic_similarity", 0) *
weights["generation_semantic_similarity"]
        score += generation_metrics.get("mean_rougeL", 0) *
weights["generation_rougeL"]

        return score

    def save_report(self, report: Dict[str, Any], filename: str = None):
        """Сохранение отчета в файл"""
        if filename is None:
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            filename = f"rag_evaluation_report_{timestamp}.json"

        with open(filename, 'w', encoding='utf-8') as f:
            json.dump(report, f, indent=2, ensure_ascii=False)

        logger.info(f"Evaluation report saved to {filename}")

    def generate_summary(self, report: Dict[str, Any]) -> str:
        """Генерация текстовой сводки"""
        metrics = report["aggregated_metrics"]

        summary = f"""
📊 ОТЧЕТ ОЦЕНКИ RAG-СИСТЕМЫ
============================

Общая информация:
- Время оценки: {report['evaluation_timestamp']}
- Количество тестовых случаев: {report['test_cases_count']}
- Общий score системы: {metrics['overall_score']:.3f}

Качество поиска (Retrieval):
- Precision: {metrics['retrieval']['mean_precision']:.3f} ±
{metrics['retrieval']['std_precision']:.3f}
- Recall: {metrics['retrieval']['mean_recall']:.3f} ± {metrics['retrieval']
```

```
['std_recall']:.3f}
- F1-Score: {metrics['retrieval']['mean_f1_score']:.3f} ±
{metrics['retrieval']['std_f1_score']:.3f}
- MRR: {metrics['retrieval']['mean_mrr']:.3f} ± {metrics['retrieval']
['std_mrr']:.3f}

Качество генерации (Generation):
- ROUGE-L: {metrics['generation']['mean_rougeL']:.3f} ±
{metrics['generation']['std_rougeL']:.3f}
- BLEU: {metrics['generation']['mean_bleu']:.3f} ± {metrics['generation']
['std_bleu']:.3f}
- Semantic Similarity: {metrics['generation']
['mean_semantic_similarity']:.3f} ± {metrics['generation']
['std_semantic_similarity']:.3f}
- Средняя длина ответа: {metrics['generation']['mean_answer_length']:.1f}
слов

Рекомендации по улучшению:
{self._generate_recommendations(metrics)}
        """
        return summary

    def _generate_recommendations(self, metrics: Dict) -> str:
        """Генерация рекомендаций по улучшению"""
        recommendations = []

        if metrics['retrieval']['mean_precision'] < 0.7:
            recommendations.append("• Улучшить точность поиска: настроить
эмбеддинг-модель или увеличить размер базы знаний")

        if metrics['retrieval']['mean_recall'] < 0.6:
            recommendations.append("• Увеличить полноту поиска: рассмотреть
использование гибридного поиска")

        if metrics['generation']['mean_semantic_similarity'] < 0.7:
            recommendations.append("• Улучшить качество генерации: настроить
промпты или использовать более мощную языковую модель")

        if metrics['generation']['mean_rougeL'] < 0.4:
            recommendations.append("• Работать над соответствием эталонным
ответам: добавить few-shot примеры в промпты")

        if not recommendations:
            recommendations.append("• Система показывает хорошие результаты!
Рекомендуется продолжить мониторинг качества.")

        return "\n".join(recommendations)
```

Этап 5: Визуализация и анализ результатов

1. **Создание модуля визуализации:**

```python
# evaluation/visualization.py
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from typing import Dict, Any
import numpy as np

class ResultsVisualizer:
    def __init__(self):
        plt.style.use('default')
        sns.set_palette("husl")

    def plot_metrics_comparison(self, report: Dict[str, Any], save_path: str
= None):
        """Визуализация сравнения метрик"""
        metrics = report["aggregated_metrics"]

        fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(15, 10))
        fig.suptitle('Сравнение метрик качества RAG-системы', fontsize=16)

        # Retrieval метрики
        retrieval_metrics = ['mean_precision', 'mean_recall',
'mean_f1_score', 'mean_mrr']
        retrieval_values = [metrics['retrieval'][m] for m in
retrieval_metrics]
        retrieval_labels = ['Precision', 'Recall', 'F1-Score', 'MRR']

        bars1 = ax1.bar(retrieval_labels, retrieval_values, alpha=0.7)
        ax1.set_title('Качество поиска (Retrieval)')
        ax1.set_ylim(0, 1)
        self._add_value_labels(ax1, bars1)

        # Generation метрики
        generation_metrics = ['mean_rouge1', 'mean_rouge2', 'mean_rougeL',
'mean_bleu']
        generation_values = [metrics['generation'][m] for m in
generation_metrics]
        generation_labels = ['ROUGE-1', 'ROUGE-2', 'ROUGE-L', 'BLEU']

        bars2 = ax2.bar(generation_labels, generation_values, alpha=0.7)
        ax2.set_title('Качество генерации (Generation)')
        ax2.set_ylim(0, 1)
        self._add_value_labels(ax2, bars2)

        # Semantic similarity
        similarity_data = [metrics['generation']
['mean_semantic_similarity']]
        ax3.bar(['Semantic\nSimilarity'], similarity_data, alpha=0.7,
color='green')
        ax3.set_title('Семантическая схожесть')
        ax3.set_ylim(0, 1)
        ax3.text(0, similarity_data[0] + 0.02, f'{similarity_data[0]:.3f}',
                 ha='center', va='bottom')
```

```
        # Overall score
        overall_data = [metrics['overall_score']]
        ax4.bar(['Overall\nScore'], overall_data, alpha=0.7, color='orange')
        ax4.set_title('Общая оценка системы')
        ax4.set_ylim(0, 1)
        ax4.text(0, overall_data[0] + 0.02, f'{overall_data[0]:.3f}',
                ha='center', va='bottom')

        plt.tight_layout()

        if save_path:
            plt.savefig(save_path, dpi=300, bbox_inches='tight')
            print(f"График сохранен в: {save_path}")

        plt.show()

    def plot_category_analysis(self, category_results: Dict[str, Any],
save_path: str = None):
        """Анализ результатов по категориям"""
        categories = list(category_results.keys())

        # Подготовка данных
        precision_scores = [category_results[cat]['retrieval']
['mean_precision'] for cat in categories]
        similarity_scores = [category_results[cat]['generation']
['mean_semantic_similarity'] for cat in categories]

        x = np.arange(len(categories))
        width = 0.35

        fig, ax = plt.subplots(figsize=(12, 6))
        bars1 = ax.bar(x - width/2, precision_scores, width,
label='Precision', alpha=0.7)
        bars2 = ax.bar(x + width/2, similarity_scores, width,
label='Semantic Similarity', alpha=0.7)

        ax.set_xlabel('Категории вопросов')
        ax.set_ylabel('Score')
        ax.set_title('Сравнение качества по категориям вопросов')
        ax.set_xticks(x)
        ax.set_xticklabels(categories)
        ax.legend()
        ax.set_ylim(0, 1)

        self._add_value_labels(ax, bars1)
        self._add_value_labels(ax, bars2)

        plt.tight_layout()

        if save_path:
            plt.savefig(save_path, dpi=300, bbox_inches='tight')

        plt.show()
```

```
    def _add_value_labels(self, ax, bars):
        """Добавление значений на столбцы графика"""
        for bar in bars:
            height = bar.get_height()
            ax.text(bar.get_x() + bar.get_width() / 2., height + 0.01,
                    f'{height:.3f}', ha='center', va='bottom')
```

Этап 6: Запуск комплексной оценки

1. **Создание скрипта для оценки:**

```
# run_evaluation.py
import asyncio
import json
from datetime import datetime
from pipeline.rag_pipeline import RAGPipeline
from evaluation.rag_evaluator import RAGEvaluator
from evaluation.visualization import ResultsVisualizer

async def main():
    print("🚀 Запуск комплексной оценки RAG-системы")
    print("=" * 50)

    # Инициализация системы
    pipeline = RAGPipeline()
    evaluator = RAGEvaluator(pipeline)
    visualizer = ResultsVisualizer()

    # Запуск оценки
    print("📋 Выполнение оценки...")
    report = await evaluator.run_comprehensive_evaluation()

    # Анализ по категориям
    print("📊 Анализ по категориям...")
    category_results = evaluator.evaluate_by_category()

    # Анализ по сложности
    print("🎯 Анализ по сложности...")
    difficulty_results = evaluator.evaluate_by_difficulty()

    # Сохранение отчетов
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

    # Основной отчет
    evaluator.save_report(report,
f"reports/full_evaluation_{timestamp}.json")

    # Визуализация
    visualizer.plot_metrics_comparison(report,
f"reports/metrics_comparison_{timestamp}.png")
```

```
    visualizer.plot_category_analysis(category_results,
f"reports/category_analysis_{timestamp}.png")

    # Вывод сводки
    summary = evaluator.generate_summary(report)
    print(summary)

    print(f"\n✅ Оценка завершена! Отчеты сохранены в папке 'reports/'")

if __name__ == "__main__":
    asyncio.run(main())
```

Требования к оформлению и отчету (для Части 4)

Критерии оценки для Части 4:

- **Удовлетворительно:** Успешно выполнены Этапы 1-2 (создание тестового датасета, оценка ретривера). Рассчитаны базовые метрики поиска.
- **Хорошо:** Дополнительно успешно выполнен Этап 3 (оценка генератора). Получены метрики качества генерации и семантической схожести.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы Этапы 4-6: комплексная оценка системы, визуализация результатов, анализ по категориям и сложности. Создан детализированный отчет с рекомендациями по улучшению.

Рекомендуемая литература

1. **RAG Evaluation Survey:** https://arxiv.org/abs/2312.10997
2. **Hugging Face Evaluate:** https://huggingface.co/docs/evaluate/index
3. **ROUGE Metric:** https://aclanthology.org/W04-1013/
4. **BLEU Metric:** https://aclanthology.org/P02-1040/
5. **ML Evaluation Best Practices:** https://madewithml.com/courses/mlops/evaluation/

Лабораторная работа №13-18: Сквозной проект

Разработка прототипа системы управления знаниями

Цель работы: Разработать полнофункциональный прототип интеллектуальной системы управления знаниями, интегрирующий технологии MLOps, векторные и графовые базы данных, семантические технологии и современные языковые модели.

Объем работы: 6 недель (36 часов лабораторных работ + 36 часов самостоятельной работы)

Часть 1: Примерные темы проектов

Интеллектуальные системы для образования

1. **Умный поиск по академическим материалам** - RAG-система для научных статей и учебных материалов
2. **Персональный образовательный ассистент** - адаптивная система рекомендаций контента
3. **Система проверки академических работ** - анализ оригинальности и качества студенческих работ
4. **Интеллектуальный тьютор по программированию** - код-ревью и рекомендации по улучшению

Бизнес-аналитика и управление

5. **Анализ отзывов клиентов** - классификация и извлечение инсайтов из пользовательских отзывов
6. **Система экспертного поиска** - нахождение сотрудников с нужными компетенциями
7. **Интеллектуальный анализ бизнес-процессов** - оптимизация workflow на основе данных
8. **Система управления корпоративными знаниями** - онтология компании и интеллектуальный поиск

Техническая документация и IT

9. **Интеллектуальный поиск по технической документации** (детально рассмотрен ниже)
10. **Система анализа багов и issue tracking** - автоматическая категоризация и рекомендации
11. **Code documentation assistant** - автоматическое обновление документации
12. **Техническая поддержка на основе знаний** - чат-бот для IT-поддержки

Здравоохранение и наука

13. **Система анализа медицинских исследований** - поиск релевантных клинических случаев
14. **Персональный health assistant** - рекомендации на основе медицинской литературы
15. **Научный литературный поиск** - семантический поиск по научным публикациям
16. **Система диагностики поддержки** - анализ симптомов и медицинской базы знаний

Юриспруденция и compliance

17. **Юридический research assistant** - поиск по законодательной базе и прецедентам
18. **Система compliance monitoring** - отслеживание изменений в регуляторике
19. **Договорной анализатор** - автоматический анализ юридических документов
20. **Система правовых консультаций** - чат-бот на основе законодательства

Медиа и контент

21. **Персональный новостной агрегатор** - семантическая фильтрация новостей
22. **Система рекомендации контента** - на основе анализа поведения пользователей
23. **Автоматическое суммирование документов** - создание кратких выжимок
24. **Система модерации контента** - классификация и фильтрация пользовательского контента
25. **Интеллектуальный архив документов** - семантический поиск и категоризация

Дополнительно

26. Допускается использовать тему курсового или дипломного проекта.

Часть 2: Пошаговая инструкция реализации проекта

"Интеллектуальный поиск по технической документации"

Этап 1: Подготовка и проектирование (Неделя 1)

Шаг 1.1: Определение требований и сбор данных

- Идентификация типов документов: API документация, мануалы, спецификации, readme файлы
- Определение форматов: PDF, Markdown, HTML, Docx, plain text
- Сбор тестового набора данных (100-500 документов)
- Определение целевых use-cases: поиск методов API, troubleshooting, best practices

Шаг 1.2: Проектирование архитектуры

- Выбор между монолитной и микросервисной архитектурой
- Проектирование схемы данных и онтологии предметной области
- Определение стека технологий (см. варианты ниже)
- Планирование пайплайна обработки данных

Шаг 1.3: Выбор стека технологий

Вариант A: Базовый стек (рекомендуется для начала)

- **База данных:** ChromaDB (векторная), SQLite (метаданные)
- **Эмбеддинги:** sentence-transformers (all-MiniLM-L6-v2)
- **Языковая модель:** GPT-2 или DialoGPT-medium
- **Бэкенд:** FastAPI
- **Фронтенд:** Streamlit или простой React
- **Контейнеризация:** Docker

Вариант B: Продвинутый стек

- **База данных:** Weaviate или Qdrant (векторная), PostgreSQL (метаданные)
- **Эмбеддинги:** OpenAI embeddings или multilingual-e5-large
- **Языковая модель:** Llama 2 7B или Mistral 7B
- **Бэкенд:** FastAPI с асинхронной обработкой
- **Поисковый движок:** Elasticsearch (гибридный поиск)
- **Мониторинг:** MLflow, Prometheus, Grafana
- **Оркестрация:** Docker Compose или Kubernetes

Вариант C: Enterprise-стек

- **Векторная БД:** Pinecone или Azure Cognitive Search
- **LLM:** GPT-4 API или Azure OpenAI
- **Бэкенд:** FastAPI с авторизацией JWT
- **Кэширование:** Redis
- **Очереди:** Celery + RabbitMQ
- **Хранилище:** Azure Blob Storage или AWS S3
- **CI/CD:** GitHub Actions, Azure DevOps

Этап 2: Реализация ядра системы (Недели 2-3)

Шаг 2.1: Настройка инфраструктуры

- Создание виртуального окружения и установка зависимостей
- Настройка Docker-окружения
- Инициализация баз данных
- Настройка MLflow для трекинга экспериментов

Шаг 2.2: Пайплайн обработки документов

- Реализация загрузчиков для разных форматов (PyPDF2, python-docx, beautifulsoup4)
- Разработка препроцессинга: очистка текста, нормализация, удаление стоп-слов
- Реализация чанкинга: semantic chunking с перекрытием
- Генерация эмбеддингов и индексация в векторной БД

Шаг 2.3: Поисковый движок

- Реализация семантического поиска с фильтрацией по метаданным
- Настройка гибридного поиска (семантический + ключевые слова)
- Реализация re-ranking модели для улучшения результатов
- Добавление фацетного поиска по категориям, датам, типам документов

Шаг 2.4: Генеративный компонент

- Интеграция языковой модели для генерации ответов
- Разработка системы промптов с контекстом из найденных документов
- Реализация механизма цитирования источников
- Добавление fallback-механизмов при ошибках генерации

Этап 3: Интеграция и интерфейсы (Неделя 4)

Шаг 3.1: Бэкенд API

- Разработка RESTful API с FastAPI
- Реализация эндпоинтов: поиск, автодополнение, похожие документы
- Добавление аутентификации и авторизации
- Реализация rate limiting и кэширования

Шаг 3.2: Фронтенд интерфейс

- Создание поискового интерфейса с фильтрами
- Реализация страницы результатов с highlighting
- Добавление интерфейса для просмотра документов
- Разработка админ-панели для управления документами

Шаг 3.3: Интеграционные тесты

- Тестирование пайплайна обработки документов
- Тестирование поисковых запросов разной сложности
- Проверка генерации ответов на edge-cases
- Нагрузочное тестирование API

Этап 4: Оценка и оптимизация (Неделя 5)

Шаг 4.1: Оценка качества

- Сбор тестового набора запросов с ground truth
- Расчет метрик качества (см. Часть 3)
- Проведение A/B тестирования разных подходов
- User acceptance testing с реальными пользователями

Шаг 4.2: Оптимизация производительности

- Профилирование bottlenecks в пайплайне
- Оптимизация запросов к векторной БД
- Кэширование частых запросов и результатов
- Настройка индексов и параметров поиска

Шаг 4.3: Улучшение качества

- Fine-tuning эмбеддинг модели на domain-specific данных
- Оптимизация промптов для генерации
- Добавление feedback loop от пользователей
- Обновление онтологии на основе анализа запросов

Этап 5: Документация и развертывание (Неделя 6)

Шаг 5.1: Документация системы

- Создание архитектурной документации
- Написание user guide для поиска
- Документация API для разработчиков

- Инструкции по развертыванию и maintenance

Шаг 5.2: Подготовка к production

- Настройка мониторинга и логирования
- Реализация health checks и метрик
- Настройка backup и recovery процедур
- Документирование процедур масштабирования

Шаг 5.3: Финальная презентация

- Подготовка демонстрации системы
- Создание презентации с ключевыми метриками
- Подготовка отчета о проделанной работе
- Планирование дальнейшего развития

Часть 3: Метрики оценки проекта

Метрики качества поиска (Retrieval Metrics)

1. **Precision@K** - точность среди первых K результатов
    - *Целевое значение:* 0.7-0.9 для @5
    - *Формула:* Relevant retrieved / Total retrieved
2. **Recall@K** - полнота среди первых K результатов
    - *Целевое значение:* 0.6-0.8 для @10
    - *Формула:* Relevant retrieved / Total relevant
3. **NDCG@K** - учет порядка релевантных документов
    - *Целевое значение:* 0.7-0.85 для @5
    - *Особенности:* Лучше для ранжирования чем Precision/Recall
4. **MRR (Mean Reciprocal Rank)** - среднее обратное ранга первого релевантного документа
    - *Целевое значение:* 0.6-0.8
    - *Формула:* Mean(1/rank_first_relevant)

Метрики качества генерации (Generation Metrics)

5. **ROUGE-L** - схожесть с reference ответами
    - *Целевое значение:* 0.4-0.6
    - *Особенности:* Хорошо для суммаризации, требует референсы
6. **BLEU** - оценка качества машинного перевода
    - *Целевое значение:* 0.3-0.5
    - *Примечание:* Адаптируется для QA систем

7. **Semantic Similarity** - косинусная схожесть эмбеддингов
    - *Целевое значение:* 0.7-0.85
    - *Метод:* Sentence transformers embeddings
8. **Answer Relevance** - оценка релевантности ответа вопросу
    - *Целевое значение:* 0.8-0.95
    - *Метод:* Классификация или человеческая оценка

Системные метрики

9. **Response Time P95** - 95-й перцентиль времени ответа
    - *Целевое значение:* < 3 секунд
    - *Измерение:* End-to-end latency
10. **Throughput** - количество запросов в секунду
    - *Целевое значение:* > 10 req/s (зависит от аппаратуры)
    - *Важно:* Для планирования масштабирования
11. **Cache Hit Rate** - процент попаданий в кэш
    - *Целевое значение:* 40-60%
    - *Оптимизация:* Уменьшает нагрузку на LLM
12. **Error Rate** - процент ошибочных ответов
    - *Целевое значение:* < 5%
    - *Мониторинг:* Критично для production

Бизнес-метрики

13. **User Satisfaction Score** - оценка удовлетворенности пользователей
    - *Целевое значение:* > 4.0/5.0
    - *Сбор:* Через feedback формы
14. **Task Success Rate** - процент успешно выполненных поисковых задач
    - *Целевое значение:* > 70%
    - *Измерение:* A/B тестирование
15. **Time to Resolution** - время решения проблемы пользователя
    - *Целевое значение:* Снижение на 30-50% vs baseline
    - *Бизнес-ценность:* Прямой ROI

Часть 4: Чек-лист проверки проекта

Функциональные требования ✅

- [ ] Система загружает документы различных форматов (PDF, MD, HTML)
- [ ] Реализован семантический поиск по содержимому документов
- [ ] Поддерживается фильтрация по метаданным (тип, дата, категория)
- [ ] Генерируются связные ответы на основе найденной информации
- [ ] Пользовательский интерфейс интуитивно понятен и отзывчив
- [ ] API документация доступна через Swagger/OpenAPI
- [ ] Реализована базовая аутентификация и авторизация
- [ ] Система обрабатывает edge-cases и ошибки корректно

Технические требования 🔧

- [ ] Код покрыт unit-тестами (минимум 70% coverage)
- [ ] Реализована контейнеризация через Docker
- [ ] Настроено логирование различных уровней
- [ ] Конфигурация вынесена в отдельные файлы
- [ ] Используется система контроля версий (Git)
- [ ] Реализованы health checks для мониторинга
- [ ] Настроен кэширование частых запросов
- [ ] Документация кода покрывает основные модули

Качество данных и ML 📊

- [ ] Реализован пайплайн предобработки документов
- [ ] Чанкинг документов настроен адекватно предметной области
- [ ] Эмбеддинг модель подобрана под специфику текстов
- [ ] Качество поиска измеряется объективными метриками
- [ ] Промпты для генерации оптимизированы под домен
- [ ] Система дает цитаты из источников при генерации
- [ ] Реализован механизм обратной связи от пользователей

Production readiness 🚀

- [ ] Response time укладывается в целевые показатели
- [ ] Error rate не превышает допустимые значения
- [ ] Реализована стратегия обработки перегрузок
- [ ] Настроен мониторинг ключевых метрик
- [ ] Документированы процедуры развертывания
- [ ] Реализованы backup механизмы для данных
- [ ] Система масштабируется горизонтально

Документация и отчетность 📝

- [ ] Архитектурная документация создана и актуальна
- [ ] User guide покрывает основные сценарии использования
- [ ] API документация полная и понятная
- [ ] Отчет по метрикам качества подготовлен
- [ ] Презентация проекта отражает ключевые достижения
- [ ] План дальнейшего развития сформулирован

Часть 5: Рекомендуемая литература

1. **Lewis, P., et al. (2020). "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks"** *NeurIPS 2020* **Аннотация:** Фундаментальная работа, представляющая архитектуру RAG. Авторы демонстрируют комбинацию dense passage retrieval с генеративными моделями, достигая state-of-the-art результатов на задачах открытого QA.

2. **Karpukhin, V., et al. (2020). "Dense Passage Retrieval for Open-Domain Question Answering"** *EMNLP 2020* **Аннотация:** Детальное исследование dense retrieval методов, показывающее превосходство над традиционными sparse методами. Важное чтение для понимания современных подходов к семантическому поиску.

3. **Izacard, G., & Grave, E. (2021). "Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering"** *EACL 2021* **Аннотация:** Исследует интеграцию retrieval и generation компонентов, предлагая методы улучшения как точности поиска, так и качества генерации.

4. **Gao, L., et al. (2023). "The AI Knowledge Stack: A Blueprint for Enterprise Knowledge Management Systems"** *arXiv:2305.10276* **Аннотация:** Практическое руководство по построению enterprise-систем управления знаниями, охватывающее архитектурные паттерны и best practices.

5. **Johnson, J., Douze, M., & Jégou, H. (2021). "Billion-scale similarity search with GPUs"** *IEEE Transactions on Big Data* **Аннотация:** Техническая статья о методах эффективного поиска ближайших соседей в больших векторных пространствах, критически важная для production систем.

6. **Raffel, C., et al. (2020). "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer"** *JMLR 2020* **Аннотация:** Фундаментальная работа по архитектуре T5, важная для понимания современных подходов к тонкой настройке языковых моделей.

7. **Wang, L., et al. (2022). "Evaluation of Retrieval-Augmented Generation: A Survey"** *arXiv:2212.13196* **Аннотация:** Комплексный обзор методов оценки RAG-систем, включая как автоматические метрики, так и человеческую оценку.

8. **Asai, A., et al. (2023). "Case-Based Reasoning for Natural Language Queries over Knowledge Bases"** *ACL 2023* **Аннотация:** Исследует подходы к решению сложных запросов через case-based reasoning, релевантно для систем работающих со структурированными знаниями.

9. **Chen, M., et al. (2023). "Towards Expert-Level Medical Question Answering with Large Language Models"** *Nature Medicine* **Аннотация:** Демонстрирует применение RAG-архитектуры в высокостакечной domain-specific области, показывая методики адаптации к специализированным доменам.

10. **Zhao, W., et al. (2023). "A Survey of Large Language Models"** *arXiv:2303.18223* **Аннотация:** Исчерпывающий обзор современных больших языковых моделей, их архитектур, возможностей и ограничений, необходимый для осознанного выбора моделей в проекте.

11. **Liu, N., et al. (2023). "Parameter-Efficient Fine-Tuning of Large Language Models: A Comprehensive Survey"** *arXiv:2303.15647* **Аннотация:** Детальный обзор методов эффективной

тонкой настройки (LoRA, Adapter, Prompt Tuning), критически важный для проектов с ограниченными вычислительными ресурсами.

12. **Sun, Y., et al. (2022). "A Survey of Knowledge-Intensive NLP with Pre-Trained Language Models"** *ACM Computing Surveys* **Аннотация:** Систематический обзор методов интеграции внешних знаний в языковые модели, включая retrieval-augmented подходы и knowledge graph интеграцию.