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/