**Отчет по дисциплине «Программная инженерия» по проекту:  
Определение версии численной модели по параметрам системы**

**Состав команды:**

* Еремин Юрий, группа РИМ-140903;
* Роговой Михаил, группа РИМ-140901;
* Рязанов Виктор, группа РИМ-140901;
* Хуснуллин Владислав, группа РИМ-140903.

Оглавление

[Глава 1. Введение 4](#_Toc186754658)

[Глава 2. Для кого и зачем нужен данный сервис? 5](#_Toc186754659)

[Выводы по разделу 6](#_Toc186754660)

[Глава 3. Модель машинного обучения 7](#_Toc186754661)

[Глава 4. Реализация API для приложения машинного обучения 8](#_Toc186754662)

[Выводы по разделу 10](#_Toc186754663)

[Глава 5. Реализация интерфейса через Streamlit 11](#_Toc186754664)

[Выводы по разделу 13](#_Toc186754665)

[Глава 6. Контейнеризация с использованием Docker 14](#_Toc186754666)

[Выводы по разделу 16](#_Toc186754667)

[Глава 7. Запуск локального приложения 17](#_Toc186754668)

[Выводы по разделу 20](#_Toc186754669)

[Глава 8. Альтернативный запуск приложения через Streamlit Cloud 21](#_Toc186754670)

[Выводы по разделу 23](#_Toc186754671)

[Глава 9. Настройка автотестов через GitHub Actions и добавление линтера 24](#_Toc186754672)

[Выводы по разделу 26](#_Toc186754673)

[Глава 10. Тестирование приложения 27](#_Toc186754674)

[Выводы по разделу 34](#_Toc186754675)

[Вывод по работе 35](#_Toc186754676)

[P.s. Если лень читать… 38](#_Toc186754677)

# Глава 1. Введение

Данный проект разработан для предоставления интерфейса, позволяющего пользователям выполнять различные задачи машинного обучения через API. Например, пользователь может увидеть *метрики* предоставляемой модели, или же получить *результаты предсказания версии численной модели* для различных параметров системы.

Основной особенностью проекта является его универсальность и возможность работы в различных средах. Это достигается благодаря использованию контейнеризации (Docker), разработке веб-приложения с помощью FastAPI и Streamlit, а также настройке автоматического тестирования и обеспечения качества кода через GitHub Actions и доавление линтера. Кроме того, проект включает инструменты для интерпретации моделей, такие как SHAP-анализ, и визуализацию Feature Importance, что делает его полезным для исследовательских задач в области высокотемпературных сверхпроводников.

Цель данного отчета — подробно рассмотреть архитектуру, функциональность и методы реализации API для приложения машинного обучения. Каждая глава будет посвящена отдельному аспекту проекта, включая описание модели машинного обучения, создание интерфейсов для взаимодействия с моделью, интеграцию с Docker, настройку тестирования и способы развертывания приложения.

Представленный проект может быть использован как пример для разработки аналогичных приложений, демонстрируя современные практики в области программной инженерии и машинного обучения. Такой подход позволяет эффективно сочетать технологии и упрощать их внедрение в повседневную практику. В последующих главах мы более детально разберем каждый этап работы над проектом.

# Глава 2. Для кого и зачем нужен данный сервис?

Реализуемый сервис предназначен исключительно в исследовательских целях, а именно в определении численной модели для расчета физической структуры. С физической стороны численная модель – это «часть» формулы:

Изображение выглядит как Шрифт, белый, линия, диаграмма

Автоматически созданное описание

Рисунок 1 – Модель Изинга.

В качестве физической модели рассматривается модель Изинга на квадратной решетке. Различные версии будут соответствовать различному набору параметров гамильтониана. Модель машинного обучения (ММО) обучаем на данных по всему диапазону модельных параметров.

Выделяют 3 версии этой формулы:

* **Версия "0"** – 2D модель Изинга в нулевом поле (в формуле пренебрегают 2-м и 3-м интегралом, оставляя лишь зависимость для первых соседей).
* **Версия "1"** – 2D модель Изинга с полем (в формуле пренебрегают 2-м интегралом, оставляя лишь зависимость от первых соседей и внешнего поля).
* Версия "2" – 2D модель Изинга со вторыми соседями (формула как на рис. 1).

В данной работе используется ММО для классификации 0-й и 1-й версии.

Также необходимо уточнить какие параметры определяют физическую систему:

* L – размерность системы (В данной работе L = const = 32 атома);
* T – температура системы;
* E – энергия системы;
* C – теплоемкость системы;
* FM – ферромагнитная часть намагниченности;
* Xfm – ферромагнитная часть магнитной восприимчивости;
* AFM – антиферромагнитная часть намагниченности;
* Xafm – антиферромагнитная часть магнитной восприимчивости.

При получении результата версии модели можно судить о численном

значении спина каждого атома системы. Данный метод исследования систем может облегчить в разы решение задачи о нахождение сверхпроводников при бОльшой температуре.

## Выводы по разделу

Резюмируя, получаем, что задача следующая: по **параметрам** системы – определить версию **0**/**1**, т.е. задача классификации. С данной задачей отлично справляются различные ММО, однако в этой работе будет использована модель XGBoost, о которой пойдет речь в следующих главе.

# Глава 3. Модель машинного обучения

Модель машинного обучения, использованная в проекте "AI Phase Diagram API", представляет собой алгоритм XGBoost. Это один из самых популярных и производительных инструментов для задач классификации и регрессии, основанный на методе градиентного бустинга над деревьями решений. XGBoost позволяет эффективно обрабатывать большие объемы данных и учитывает взаимосвязи между признаками для повышения точности предсказаний.

Обучение модели выполнено на основе данных, содержащихся в файле Main\_Data.csv. Эти данные включают в себя различные характеристики объектов, такие как температура, магнитная восприимчивость и другие параметры, описанные в вышеупомянутом разделе. Модель обучена определять целевой класс, анализируя предоставленные признаки.

Результаты работы модели включают:

1. **Метрики качества:** API позволяет пользователю оценить производительность модели, предоставляя такие показатели, как точность, полнота, F1-мера и ROC-AUC, также использование кросс-валидации.
2. **Интерпретация:** с использованием SHAP-анализа API предоставляет объяснение, какие признаки и в какой степени повлияли на результат предсказания. Это важно для повышения прозрачности и доверия к модели.
3. **Предсказание класса:** на основе входных данных модель выдает вероятность принадлежности объекта к каждому из возможных классов. Это позволяет пользователю принимать решения на основе количественной оценки риска или выгоды.

Использование модели интегрировано в функционал API, о реализации которой пойдет дальше речь.

# Глава 4. Реализация API для приложения машинного обучения

Разработка API для взаимодействия с моделью машинного обучения выполнена с использованием веб-фреймворка FastAPI. Этот фреймворк обеспечивает простой и быстрый запуск REST API, высокую производительность и возможность генерации интерактивной документации.

**Основные компоненты приложения**:

* **Используемый фреймворк:** API приложение разработано с использованием FastAPI, что позволяет создавать масштабируемые и удобные в обслуживании приложения.
* **Модель и данные:**
  + Модель машинного обучения загружается из файла **xgboost\_model.pkl**.
  + Для обучения и тестирования используются данные из файла **Main\_Data.csv**.
  + Код API реализован в файле **main.py**, который связывает модель и эндпоинты приложения.

**API эндпоинты:**

* /metrics/: Возвращает ключевые метрики модели (Accuracy, Precision, Recall, F1-score, ROC-AUC).
* /roc\_curve/: Генерирует и возвращает ROC-кривую как изображение.
* /cross\_validation/: Выполняет 5-кратную кросс-валидацию, возвращая результаты для каждого фолда и средний ROC-AUC.
* /cross\_validation\_roc\_auc/: Построение графика ROC-AUC по результатам кросс-валидации.
* /feature\_importance/: Генерация графика важности признаков модели.
* /shap\_analysis/: Визуализация анализа SHAP для интерпретации модели.
* /predict/: Принимает CSV-файл с данными, возвращает предсказания модели, вероятности и выполняет группировку по метке sample.

Ниже представлен Рисунок 2, на котором видна локальная реализация FastAPI, доступного по ссылке <http://localhost:8000/docs>. Также необходимо отметить, что первые 6 функций использованы в качестве GET, а 7-я функция в качестве POST, т.к. подразумевает загрузки «слепых» данных от пользователя для предсказания версии численной модели.

Изображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание

Рисунок 2 – Локальный запуск API.

**Особенности работы:**

* Приложение проверяет корректность данных и обрабатывает ошибки ввода, включая отсутствие обязательных колонок или некорректные типы данных.
* Подготовлен файл requirements.txt с перечислением всех необходимых библиотек для работы приложения.
* В файле README предоставлена инструкция по запуску API.
* В репозитории GitHub (<https://github.com/Maddenss/AI_phase_diagram_API/blob/master/test_data.zip> ) выложены все необходимые файлы для работы API, включая тестовые данные (Model\_0\_1\_test.data.csv и Model\_0\_1\_test.data\_2.csv).

## Выводы по разделу

Разработанное API предоставляет полный набор функций для анализа, предсказаний и оценки качества модели. Дополнительно реализованы визуализации ключевых графиков (ROC-кривые, важность признаков), что упрощает интерпретацию и анализ данных. Визуализация графиков выполнена с использованием библиотеки Matplotlib. Интеграция с тестовыми данными и наличие документации делают приложение удобным для использования и расширения.

# Глава 5. Реализация интерфейса через Streamlit

Файл **app.py** является веб-интерфейсом для взаимодействия с моделью машинного обучения и API. Он создан с использованием Streamlit — фреймворка для разработки интерактивных веб-приложений на Python. Этот файл обеспечивает пользователю доступ ко всем ключевым функциям API через удобный графический интерфейс.

**Основные функции файла app.py:**

1. **Отображение метрик модели:**
   * При нажатии кнопки "Рассчитать метрики" отправляется запрос к эндпоинту /metrics/.
   * Если запрос успешен, пользователь видит ключевые метрики модели: точность, полноту, F1-мера, ROC-AUC и другие.
2. **Построение ROC-кривой:**
   * Кнопка "Построить ROC-кривую" отправляет запрос к эндпоинту /roc\_curve/.
   * Возвращённое изображение ROC-кривой отображается в приложении.
3. **Кросс-валидация:**
   * При нажатии кнопки "Запустить кросс-валидацию" выполняется запрос к эндпоинту /cross\_validation/.
   * Отображаются результаты для каждого фолда, включая матрицы ошибок и значения ROC-AUC. Также показывается среднее значение ROC-AUC.
4. **Построение графика ROC-AUC для кросс-валидации:**
   * Кнопка "Построить график ROC-AUC" отправляет запрос к эндпоинту /cross\_validation\_roc\_auc/.
   * График визуализируется в приложении.
5. **Анализ важности признаков:**
   * Нажатие кнопки "Показать важность признаков" вызывает запрос к эндпоинту /feature\_importance/.
   * Отображается график важности признаков модели.
6. **SHAP-анализ:**
   * Кнопка "Построить SHAP-анализ" отправляет запрос к эндпоинту /shap\_analysis/.
   * Визуализация SHAP-анализов предоставляет пользователю объяснение работы модели, показывая вклад каждого признака в предсказания.
7. **Предсказания:**
   * Пользователь загружает CSV-файл с данными через встроенный загрузчик.
   * Файл отправляется к эндпоинту /predict/, а результаты предсказаний возвращаются и отображаются в таблице.
   * Пользователь может скачать результаты в виде CSV-файла через кнопку "Скачать результаты".

**Технические особенности реализации:**

* **Взаимодействие с API:** Все функции приложения связаны с API, развернутым с использованием FastAPI. Запросы выполняются через библиотеку requests.
* **Обработка ошибок:** Приложение проверяет статус ответа от API и выводит сообщения об ошибках в случае их возникновения.
* **Визуализация:** Графики и изображения (ROC-кривые, SHAP-анализ и важность признаков) обрабатываются с использованием библиотеки Matplotlib и отображаются через Streamlit.
* **Обработка данных:** Результаты запросов преобразуются в удобный для пользователя формат, включая таблицы и интерактивные элементы.

**Удобство использования:**

Файл app.py делает функционал модели доступным через интуитивно понятный интерфейс. Это особенно полезно для пользователей, не знакомых с техническими аспектами работы API или машинного обучения. Streamlit позволяет легко добавлять новые функции и адаптировать интерфейс под конкретные требования.

**Роль интерфейса(app.py) в проекте:**

* Предоставляет графический интерфейс для взаимодействия с моделью.
* Упрощает анализ данных и интерпретацию модели.
* Делает проект доступным для широкой аудитории, включая пользователей с минимальным опытом работы с ML.

## Выводы по разделу

Интерфейс играет центральную роль в предоставлении пользователю доступного и функционального интерфейса для работы с моделью машинного обучения и API. Его возможности включают визуализацию ключевых метрик и графиков, удобное управление данными и получение предсказаний. Приложение, построенное на основе Streamlit, обеспечивает простоту использования и гибкость, что делает его подходящим как для технических специалистов, так и для пользователей без глубоких знаний в области программирования или машинного обучения. Благодаря этому интерфейс становится важным инструментом, обеспечивающим доступность и эффективность работы с моделью.

# Глава 6. Контейнеризация с использованием Docker

В данном проекте для обеспечения универсальности и удобства развертывания приложения была использована технология контейнеризации Docker. Docker позволяет упаковать приложение и все его зависимости в изолированный контейнер, который может быть запущен на любой системе, поддерживающей Docker. Это значительно упрощает процесс разработки, тестирования и развертывания приложения, особенно в условиях, когда необходимо обеспечить одинаковую среду выполнения на разных этапах жизненного цикла проекта.

**Основные компоненты Docker в проекте:**

**Dockerfile для FastAPI (бэкенд-контейнер):**

Для контейнеризации FastAPI приложения был создан файл Dockerfile. Этот файл содержит инструкции для сборки образа, который включает в себя все необходимые зависимости и настройки для запуска API. Вот основные шаги, описанные в Dockerfile:

1. **Использование официального образа Python**: В качестве базового образа используется *python:3.9-slim*, который содержит минимальную версию *Python 3.9*.
2. **Установка рабочей директории**: Рабочая директория внутри контейнера устанавливается как */app*.
3. **Копирование и установка зависимостей**: Файл *requirements.txt*, содержащий список всех необходимых библиотек, копируется в контейнер, и зависимости устанавливаются с помощью команды *pip install*.
4. **Копирование исходного кода**: Весь исходный код проекта копируется в рабочую директорию контейнера.
5. **Запуск FastAPI**: Команда *CMD* запускает *FastAPI* приложение с использованием *uvicorn*, указывая хост и порт, на котором будет доступно API.

**Dockerfile для Streamlit (фронтенд-контейнер):**

Для контейнеризации Streamlit приложения был создан отдельный файл **Dockerfile.streamlit**. Этот файл аналогичен Dockerfile для FastAPI, но содержит инструкции для запуска Streamlit приложения:

1. **Использование официального образа Python**: Также используется *python:3.9-slim*.
2. **Установка рабочей директории**: Рабочая директория устанавливается как */app*.
3. **Копирование и установка зависимостей**: Файл *requirements.txt* копируется, и зависимости устанавливаются.
4. **Копирование исходного кода**: Весь исходный код копируется в контейнер.
5. **Запуск Streamlit**: Команда *CMD* запускает *Streamlit* приложение, указывая хост и порт, на котором будет доступен интерфейс.

**Docker Compose:**

Для одновременного запуска *FastAPI* и *Streamlit* приложений используется **docker-compose.yml**. Этот файл описывает два сервиса: *fastapi* и *streamlit*. Каждый сервис собирается из соответствующего *Dockerfile*, и для каждого из них указываются порты, которые будут доступны на хосте. Также используется параметр *depends\_on*, который гарантирует, что *Streamlit* приложение будет запущено только после успешного запуска *FastAPI*.

**Преимущества использования Docker:**

1. **Изоляция среды**: Docker обеспечивает изоляцию приложения и его зависимостей, что исключает проблемы, связанные с различиями в окружении на разных машинах.
2. **Упрощение развертывания**: Контейнеры могут быть легко развернуты на любой системе, поддерживающей Docker, что упрощает процесс переноса приложения между этапами разработки, тестирования и производства.
3. **Масштабируемость**: Docker Compose позволяет легко масштабировать приложение, добавляя новые сервисы или изменяя конфигурацию существующих.
4. **Совместимость**: Docker обеспечивает совместимость между различными операционными системами, что делает приложение более универсальным.

## Выводы по разделу

Использование Docker в данном проекте позволило создать универсальную и легко развертываемую среду для работы с FastAPI и Streamlit приложениями. Контейнеризация обеспечивает изоляцию и стабильность работы приложения, а также упрощает процесс разработки и развертывания. Docker Compose делает управление несколькими сервисами простым и эффективным, что особенно важно для сложных проектов, включающих несколько компонентов. В следующей главе мы рассмотрим процесс тестирования и обеспечения качества кода в проекте.

# Глава 7. Запуск локального приложения

В этом разделе мы рассмотрим, как запустить локальное приложение, состоящее из FastAPI и Streamlit, с использованием Docker. Весь процесс разбит на несколько простых шагов, которые позволят вам быстро развернуть проект на вашем компьютере.

**Предварительные требования:**

Перед началом убедитесь, что у вас установлены следующие инструменты:

1. **Docker** — для создания и запуска контейнеров. Установить Docker можно с [официального сайта](https://www.docker.com/).
2. **Docker Compose** — для управления несколькими контейнерами. Обычно он устанавливается вместе с Docker.
3. **Git** — для клонирования репозитория (если вы хотите использовать исходный код из репозитория, и не копировать по одному файлу в свою папку на ПК).

**Шаги для запуска приложения:**

**Клонирование репозитория (опционально):**

Если вы хотите использовать исходный код из репозитория, выполните следующую команду для клонирования проекта:

*git clone https://github.com/Maddenss/AI\_phase\_diagram\_API.git*

*cd AI\_phase\_diagram\_API*

Если у вас уже есть все файлы проекта, переходите к следующему шагу.

**Сборка и запуск контейнеров:**

В проекте используется **Docker Compose** для одновременного запуска FastAPI и Streamlit приложений. Для сборки и запуска контейнеров выполните следующую команду в корневой директории проекта:

*docker-compose up –build*

Эта команда выполнит следующие действия:

1. **Соберет образы** для FastAPI и Streamlit на основе Dockerfile и Dockerfile.streamlit.
2. **Запустит контейнеры** для обоих сервисов.
3. **Свяжет контейнеры** между собой, чтобы Streamlit мог взаимодействовать с FastAPI.

Ниже представлен Рисунок 3, который показывает вывод терминала, после сборки и запуска контейнеров:

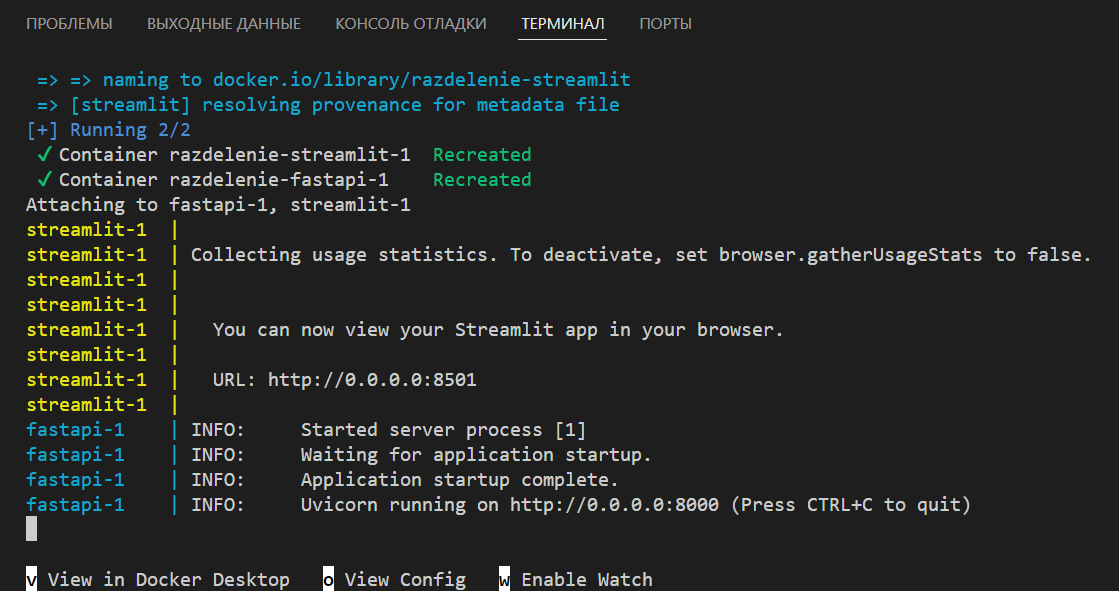


Рисунок 3 – Сборка и запуск контейнеров.

**Доступ к приложениям:**

После успешного запуска контейнеров вы сможете получить доступ к приложениям через браузер:

1. **FastAPI**: Откройте браузер и перейдите по адресу <http://localhost:8000/docs>. Здесь вы увидите интерактивную документацию API, где можно тестировать все доступные эндпоинты.
2. **Streamlit**: Перейдите по адресу [http://localhost:8501](http://localhost:8501/). Здесь вы найдете графический интерфейс для взаимодействия с моделью машинного обучения. Вы сможете загружать данные, получать предсказания и визуализировать результаты.

Ниже представлен Рисунок 4, который показывает локальное развертывание интерфейса на основе Streamlit:

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

Рисунок 4 – Локальное развертывание интерфейса Streamlit.

**Остановка контейнеров:**

Чтобы остановить контейнеры, выполните следующую команду в терминале:

*docker-compose down*

Эта команда остановит и удалит все запущенные контейнеры.

## Выводы по разделу

Запуск локального приложения с использованием Docker и Docker Compose позволяет быстро развернуть проект и начать работу с моделью машинного обучения. FastAPI предоставляет мощный API для взаимодействия с моделью, а Streamlit делает этот процесс доступным через удобный графический интерфейс. Следуя описанным шагам, вы сможете легко запустить приложение на своем компьютере и начать исследовать данные.

# Глава 8. Альтернативный запуск приложения через Streamlit Cloud

Запуск приложения через Streamlit Cloud — это удобный и быстрый способ сделать ваш проект доступным для широкой аудитории. В отличие от локального запуска, этот метод не требует настройки дополнительных сервисов и позволяет сосредоточиться на разработке функциональности. Файл **streamlit\_app.py** специально адаптирован для работы в облачной среде, что делает его идеальным выбором для развертывания через Streamlit Cloud.

**Что такое Streamlit Cloud?**

**Streamlit Cloud** – это облачный сервис, который мы использовали для развертывания нашего приложения в интернете. Streamlit Cloud интегрируется с GitHub, что позволяет автоматически развернуть приложение при каждом обновлении кода в репозитории. Это особенно удобно для демонстрации проектов и совместной работы.

**Чем отличается файл streamlit\_app.py?**

Файл **streamlit\_app.py** – это версия приложения, которую мы адаптировали для запуска через Streamlit Cloud. В отличие от локального запуска, где используется FastAPI для обработки запросов, в streamlit\_app.py вся логика обработки данных и взаимодействия с моделью машинного обучения реализована непосредственно в Streamlit. Это упростило развертывание, так как не потребовалось настройки дополнительных сервисов.

**Основные отличия:**

1. **Отсутствие FastAPI**: В streamlit\_app.py все функции, такие как расчет метрик, построение графиков и предсказания, выполняются непосредственно в Streamlit. Это сделало код более компактным и удобным для развертывания в облаке.
2. **Локальная обработка данных**: В отличие от локального запуска, где данные обрабатываются через API, в streamlit\_app.py все вычисления выполняются на стороне Streamlit. Это упростило архитектуру приложения, хотя может быть менее эффективно для больших объемов данных.
3. **Интеграция с GitHub**: Streamlit Cloud напрямую интегрируется с GitHub, что позволило нам автоматически развертывать приложение при каждом обновлении кода в репозитории**.**

**Как запустить приложение через Streamlit Cloud?**

1. **Регистрация на Streamlit Cloud:**

Мы зарегистрировались на [Streamlit Cloud](https://streamlit.io/cloud" \t "_blank) и авторизовались через GitHub. Это упростило процесс интеграции с нашим репозиторием.

2. **Подключение репозитория:**

После регистрации мы подключили наш репозиторий GitHub к Streamlit Cloud. Указали путь к файлу streamlit\_app.py и выбрали ветку, которую хотели использовать для развертывания.

3. **Настройка окружения:**

Streamlit Cloud автоматически создал окружение для нашего приложения. Все необходимые зависимости, указанные в файле requirements.txt, были установлены автоматически.

4. **Запуск приложения:**

После настройки Streamlit Cloud развернул наше приложение и предоставил уникальную ссылку, по которой оно стало доступно. Мы поделились этой ссылкой с другими пользователями, чтобы они могли взаимодействовать с приложением.

5. **Обновление приложения:**

Каждый раз, когда мы обновляли код в репозитории GitHub, Streamlit Cloud автоматически пересобирал и развертывал приложение. Это позволило нам быстро вносить изменения и тестировать новые функции.

**Преимущества запуска через Streamlit Cloud**

1. **Простота развертывания:** Нам не потребовалось настраивать серверы или управлять контейнерами. Streamlit Cloud автоматически развернул приложение и обеспечил его доступность.
2. **Доступность:** Приложение стало доступно через веб-браузер, что позволило нам делиться им с другими пользователями без необходимости локальной установки.
3. **Интеграция с GitHub:** Автоматическое обновление приложения при изменении кода в репозитории упростило процесс разработки и тестирования.
4. **Бесплатный тариф:** Мы использовали бесплатный тарифный план Streamlit Cloud, который идеально подошел для нашего проекта.

## Выводы по разделу

Запуск приложения через Streamlit Cloud — это способ, который мы также использовали для обеспечения доступности нашего проекта. В отличие от локального запуска, этот метод не потребовал настройки дополнительных сервисов и позволил нам сосредоточиться на разработке функциональности. Файл **streamlit\_app.py** был специально адаптирован для работы в облачной среде, что сделало его идеальным выбором для развертывания через Streamlit Cloud. Также можно подключиться к нашему приложению, через ссылку: <https://ai-phase-diagram.streamlit.app/>, которое подключено к нашему репозиторию: [https://github.com/Maddenss/AI\_phase\_diagram\_API/blob/master](https://github.com/Maddenss/AI_phase_diagram_API/blob/master.) .

# Глава 9. Настройка автотестов через GitHub Actions и добавление линтера

В нашем проекте мы использовали **GitHub Actions** для автоматизации процессов тестирования и проверки качества кода. GitHub Actions — это мощный инструмент, который позволяет запускать различные процессы (например, тестирование, сборку или деплой) на основе событий, таких как push или pull\_request. В этом разделе мы подробно рассмотрим, как были настроены автотесты и линтер, а также какие преимущества это принесло проекту.

**Принцип работы автотестов с использованием GitHub Actions:**

Автотесты в нашем проекте запускаются в двух основных сценариях:

1. **Push в ветку master**: При каждом обновлении ветки master автоматически запускаются тесты, чтобы убедиться, что изменения не нарушили работоспособность приложения.
2. **Создание или обновление pull request**: При создании или обновлении pull request, направленного в ветку master, также запускаются тесты. Это позволяет проверить, что новые изменения совместимы с основной веткой.

**Принцип работы автотестов:**

Автотесты выполняются на виртуальной машине с операционной системой **Ubuntu 24.04**. Процесс тестирования состоит из нескольких шагов:

1. **Клонирование репозитория**: Содержимое репозитория загружается в контейнер, чтобы можно было работать с его файлами.
2. **Настройка PYTHONPATH**: Текущая директория устанавливается в переменную окружения PYTHONPATH, чтобы Python мог найти нужные модули внутри репозитория.
3. **Настройка Python**: Устанавливается Python версии **3.9**, которая используется в нашем проекте.
4. **Установка зависимостей**: Устанавливаются все библиотеки, указанные в файле requirements.txt.
5. **Запуск линтера**: Проверяется стиль кода на соответствие стандартам. В нашем проекте используется линтер **flake8**, который проверяет код на соответствие стандарту **PEP8**. Максимальная длина строки установлена в 140 символов.
6. **Запуск тестов**: Тесты запускаются с помощью **pytest**. Мы используем флаг --disable-warnings для отключения предупреждений и флаг -q для режима подробного вывода.
7. **Загрузка результатов тестов**: Если тесты завершились сбоем, результаты сохраняются в виде артефактов, что облегчает диагностику проблем.

**Результаты использования GitHub Actions**

Внедрение GitHub Actions в наш проект принесло несколько важных преимуществ:

1. **Автоматизация**: При каждом изменении в ветке master или при создании pull request все тесты выполняются автоматически. Это позволяет быстро выявлять ошибки и обеспечивает стабильность кода.
2. **Поддержка стиля и чистоты кода**: Линтер помогает поддерживать единый стиль написания кода, что делает его более читаемым и удобным для работы.
3. **Мониторинг и диагностика**: В случае сбоя результаты тестов сохраняются, что упрощает процесс диагностики и исправления ошибок.
4. **Тестирование на стандартной платформе**: Код тестируется на стандартной платформе (Ubuntu 24.04), что минимизирует вероятность ошибок, связанных с различиями в окружении.

## Выводы по разделу

Использование GitHub Actions для автоматизации тестирования и проверки качества кода стало важной частью нашего проекта. Это позволило нам обеспечить стабильность и надежность приложения, а также поддерживать высокий уровень качества кода. Линтер и автотесты, настроенные через GitHub Actions, помогают быстро выявлять и исправлять ошибки, что делает процесс разработки более эффективным и предсказуемым.

# Глава 10. Тестирование приложения

В рамках нашего проекта мы провели комплексное тестирование приложения с моделью машинного обучения, чтобы обеспечить его стабильность, надежность и соответствие требованиям. Тестирование включало различные виды тестов, которые позволили проверить как отдельные компоненты приложения, так и их взаимодействие. В этом разделе мы подробно рассмотрим, какие тесты были проведены и какие результаты были получены.

**Виды тестов:**

**1. Модульные тесты (Unit Tests)**

Модульные тесты предназначены для проверки работы отдельных функций или методов приложения. Мы написали два модульных теста, которые проверяют корректность загрузки данных и модели.

**test\_unit\_1.py**

Этот тест проверяет корректность загрузки модели машинного обучения. Мы убедились, что модель успешно загружается из файла xgboost\_model.pkl и имеет необходимые методы, такие как predict.

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Автоматически созданное описание

Рисунок 5 – Результат работы теста test\_unit\_1.py

**Результаты**: Тест успешно прошел, что подтверждает корректность загрузки модели.

**test\_unit\_2.py**

Этот тест проверяет корректность загрузки данных из файла Main\_Data.csv. Мы проверили, что данные загружаются в виде DataFrame и Series, содержат ожидаемые колонки и не являются пустыми.

Изображение выглядит как текст, Шрифт, линия, число

Автоматически созданное описание

Рисунок 6 – Результат работы теста test\_unit\_2.py

**Результаты:** Тест успешно прошел, что подтверждает корректность загрузки данных.

**2. Интеграционные тесты (Integration Tests)**

Интеграционные тесты проверяют взаимодействие между различными компонентами приложения. Мы написали тест test\_integration.py, который проверяет корректность работы API и взаимодействие между загрузкой данных, моделью и кросс-валидацией.

**test\_integration.py**

Этот тест проверяет работу следующих эндпоинтов:

* /metrics/ — возвращает метрики модели.
* /roc\_curve/ — возвращает ROC-кривую.
* /cross\_validation/ — выполняет кросс-валидацию.
* /feature\_importance/ — возвращает график важности признаков.
* /shap\_analysis/ — возвращает SHAP-анализ.

Также тест проверяет обработку ошибок при загрузке некорректного файла и взаимодействие между загрузкой данных и модели.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рисунок 7 – Результат работы теста test\_integration.py

**Результаты**: Все интеграционные тесты успешно прошли, что подтверждает корректность работы API и взаимодействия между компонентами.

**3. Регрессионные тесты (Regression Tests)**

Регрессионные тесты предназначены для проверки того, что новые изменения в коде не нарушили существующую функциональность. Мы написали тест test\_regression.py, который сравнивает текущие предсказания модели с эталонными, сохраненными в файле predictions.json.

**test\_regression.py**

Этот тест проверяет, что предсказания модели остаются неизменными после изменений в коде.

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Автоматически созданное описание

Рисунок 8 – Результат работы теста test\_regression.py

**4. Приемочные тесты (Acceptance Tests)**

Приемочные тесты проверяют, что приложение соответствует требованиям и ожиданиям пользователей. Мы разработали тест test\_priem.py, который проверяет корректность обработки предсказаний и обработку ошибок при загрузке некорректного файла.

**test\_priem.py**

Этот тест проверяет, что API возвращает корректные результаты для загруженных данных и обрабатывает ошибки при загрузке некорректного файла.

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Автоматически созданное описание

Рисунок 9 – Результат работы теста test\_priem.py

**Результаты**: Приемочный тест успешно завершился, что подтверждает соответствие приложения требованиям пользователей.

**5. Нагрузочные тесты (Load Tests)**

Нагрузочные тесты проверяют, как приложение ведет себя под высокой нагрузкой. Мы написали тест test\_load.py, который имитирует большое количество запросов к API, чтобы проверить его производительность и стабильность.

**test\_load.py**

Этот тест проверяет стабильность системы при многократных запросах и рассчитывает среднее время отклика.

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Автоматически созданное описание

Рисунок 10 – Результат работы теста test\_load.py

**Результаты**: Нагрузочный тест показал, что приложение справляется с высокой нагрузкой без сбоев, что подтверждает его стабильность.

**6. Параметризованные тесты (Parametrized Tests)**

Параметризованные тесты позволяют проверить одну и ту же функцию с различными входными данными. Мы написали тест test\_param.py, который проверяет работу модели с различными входными параметрами и вычисляет метрику MSE (Mean Squared Error).

**test\_param.py**

Этот тест проверяет корректность работы модели на различных входных данных и вычисляет метрику MSE.

Изображение выглядит как текст, линия, Шрифт, снимок экрана

Автоматически созданное описание

Рисунок 11 – Результат работы теста test\_param.py

**Результаты**: Параметризованный тест успешно прошел, что подтверждает корректность работы модели при различных входных данных.

**Автотестирование через GitHub Actions:**

Для автоматизации процесса тестирования мы настроили **GitHub Actions**, который запускает все тесты при каждом изменении в ветке master или при создании pull request. Это позволяет нам быстро выявлять ошибки и обеспечивать стабильность кода.

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 12 – Результат автотестирования API приложения через GitHub Actions

**Результаты**: Автотестирование через GitHub Actions успешно завершилось, что подтверждает корректность работы приложения. Однако были выявлены предупреждения (warnings) типа DeprecationWarnings, связанные с библиотекой SHAP. Эти предупреждения не влияют на функциональность приложения, но требуют внимания в будущих обновлениях.

## Выводы по разделу

В ходе тестирования мы проверили все ключевые аспекты работы приложения, включая модульные, интеграционные, регрессионные, приемочные, нагрузочные и параметризованные тесты. Все тесты успешно прошли, что подтверждает стабильность и надежность приложения. Автотестирование через GitHub Actions позволило нам автоматизировать процесс проверки кода и быстро выявлять ошибки. В будущем мы планируем устранить предупреждения, связанные с библиотекой SHAP, чтобы повысить качество кода.

# Вывод по работе

В рамках данного проекта была проделана значительная работа по разработке сервиса для определения версии численной модели по параметрам системы. Основные этапы и результаты работы можно резюмировать следующим образом:

1. **Введение (Глава 1)**  
   Проект был успешно реализован с использованием современных технологий, таких как FastAPI, Streamlit и Docker. Это обеспечило универсальность и возможность работы в различных средах. Основная цель — предоставление интерфейса для выполнения задач машинного обучения через API — была достигнута.
2. **Для кого и зачем нужен данный сервис? (Глава 2)**  
   Сервис разработан для исследовательских целей, в частности для определения версии численной модели по параметрам физической системы. Он позволяет упростить анализ данных и ускорить процесс поиска сверхпроводников при высоких температурах.
3. **Модель машинного обучения (Глава 3)**  
   В качестве модели машинного обучения был выбран алгоритм XGBoost, который показал высокую точность в задачах классификации. Модель успешно обучена на данных и интегрирована в API, что позволяет пользователям получать предсказания и анализировать результаты.
4. **Реализация API для приложения машинного обучения (Глава 4)**  
   API разработан с использованием FastAPI, что обеспечило высокую производительность и простоту взаимодействия с моделью. Реализованы эндпоинты для получения метрик, построения графиков и выполнения предсказаний, что делает приложение удобным для использования.
5. **Реализация интерфейса через Streamlit (Глава 5)**  
   Интерфейс приложения создан с использованием Streamlit, что обеспечило интуитивно понятный и удобный графический интерфейс для пользователей. Это особенно полезно для тех, кто не имеет глубоких знаний в области машинного обучения.
6. **Контейнеризация с использованием Docker (Глава 6)**  
   Использование Docker позволило упаковать приложение и все его зависимости в изолированные контейнеры, что упростило процесс развертывания и обеспечило стабильность работы в различных средах.
7. **Запуск локального приложения (Глава 7)**  
   Локальный запуск приложения с использованием Docker Compose был успешно реализован. Это позволяет быстро развернуть проект на любом компьютере и начать работу с моделью машинного обучения.
8. **Альтернативный запуск приложения через Streamlit Cloud (Глава 8)**  
   Приложение также было развернуто в облаке через Streamlit Cloud, что сделало его доступным для широкой аудитории. Это особенно удобно для демонстрации проекта и совместной работы.
9. **Настройка автотестов через GitHub Actions и добавление линтера (Глава 9)**  
   Для обеспечения качества кода и стабильности приложения были настроены автотесты через GitHub Actions. Это позволило автоматически проверять код при каждом изменении в репозитории, а также использовать линтер (flake8) для поддержания стиля и чистоты кода.
10. **Тестирование приложения (Глава 10)**  
    Были проведены различные виды тестов, включая модульные, интеграционные, регрессионные, приемочные, нагрузочные и параметризованные тесты. Все тесты успешно прошли, что подтвердило стабильность и надежность приложения.

**Итоговый вывод**

Проект успешно прошел все этапы разработки и тестирования, показав высокую производительность и надежность. Использование современных технологий и подходов позволило создать универсальное и удобное решение для задач машинного обучения. В будущем планируется устранить предупреждения, связанные с библиотекой SHAP, и продолжить улучшение функциональности приложения. Также в глобальном смысле, последующее развитие проекта заключено в адаптации: под определение версии численной модели 0/2 и большего интерактивного использования с пользователем.

# P.s. Если лень читать…

**Структура проекта**:

AI\_phase\_diagram\_API/

├── main.py # FastAPI приложение

├── app.py # Streamlit интерфейс

├── streamlit\_app.py # Streamlit Cloud версия

├── xgboost\_model.pkl # Модель машинного обучения

├── Main\_Data.csv # Основные данные

├── test\_unit\_1.py # Модульный тест (загрузка модели)

├── test\_unit\_2.py # Модульный тест (загрузка данных)

├── test\_integration.py # Интеграционные тесты

├── test\_regression.py # Регрессионный тест

├── test\_priem.py # Приемочные тесты

├── test\_load.py # Нагрузочный тест

├── test\_param.py # Параметризованные тесты

├── Dockerfile # Dockerfile для FastAPI

├── Dockerfile.streamlit # Dockerfile для Streamlit

├── docker-compose.yml # Docker Compose конфигурация

├── requirements.txt # Зависимости Python

├── README.md # Описание проекта

├── predictions.json # Эталонные предсказания

├── test\_data.csv # Тестовые данные

└── media/ # Папка с изображениями

**Краткое руководство по запуску проекта:**

**1. Локальный запуск через Docker:**

1. Установите Docker и Docker Compose.
2. Клонируйте репозиторий:

*git clone https://github.com/Maddenss/AI\_phase\_diagram\_API.git*

*cd AI\_phase\_diagram\_API*

1. Соберите и запустите контейнеры:

*docker-compose up –build*

1. Откройте в браузере:
   * FastAPI: <http://localhost:8000/docs>
   * Streamlit: [http://localhost:8501](http://localhost:8501/)

**2. Запуск через Streamlit Cloud:**

1. Перейдите по ссылке: <https://ai-phase-diagram.streamlit.app/>.
2. Используйте интерфейс для загрузки данных и получения предсказаний.

**3. Запуск тестов:**

1. Убедитесь, что установлены зависимости:

*pip install -r requirements.txt*

1. Запустите тесты:

*pytest*

**4. Запуск автотестов через GitHub Actions:**

1. Автотесты запускаются автоматически при каждом push в ветку master или создании pull request.
2. Результаты тестов можно посмотреть в разделе Actions на GitHub.

**5. Что где лежит**

* API: main.py
* Интерфейс: app.py
* Модель: xgboost\_model.pkl
* Данные: Main\_Data.csv
* Тесты: test\_\*.py
* Docker: Dockerfile, Dockerfile.streamlit, docker-compose.yml

**Итог:**

* **Локально**: docker-compose up --build
* **Облако**: [Streamlit Cloud](https://ai-phase-diagram.streamlit.app/" \t "_blank)
* **Тесты**: pytest
* **Автотесты**: GitHub Actions (автоматически).