

Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности

Работа допущена к защите
Руководитель ОП
_____ А. В. Щукин
«_____» _____ 2025 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
РАБОТА БАКАЛАВРА
РАЗРАБОТКА СИСТЕМЫ ФОРМИРОВАНИЯ КОРРЕКТИРУЮЩИХ И
ПРЕДУПРЕДИТЕЛЬНЫХ ДЕЙСТВИЙ НА ОСНОВЕ ИЗМЕНЕНИЙ
РЕПОЗИТОРИЯ КОДА

по направлению подготовки 09.03.03 Прикладная информатика

Направленность (профиль) 09.03.03_03 Интеллектуальные инфокоммуникационные технологии

Выполнил
студент гр. 5130903/10301

А. Е. Ильчук

Руководитель
Старший преподаватель
ВШ ПИ

В. А. Пархоменко

Консультант
по нормоконтролю

Е. Е. Андрианова

Санкт-Петербург
2025

Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности

УТВЕРЖДАЮ

Руководитель ОП

_____ А. В. Щукин

«_____» _____ 2025г.

ЗАДАНИЕ

на выполнение выпускной квалификационной работы

студенту Ильчуку Александру Евгеньевичу гр. 5130903/10301

1. Тема работы: Разработка системы формирования корректирующих и предупредительных действий на основе изменений репозитория кода.
2. Срок сдачи студентом законченной работы: 17.05.2025.
3. Исходные данные по работе:
 - 3.1. An experience in automatically extracting CAPAs from code repositories: [Электронный ресурс]. URL: <https://arxiv.org/pdf/2212.09910> (дата обращения: 12.12.2024).
 - 3.2. A meta-analytical comparison of Naive Bayes and Random Forest for software defect prediction: [Электронный ресурс]. URL: https://link.springer.com/chapter/10.1007/978-3-031-35501-1_14#citeas (дата обращения: 12.12.2024).
 - 3.3. Examining the Success of an Open Source Software Project Analysing Its Repository: [Электронный ресурс]. URL: <https://doi.org/10.5281/zenodo.10046579> (дата обращения: 12.12.2024).
 - 3.4. Github API documentation: [Электронный ресурс]. URL: <https://docs.github.com/en/rest?apiVersion=2022-11-28> (дата обращения: 12.12.2024).
 - 3.5. PyGithub documentation: [Электронный ресурс]. URL: <https://pygithub.readthedocs.io/en/stable/> (дата обращения: 12.12.2024).
 - 3.6. The k-means Algorithm: A Comprehensive Survey and Performance Evaluation: [Электронный ресурс]. URL: <https://www.mdpi.com/2079-9292/9/8/1295> (дата обращения: 12.12.2024).
4. Содержание работы (перечень подлежащих разработке вопросов):

- 4.1. Исследование методов и средств формирования SARAs на основе изменений репозитория кода.
- 4.2. Проектирование системы формирования SARAs на основе изменений репозитория кода.
- 4.3. Реализация системы формирования SARAs на основе изменений репозитория кода.
- 4.4. Тестирование и апробация системы формирования SARAs на основе изменений репозитория кода.
5. Перечень графического материала (с указанием обязательных чертежей):
 - 5.1. Диаграмма вариантов использования.
 - 5.2. Архитектура разработанной программы.
6. Перечень используемых информационных технологий, в том числе программное обеспечение, облачные сервисы, базы данных и прочие сквозные цифровые технологии (при наличии):
 - 6.1. Python 3.9
 - 6.2. Git, GitHub
 - 6.3. pandas
 - 6.4. NumPy
 - 6.5. scikit-learn
 - 6.6. deep-forest
 - 6.7. Plotly
 - 6.8. Dash
7. Консультанты по работе:
 - 7.1. Ст. преподаватель ВШ ПИ, Е. Е. Андрианова (нормоконтроль).
8. Дата выдачи задания: 12.12.2024.

РЕФЕРАТ

На 67 с., 14 рис., 4 табл., 2 прил., 11 лист.

Ключевые слова: Репозиторий, Анализ данных, Метрики, Качество кода.

Тема выпускной квалификационной работы: «Разработка системы формирования корректирующих и предупредительных действий на основе изменений репозитория кода».

Объект исследования – процесс автоматизированного анализа истории коммитов в Git-репозиториях с целью выявления «рисковых» изменений и генерации корректирующих и предупреждающих рекомендаций (САРА).

Предмет исследования – методы извлечения и обработки метрик коммитов, алгоритмы кластеризации и классификации «рисковых» изменений, а также подходы к формированию рекомендаций САРА и их визуализации.

Цель работы – разработать модульную систему, способную автоматически собирать данные о коммитах из удалённых и локальных репозиториях, проводить статический анализ, обучать модель машинного обучения для предсказания риска коммита и формировать для каждого потенциально «рискового» изменения набор рекомендаций САРА.

В работе выполнены следующие основные этапы:

- А. Сбор и подготовка данных. Реализован класс `GitHubRepoAnalyzer`, который через `GitHub API` или локальный клон последовательно извлекает историю коммитов, собирает базовые метрики (число добавленных/удалённых строк, число изменённых файлов, время между коммитами) и дополняет их результатами статических анализаторов (`pylint`, `Bandit`, `ESLint`, `Checkstyle`).
- В. Генерация псевдометок и классификация. Для автоматической разметки выполнена кластеризация методом `KMeans`, разделяющая коммиты на «нормальные» и «аномальные» по признакам объёма и качества кода. На основе полученных меток обучается модель, обернутая в универсальный класс `CommitRiskModel`, позволяющий тестировать различные классификаторы (`LogisticRegression`, `RandomForest`, `SVM`, `LightGBM`, `DeepForest` и др.) без изменения кода. Модель предсказывает вероятность «рискового» коммита и вычисляет важность признаков.
- С. Формирование рекомендаций САРА. На основании вероятности риска и значений ключевых признаков (`lines_added`, `lines_deleted`, число статических предупреждений) формируется набор текстовых рекомендаций. При превы-

шении пороговых значений формируются предупреждающие сообщения и рекомендации по доработке (добавление тестов, рефакторинг, дополнительная проверка кода). Система автоматически создаёт Pull Request с файлом в формате Markdown, содержащим CAPA-рекомендации.

D. Визуализация. Разработан интерактивный дашборд на Dash + Plotly, включающий разделы:

- распределение метрик коммитов (lines_added, lines_deleted, число файлов),
- анализ риска (распределение коммитов по вероятностям, важность признаков),
- активность авторов (графики среднего риска по каждому разработчику),
- таблица коммитов с рекомендациями CAPA и возможностью фильтрации по проекту, автору и дате.

В качестве тестовых примеров использовались несколько реальных учебных и рабочих репозиториях. Проведено сравнительное тестирование классификаторов на основе метрик Precision, Recall, F1-score и ROC-AUC; для каждого проекта определены оптимальные алгоритмы.

Система автоматизирует полный цикл анализа коммитов и выдачи рекомендаций CAPA, сокращая время обратной связи, улучшая качество кода и снижая число «рисковых» изменений.

ABSTRACT

On 67 pages, 14 figures, 4 tables, 2 appendices, 11 listings

Keywords: Repository, Data analysis, Metrics, Code quality

The subject of the graduate qualification work is «Developing tools for artificially extracting corrective preventive actions from a code repository using analysis tools».

The object of research is the process of automated analysis of commit history in Git repositories to identify “risky” changes and generate corrective and preventive (CAPA) recommendations.

The subject of research includes methods for extracting and processing commit metrics, clustering and classification algorithms for “risky” changes, as well as approaches to forming CAPA recommendations and their visualization.

The aim of this work is to develop a modular system capable of automatically collecting commit data from remote and local repositories, performing static analysis, training a machine learning model to predict commit risk, and generating a set of CAPA recommendations for each potentially “risky” change.

The main stages completed in this work are as follows:

- A. **Data Collection and Preparation.** The `GitHubRepoAnalyzer` class was implemented, which uses the GitHub API or a local clone to sequentially retrieve commit history, gather basic metrics (number of lines added/deleted, number of files changed, time between commits), and augment them with results from static analyzers (pylint, Bandit, ESLint, Checkstyle).
- B. **Pseudo-Label Generation and Classification.** For automated labeling, clustering with the KMeans algorithm was performed, splitting commits into “normal” and “anomalous” based on code volume and quality metrics. On the basis of these pseudo-labels, a model is trained using the universal `CommitRiskModel` class, which allows testing various classifiers (LogisticRegression, RandomForest, SVM, LightGBM, DeepForest, etc.) without modifying the code. The model predicts the probability of a commit being “risky” and calculates feature importances.
- C. **CAPA Recommendation Generation.** Based on the predicted risk probability and key feature values (`lines_added`, `lines_deleted`, number of static warnings), a set of textual recommendations is formed. When threshold values are exceeded, warning messages and improvement suggestions are generated (adding tests, refactoring, additional code review). The system automatically creates a Pull Request with a Markdown file containing the CAPA recommendations.
- D. **Visualization.** An interactive dashboard was developed using Dash + Plotly, which includes:
 - distribution of commit metrics (`lines_added`, `lines_deleted`, number of files),
 - risk analysis (distribution of commits by probability, feature importance),
 - author activity (average risk charts per developer),
 - a commit table with CAPA recommendations and filtering options by project, author, and date.

Several real educational and work repositories were used as test examples. Comparative testing of classifiers was conducted using Precision, Recall, F1-score, and ROC-AUC metrics; optimal algorithms were identified for each project.

The system automates the full cycle of commit analysis and CAPA recommendation generation, reducing feedback time, improving code quality, and decreasing the number of “risky” changes.

The object of research is the process of automated analysis of commit history in Git repositories to identify “risky” changes and generate corrective and preventive (CAPA) recommendations.

The subject of research includes methods for extracting and processing commit metrics, clustering and classification algorithms for “risky” changes, as well as approaches to forming CAPA recommendations and their visualization.

The aim of this work is to develop a modular system capable of automatically collecting commit data from remote and local repositories, performing static analysis, training a machine learning model to predict commit risk, and generating a set of CAPA recommendations for each potentially “risky” change.

The main stages completed in this work are as follows:

- A. Data Collection and Preparation. The `GitHubRepoAnalyzer` class was implemented, which uses the GitHub API or a local clone to sequentially retrieve commit history, gather basic metrics (number of lines added/deleted, number of files changed, time between commits), and augment them with results from static analyzers (`pylint`, `Bandit`, `ESLint`, `Checkstyle`).
- B. Pseudo-Label Generation and Classification. For automated labeling, clustering with the `KMeans` algorithm was performed, splitting commits into “normal” and “anomalous” based on code volume and quality metrics. On the basis of these pseudo-labels, a model is trained using the universal `CommitRiskModel` class, which allows testing various classifiers (`LogisticRegression`, `RandomForest`, `SVM`, `LightGBM`, `DeepForest`, etc.) without modifying the code. The model predicts the probability of a commit being “risky” and calculates feature importances.
- C. CAPA Recommendation Generation. Based on the predicted risk probability and key feature values (`lines_added`, `lines_deleted`, number of static warnings), a set of textual recommendations is formed. When threshold values are exceeded, warning messages and improvement suggestions are generated (adding tests, refactoring, additional code review). The system automatically creates a Pull Request with a Markdown file containing the CAPA recommendations.
- D. Visualization. An interactive dashboard was developed using `Dash` + `Plotly`, which includes:
 - distribution of commit metrics (`lines_added`, `lines_deleted`, number of files),

- risk analysis (distribution of commits by probability, feature importance),
- author activity (average risk charts per developer),
- a commit table with CAPA recommendations and filtering options by project, author, and date.

Several real educational and work repositories were used as test examples. Comparative testing of classifiers was conducted using Precision, Recall, F1-score, and ROC-AUC metrics; optimal algorithms were identified for each project.

The system automates the full cycle of commit analysis and CAPA recommendation generation, reducing feedback time, improving code quality, and decreasing the number of “risky” changes.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	11
1 ИССЛЕДОВАНИЕ МЕТОДОВ И СРЕДСТВ ФОРМИРОВАНИЯ SARAS НА ОСНОВЕ ИЗМЕНЕНИЙ РЕПОЗИТОРИЯ КОДА	13
1.1. Современные подходы к формированию SARAS на основе анализа репозитория кода.....	13
1.2. Системы управления SARAS и интеграция в процессы разработки	13
1.3. Методы анализа коммитов и обучение на данных репозитория	14
1.3.1. Предсказание дефектных коммитов.....	14
1.3.2. Классификация коммитов по типам работ.....	15
1.4. Инструменты и системы автоматизации SARAS	16
1.5. Диаграмма вариантов использования.....	19
1.6. Выводы	21
2 ПРОЕКТИРОВАНИЕ СИСТЕМЫ ФОРМИРОВАНИЯ SARAS НА ОС- НОВЕ ИЗМЕНЕНИЙ РЕПОЗИТОРИЯ КОДА	21
2.1. Требования к системе	21
2.2. Нефункциональные требования	22
2.3. Выбор технологий и инструментов	23
2.4. Архитектура системы	24
2.4.1. Компонент сбора данных.....	25
2.4.2. Компонент статического анализа и формирования метрик	25
2.4.3. Компонент классификации.....	25
2.4.4. Компонент генерации рекомендаций	26
2.4.5. Веб-приложение визуализации на Dash.....	26
2.5. Диаграмма классов	26
2.6. Заключение	28
3 РЕАЛИЗАЦИЯ СИСТЕМЫ ФОРМИРОВАНИЯ SARAS НА ОСНОВЕ ИЗМЕНЕНИЙ РЕПОЗИТОРИЯ КОДА	29
3.1. Извлечение и обработка данных из GitHub	30
3.1.1. Аутентификация и инициализация подключения	30
3.1.2. Получение списка коммитов.....	31
3.1.3. Извлечение деталей коммита и подсчёт метрик	32
3.2. Интеграция модели CommitRiskModel	34
3.2.1. Постановка задачи и необходимость генерации псевдометок.....	35
3.2.2. Код генерации псевдометок.....	35
3.2.3. Выбор и обработка признаков	36

3.2.4. Обучение классификатора.....	37
3.2.5. Предсказание риска и вероятностей	37
3.2.6. Интерпретация модели — важность признаков	38
3.3. Реализация панели визуализации на фреймворке Dash.....	38
3.3.1. Структура и организация интерфейса.....	39
3.3.2. Пример построения гистограммы с использованием plotly.express	39
3.3.3. Динамическое обновление интерфейса и фильтрация данных.....	40
3.3.4. Генерация и отображение рекомендаций CAPA.....	41
3.4. Интеграция компонентов в единую систему	42
3.5. Выводы	43
4 ТЕСТИРОВАНИЕ СИСТЕМЫ ФОРМИРОВАНИЯ CAPAS НА ОСНОВЕ ИЗМЕНЕНИЙ РЕПОЗИТОРИЯ КОДА	43
4.1. Введение.....	43
4.2. Модульное тестирование	44
4.3. Мутационное тестирование	46
4.3.1. Итоги прогона	47
4.3.2. Результат мутационного тестирования	47
4.3.3. Результаты после исправлений	48
4.4. Нагрузочное тестирование	48
4.5. Набор датасетов.....	49
4.6. Сравнение моделей классификации.....	51
4.6.1. Выводы	52
4.7. Апробация и интерпретация результатов.....	52
4.7.1. Описание страницы рекомендаций по коммитам.....	56
4.8. Сравнение экспертных меток и рекомендаций, выданных моделью	57
4.8.1. Рекомендации экспертов	58
4.8.2. Рекомендации модели	59
4.8.3. Сравнение рекомендаций экспертов и модели.....	60
4.9. Выводы	60
ЗАКЛЮЧЕНИЕ	63
СЛОВАРЬ ТЕРМИНОВ.....	65
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	65
БИБЛИОГРАФИЧЕСКИЙ СПИСОК.....	66
ПРИЛОЖЕНИЕ 1. ИСХОДНЫЙ КОД РАЗРАБОТАННОГО РЕШЕНИЯ ...	68
ПРИЛОЖЕНИЕ 2. СРАВНЕНИЕ МОДЕЛЕЙ КЛАССИФИКАЦИИ ДЛЯ РАЗНЫХ ПРОЕКТОВ	91

ВВЕДЕНИЕ

Актуальность исследования. Современные разработки программного обеспечения (далее - ПО) становятся всё более сложными, требуя высоких стандартов качества и точного контроля за процессом разработки. В условиях глобализации и быстрого развития технологий компании стремятся не только создавать качественные продукты, но и эффективно управлять процессом их создания. Однако существующие подходы к анализу и управлению качеством ПО требуют значительных временных и человеческих ресурсов. Методы анализа данных и машинного обучения, которые уже зарекомендовали себя в смежных областях, могут быть использованы для автоматизации контроля качества и выявления проблем на ранних этапах. Одной из ключевых задач в этой области является анализ данных из репозитория исходного кода, таких как GitHub.

Коммиты в репозиториях содержат важную информацию о внесённых изменениях: количество добавленных и удалённых строк кода, изменённые файлы, временные интервалы между изменениями. Анализ этих данных позволяет выявить потенциальные отклонения от нормального процесса разработки и предложить корректирующие и предупреждающие действия (далее - САРА). Несмотря на широкий спектр существующих инструментов для анализа данных из репозитория, большинство из них либо недостаточно автоматизированы, либо не позволяют выявлять комплексные закономерности в данных.

Цель исследования - разработка системы, которая позволит автоматизировать процесс анализа коммитов и извлечения САРА на основе методов машинного обучения и кластеризации.

Задачи исследования:

- Провести обзор существующих методов анализа данных из репозитория кода.
- Изучить применимость методов кластеризации и алгоритмов машинного обучения для анализа коммитов.
- Разработать систему для автоматического извлечения данных о коммитах из нескольких репозиториях GitHub.
- Реализовать механизм выявления аномалий и классификации коммитов на основе предложенных методов.
- Создать интерактивный дашборд для визуализации результатов анализа.
- Оценить эффективность предложенного подхода на реальных данных.

Подробнее актуальность исследования и обзор методов рассмотрены в разделе 1.1.

Таким образом, исследование направлено на решение задачи повышения эффективности управления качеством программного обеспечения за счёт использования современных технологий анализа данных. Предложенная система должна не только автоматизировать процесс анализа данных, но и предоставлять разработчикам полезные рекомендации для улучшения качества кода и предотвращения потенциальных проблем.

ГЛАВА 1. ИССЛЕДОВАНИЕ МЕТОДОВ И СРЕДСТВ ФОРМИРОВАНИЯ CAPAS НА ОСНОВЕ ИЗМЕНЕНИЙ РЕПОЗИТОРИЯ КОДА

1.1. Современные подходы к формированию CAPA на основе анализа репозитория кода

Корректирующие и предупреждающие действия CAPA представляют собой ключевой элемент систем управления качеством, направленных на выявление и устранение дефектов, а также предупреждение их повторного возникновения [11]. Изначально применяемые в регулируемых отраслях (медицина, фармацевтика и т.д.), концепции CAPA становятся актуальными и для разработки программного обеспечения, где цель сводится к автоматизации обнаружения проблем в коде и предложению мер по их исправлению или предотвращению. В контексте анализа изменений в репозиториях кода задача формирования CAPA сводится к систематическому анализу истории коммитов, метрик и паттернов изменений с целью выявления аномалий и генерации рекомендаций для разработчиков.

Существует несколько направлений и инструментов, применимых к этой задаче. Одни ориентированы на управление качеством как таковым (системы QMS и CAPA-менеджмента), другие – на технический анализ исходного кода и истории версий (статический анализ, ЛТ-предсказание дефектов, классификация коммитов). Современные подходы активно используют методы машинного обучения и искусственного интеллекта для выявления закономерностей в репозиториях и автоматической генерации CAPA [2][13]. В этой главе рассматриваются существующие модели и инструменты, а также их сравнение по критериям автоматизации, интеграции в CI/CD, применимости к истории коммитов и интерпретируемости результатов.

1.2. Системы управления CAPA и интеграция в процессы разработки

В классических системах контроля качества (например, ISO 9001, FDA 21 CFR Part 820) CAPA оформляются документально и отслеживаются средствами QMS. Такие системы (Quality Management Systems) обеспечивают формализацию процесса: сбор инцидентов, расследование причин, планирование и выполнение действий, верификацию эффективности. Большинство коммерческих решений по CAPA (Qualityze, MasterControl, SimplerQMS и т.д.) предлагают удобные ин-

терфейсы для заполнения карточек CARA и управления ими, но они не заточены под анализ кода или репозитория. Автоматизация в этих системах ограничена триггерами (например, создание CARA на основе записи о сбое теста или жалобы), и интеграция с процессами CI/CD чаще всего осуществляется через ручные интерфейсы или API. Подходы такого рода обеспечивают качественный учет проблем и статус выполнения мер, однако они не анализируют непосредственно изменения в коде и не извлекают CARA автоматически на основе метрик репозитория [11].

Напротив, современные инструментальные решения в области разработки ПО стремятся к раннему обнаружению проблем в процессе написания кода. Сюда относятся системы статического анализа кода (SonarQube, CodeQL, Coverity), инструменты анализа сборок и покрытий (Codecov), а также инструменты анализа истории версий и производительности (Pluralsight Flow, CodeScene и др.). Такие инструменты обычно легко интегрируются в конвейер CI/CD (GitLab CI, Jenkins, GitHub Actions и др.), автоматизированно собирают метрики кода и могут генерировать отчеты с найденными дефектами или подозрениями. Например, SonarQube при подключении к Git-серверу автоматически анализирует каждый коммит или пулл-реквест на наличие проблем (code smells, багов, уязвимостей) и может предлагать меры по их устранению. Выходные данные статических анализаторов часто имеют стандартизированный формат (например, SARIF – Static Analysis Results Interchange Format), что облегчает интеграцию и агрегирование результатов [11]. Недостатком здесь является то, что такие инструменты обычно фокусируются на анализе текущего состояния кода, но не делают выводов о процессе разработки в целом или о трендах аномалий в истории коммитов. Тем не менее, их отчеты служат основой для корректирующих действий (исправление найденных дефектов) и косвенно могут стимулировать превентивные меры по устранению проблем.

1.3. Методы анализа коммитов и обучение на данных репозитория

1.3.1. Предсказание дефектных коммитов

Одним из популярных подходов в анализе изменений является Just-In-Time Software Defect Prediction (JIT-SDP), направленный на обнаружение потенциально «багогенерирующих» коммитов до интеграции изменений. Модели JIT-SDP строятся на исторических данных о прошлых коммитах (метриках изменений, метаданных, сообщениях к коммитам) и обучаются классифицировать новые

коммиты как безопасные или потенциально дефектные. Например используя метод одно-классовой классификации для идентификации аномалий среди коммитов [12]. Для её реализации модель обучается на «нормальных» (не дефектных) коммитах и затем помечает отклонения как подозрительные. Эксперименты показали, что при высоком дисбалансе классов (мало баговых коммитов относительно нормальных) одно-классовые алгоритмы (One-Class SVM, Isolation Forest и т.д.) превосходят традиционные классификаторы по точности обнаружения дефектных изменений [12]. JIT-SDP можно встроить непосредственно в систему контроля версий: модель принимает данные о новом коммите при попытке его зафиксировать и возвращает прогноз (например, в виде уведомления разработчику), что позволяет принять коррективные действия (дополнительная проверка, тестирование) до слияния. Такие подходы полностью автоматизированы (после первоначального обучения) и могут работать в CI/CD без вмешательства человека. Однако они часто ограничены точностью модели и могут давать ложные срабатывания, поэтому вопрос интерпретируемости результатов здесь актуален: разработчикам важно понимать, какие особенности коммита вызвали подозрение. Для повышения интерпретируемости предлагаются методы объяснения предсказаний (например, SHAP/ LIME или специальные эксплейнеры для JIT-SDP), хотя это выходит за рамки базовых моделей [12].

1.3.2. Классификация коммитов по типам работ

Другой распространённый подход – классификация коммитов по видам технических действий (например, категориальное разделение на корректирующие, адаптивные, совершенствующие задачи). Этот метод чаще используется для анализа уже сделанных изменений с целью обзора тенденций и построения рекомендаций. В литературе выделяют три классических типа изменений: Corrective (исправление дефектов), Adaptive (обработка требований/увеличение функциональности) и Perfective (улучшение производительности/рефакторинг) [13]. Основные характеристики таких моделей: они используют признаковые описания коммитов (количество строк, слова из сообщений, данные о затронутых файлах) и типичные алгоритмы машинного обучения (Decision Tree, Random Forest, Naive Bayes, нейросети) [13]. Результаты таких классификаторов помогают в формировании превентивных мер: например, если обнаруживается, что большое число коммитов носит корректирующий характер из-за недостаточного тестового покрытия, си-

стема может рекомендовать усилить модульное тестирование. В последние годы к этой задаче привлекаются большие языковые модели. Например, GPT-3 может в режиме zero-/few-shot эффективно классифицировать сообщения коммитов по категориям технической поддержки [13]. В частности, достигается ~75% точности классификации по трём категориям [13]. Это свидетельствует о высокой автоматизируемости такого подхода и возможности использования предобученных моделей, однако интерпретируемость итоговых меток, особенно при применении LLM остается проблемой.

Классификаторы коммитов в целом демонстрируют высокую применимость к истории репозитория, поскольку анализируют каждое изменение во временной последовательности. Они хорошо интегрируются в CI/CD и инструменты мониторинга (через плагины GitHub/GitLab) – фактически после фиксации коммита производится моментальная классификация с логированием результата. Однако такие системы требуют обширных размеченных данных для обучения (или хорошо сформулированных правил) и могут не учесть все контексты проекта, что снижает их точность для разных проектов [13][15]. Тем не менее, при правильной настройке они позволяют формировать САРА как в корректирующей, так и превентивной части: например, классифицируя «адаптивные» коммиты, можно обнаружить систематические изменения в требованиях и заранее планировать архитектурные доработки.

1.4. Инструменты и системы автоматизации САРА

На практике для генерации САРА в области разработки ПО применяются комбинированные решения, объединяющие анализ кода и процесс управления. Некоторые примеры:

- Встроенные боты и скрипты в репозиторий.

Примером является система TOM (Theoretically Objective Measurements), описанная Bugaenko и соавторами [2]. Для её использования необходимо лишь добавить бота @Osara в репозиторий. Он автоматически собирает метрики по каждому коммиту и на их основе предлагает САРА. Такой подход максимально автоматизирован и интегрируется через механизмы GitHub (было реализовано создание pull request с рекомендациями САРА). Информация о том, какие действия требует каждый коммит, хранится вместе

с кодом, что повышает прозрачность процесса. Хотя и система была описана, готового решения по @0сара нет в открытом доступе.

- Плагины и дополнения к системам CI/CD.

Многие CI-системы (GitLab, Jenkins, GitHub Actions) поддерживают установку плагинов, выполняющих анализ коммитов или статический анализ. Например, SonarQube имеет плагин SonarScanner для GitLab CI, а GitHub Actions – для автоматического запуска анализа при каждом PR. Некоторые сервисы (например, bugasura.io) предлагают «AI-инспектора», который отслеживает журналы сборок и автоматически сигнализирует о паттернах отказов. Такие инструменты часто обеспечивают интеграцию в процесс разработки и могут генерировать тикеты или отчеты САРА. Форматы вывода обычно совместимы со стандартными трекерами (JIRA, GitHub Issues), что облегчает стандартизацию.

- Инструменты анализа pull-реквестов и метаданных.

Существуют системы, которые обучены классифицировать Pull Request как «корректирующий» или «не корректирующий» [2]. По результатам анализа PR или истории веток такие инструменты могут «присваивать» САРА к прошедшим исправлениям и накапливать статистику. Это позволяет накапливать базу знаний о типичных рекомендациях и соотносить их с шаблонами изменений. Однако подобные решения пока редко встречаются в свободном доступе и требуют развитой инфраструктуры данных [2].

- Статические анализаторы с функцией САРА.

Ряд современных SAST-инструментов (Fortify, Checkmarx, SonarQube) начинают включать в отчеты «рекомендации» помимо указания на проблему. Например, при обнаружении уязвимости система может автоматически предложить типовую корректирующую меру (обновить библиотеку, изменить код), что можно рассматривать как САРА в узком смысле. Хотя это скорее рекомендации на уровне кода, а не на уровне процесса разработки, их вывод можно экспортировать для дальнейшей автоматической обработки. Интеграция в CI/CD у подобных систем хорошо налажена, а результаты достаточно интерпретируемы (описаны правила, чек-листы).

Таким образом, современные инструменты сходятся на том, что интеграция в процессы CI/CD и автоматизация – ключевое требование. Боты и плагины обеспечивают автоматический сбор данных из репозитория и формирование САРА-рекомендаций без участия человека, хотя зачастую требуют первоначальной

настройки моделей и правил. Статические анализаторы обеспечивают проверку качества кода по стандартным правилам, и их выходы можно использовать для инициирования CAPA в QMS (например, через настройку связей с Jira или другой системой учета).

Таблица 1.1

Сравнение инструментов формирования CAPA

Решение	Инструмент	Открытый исходный код	CI/CD интеграция	Анализ истории коммитов	Статический анализ кода	Автовыдача CAPA	Формат CAPA
MasterControl CAPA	QMS платформа	Нет	Нет	Нет	Нет	Осуществляется вручную	Полный аудит
CommitGuru	JIT-Defect Prediction	Да	Нет	Да	Частично	Да	Только степень риска, вероятность дефекта
@0capa (ТОМ)	Бот-анализ репозитория	Частично	Да	Да	Линтеры	Автоматическая Pull-Request	Текстовые шаблонные советы
Bugasura.io	CI-logs AI-inspector	Нет	Да	Нет	ESLint	Автоматическая Issue	Текстовые рекомендации при сбоях
Разрабатываемое решение	ML-анализ коммитов и статический анализ	Да	Да (интеграция через CI/CD пайплайны)	Да (анализ истории коммитов)	Да (Pylint, Checkstyle, ESLint и др.)	Автоматическая генерация рекомендаций CAPA	Подробные рекомендации с приоритетами и причинами риска

Ниже — текстовое описание каждого из выбранных критериев сравнения инструментов формирования CAPA:

Подход. Определяет алгоритмическую или организационную основу, на которой построен инструмент. QMS-платформа (MasterControl CAPA) реализует классический жизненный цикл CAPA: сбор инцидента, расследование, план действий, проверка эффективности. JIT-Defect Prediction (CommitGuru) — модель, обученная на истории коммитов, предсказывает «риск» в момент фиксации изменений. Бот-анализ репозитория (@0capa/ТОМ) — GitHub-App, который на каждый пуш автоматически собирает метрики и создаёт PR с рекомендациями. CI-logs AI-inspector (Bugasura.io) — анализирует логи CI-пайплайна и на их основе формирует тикеты с советами.

Открытый исходный код. Показывает, можно ли самостоятельно посмотреть и модифицировать реализацию: Да (CommitGuru, @0capa) — инструменты с открытым исходным кодом, которые можно хостить и дорабатывать под свои

нужды. Нет (MasterControl, Bugasura.io) — закрытые решения без публичного репозитория.

Интеграция в CI/CD. Наличие штатных плагинов/webhook'ов для автоматического запуска вместе с билдом или PR: Да (@0capa, Bugasura.io) — инструмент подключается к GitHub Actions/Jenkins/GitLab CI и срабатывает на пуш/PR; Частичная (CommitGuru) — имеет API-доступ, но не поставляется в виде готового CI-плагина; Нет (MasterControl) — работает вне пайплайна, задачи заводятся вручную.

Анализ истории коммитов. Учитывает ли инструмент тренды и метрики прошлых коммитов, а не только текущее состояние кода: Да (CommitGuru, @0capa) — применяет ЛТ-модели или паттерны на всей истории изменений; Нет (MasterControl, Bugasura.io) — фокусируется только на одном коммите или логах сборки.

Статический анализ кода. Использует ли линтеры/SAST-сканеры для поиска дефектов: Линтеры (@0capa подключает pylint/ESLint/Checkstyle); ESLint (Bugasura.io анализирует только JS-файлы); Нет (MasterControl, CommitGuru) — не выполняют статический анализ.

Выдача CAPA. Как инструмент оформляет рекомендации разработчикам. Вручную (MasterControl) — CAPA создаётся пользователем в рамках QMS-процесса; Авто Pull-Request (@0capa) — бот сам открывает PR с файлом CapaRecommendations.md; Авто Issue (Bugasura.io) — генерирует тикет в GitHub Issues при падении сборки; Нет (CommitGuru) — выдаёт лишь числовой «risk-score», без конкретных рекомендаций.

Детализация рекомендаций. Уровень подробностей и формат советов: Полный аудит (MasterControl) — многословные отчёты, шаблоны расследования, вложения; Вероятность дефекта (CommitGuru) — только процент или оценка риска, без пояснений; Шаблонные текстовые советы (@0capa) — заранее подготовленный набор рекомендаций; Текстовые рекомендации при сбоях (Bugasura.io) — описание ошибки из логов и указание на шаги CI.

1.5. Диаграмма вариантов использования

Для формализации функциональных требований к системе построим диаграмму вариантов использования. Она позволяет наглядно показать, какие внешние акторы взаимодействуют с системой и какие сервисы они должны вызывать. Основными акторами системы выступят разработчики и проектные менеджеры. Разработчик

может использовать систему для просмотра результатов анализа в интерактивном дашборде, а также изучить автоматически выданные под коммиты рекомендации, полученные прямыми в репозиторий с помощью pull-request. Проектный менеджер может отслеживать общие метрики качества, такие как, например, средний объём изменений, число багфиксов, количество предупреждений, изучать тренды на дашборде, в результате чего принимать стратегические решения по персоналу или изменению процесса разработки. Github Api - инициализирует весь конвейер: Извлекает историю коммитов и вычисляет базовые метрики (добавленные/удалённые строки, число затронутых файлов, интервалы между коммитами). Система запускает статический анализ кода (pylint, ESLint, Checkstyle и т.п.) и сохраняет предупреждения, объединяет все метрики в единый набор признаков и передаёт их модели машинного обучения. По результатам классификации формируются рекомендации для аномальных коммитов. GitHub Api создаёт в репозитории pull-request с файлом CapaRecommendations.md, содержащим эти рекомендации.

Таким образом, диаграмма Use Case (см. Рис. 1.1) будет содержать три «актора» (Разработчик, Менеджер, GitHub API) и 7 основных прецедентов: Извлечение истории коммитов, статический анализ кода, обучение модели, выявление аномалий, выдача рекомендаций, создание pull-request'а с рекомендациями, просмотр дашборда.

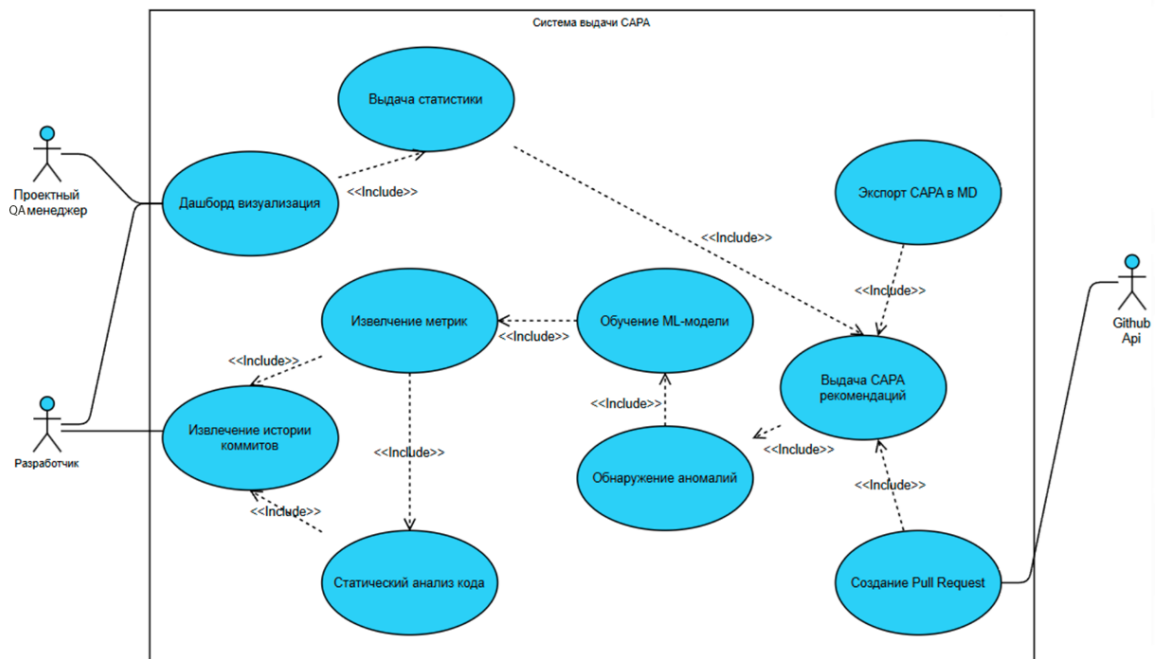


Рис.1.1. Use Case-диаграмма системы автоматического формирования CAPA

1.6. Выводы

Современные практики формирования САРА на основе анализа репозитория кода объединяют строгие процедуры управления качеством и динамическую ML-аналитику истории изменений. Классический цикл САРА — сбор инцидентов, диагностика, планирование и проверка эффективности мер — дополняется ЛТ-предсказанием дефектов и классификацией коммитов на основе моделей машинного обучения, а также «бот-аналитикой» (ТОМ, GitHub Apps), когда система сама собирает метрики и создаёт pull-request'ы с рекомендациями. При выборе решения критичны автоматизируемость, интеграция в CI/CD, учёт истории коммитов и интерпретируемость результатов. Статические анализаторы (SonarQube) надёжно автоматизируются, но не отслеживают эволюцию проекта, а ML-модели улавливают сложные паттерны в истории, но требуют обучения и часто выступают «чёрным ящиком». Наименее работоспособным оказывается чистый QMS-софт без аналитики кода. Лучшие практики — это гибридный подход: статический анализ, ML-классификация и интеграция с баг-трекером для автосоздания задач САРА.

ГЛАВА 2. ПРОЕКТИРОВАНИЕ СИСТЕМЫ ФОРМИРОВАНИЯ SARAS НА ОСНОВЕ ИЗМЕНЕНИЙ РЕПОЗИТОРИЯ КОДА

2.1. Требования к системе

Исходя из информации, полученной при анализе информационных источников были установлены следующие требования к разрабатываемой системе:

- Система должна автоматически извлекать историю коммитов и связанные метрики из репозитория GitHub (авторы, даты, количество добавленных/удалённых строк, изменения в файлах).
- Необходимо выполнять статический анализ кода на разных языках (например, Python, JavaScript, Java). Система должна запускать такие инструменты, как pylint, bandit для Python, eslint для JavaScript, checkstyle для Java, чтобы определить проблемы качества и уязвимости. Результаты анализа (количество предупреждений, метрики сложности и т.п.) обогащают данные коммитов.
- На основании извлечённых признаков система должна определять «аномальные» изменения (например, дефектно-опасные коммиты). Здесь применяются

методы Just-In-Time предсказания дефектов: обучение моделей машинного обучения для классификации коммита по риску [14].

- На основе обнаруженных аномалий система формирует корректирующие и предупреждающие действия. Это могут быть комментарии или автоматически созданные задачи/PR в GitHub с рекомендациями по улучшению процесса (например, «увеличить покрытие тестами», «провести рефакторинг сложного модуля»). Функционал сходен с подходом ТОМ: бот анализирует метрики и создает issue с описанием аномалий и действий [2].
- Система должна представлять результаты анализа в наглядном виде. Планируется интерактивный дашборд с графиками активности коммитов, распределением метрик и отмеченными аномальными событиями. Пользователь сможет просматривать тренды изменений и статусы САРА.

2.2. Нефункциональные требования

- Масштабируемость: решение должно уметь обрабатывать большие репозитории (сотни коммитов) и несколько проектов параллельно. Архитектура должна легко адаптироваться к новым языкам и инструментам. Развитием этой идеи будет наличие в проекте класса RepoAnalyzer, который будет построен универсально: для каждого типа файла в репозитории будет отображение на список анализаторов (например, '.py': [PylintAnalyzer, BanditAnalyzer], '.js': [ESLintAnalyzer], '.java': [CheckstyleAnalyzer]). Добавление нового языка сводится к добавлению пары (расширение → анализатор) в конфигурацию. Таким образом поддерживается гибкость и масштабируемость системы.
- Отказоустойчивость: система должна корректно обрабатывать неуспешные запросы к API (повторять/логировать ошибки) и обеспечивать целостность данных.
- Интерпретируемость: интерфейс (дашборд) должен быть интуитивно понятным для пользователя и информативным. Информация на графиках должна быть исчерпывающей для понимания процесса разработки. Предложенные САРА должны чётко определять найденную проблему.
- Безопасность: для взаимодействия с GitHub используются официальные API и безопасное хранилище токенов.
- Интеграция: Механизм выдачи САРА для разработчиков оформляется в виде GitHub-Pull Request, что упростит интеграцию с процессом разработки.

2.3. Выбор технологий и инструментов

Основным языком разработки был выбран Python. Благодаря богатой экосистеме библиотек для анализа данных и машинного обучения. В частности, pandas обеспечивает удобную обработку табличных данных, scikit-learn – основа для обучения моделей классификации. Альтернативы (например, C++ или Java) имеют аналогичные библиотеки, но Python позволяет быстро прототипировать и интегрировать различные компоненты (API, ML, веб).

Для доступа к данным репозитория (коммиты, файлы, статистика изменений). Используется GitHub API. Альтернатива – локальное клонирование через git CLI, но API быстрее предоставляет агрегированные данные (например, статистику добавлений/удалений).

Dash + Plotly: Выбран для разработки веб-интерфейса/дашборда. Dash – высокоуровневый фреймворк на Python, построенный поверх Flask и React, облегчает создание интерактивных приложений с помощью Python-кода. Dash даёт большую гибкость в настройке интерфейса и графиков. Plotly обеспечивает красивые и интерактивные графики без необходимости писать JavaScript.

pandas: Де-факто стандарт для работы с данными в Python. Позволяет быстро агрегировать и трансформировать данные коммитов перед подачей в модели. Альтернативы: использовать NumPy напрямую или базы данных, но pandas удобнее для аналитики и визуализации.

KMeans: Используется для кластеризации коммитов и определения «границы аномалии». Алгоритм прост и хорошо масштабируется. Идея – отсортировать коммиты по удалённости до центра кластера и пометить самые далекие как аномалии. Альтернативы могли быть методы на основе плотности (DBSCAN) или статистического анализа, но KMeans достаточно для первоначального порога, при этом не требуется ввод дополнительных сложных гиперпараметров.

Статические анализаторы: Pylint (строгий линтер для Python), Bandit (фокус на уязвимости Python), ESLint (статический анализ JS/TS), Checkstyle (Java). Эти инструменты бесплатны, широко используются в индустрии и генерируют стандартизованный вывод, удобный для парсинга. Например, альтернативный SonarQube/CodeQL требовал бы более тяжёлой инфраструктуры, тогда как лёгкие линтеры легко интегрировать в пайплайн. Выбор pylint/ESLint обусловлен их широкой поддержкой сообществом и гибкостью настроек.

2.4. Архитектура системы

Проектируемая система реализована с учётом модульного, расширяемого и масштабируемого подхода. Архитектура разделена на несколько ключевых компонентов (см Рис. 2.1), которые взаимодействуют последовательно, обеспечивая надёжный сбор, обработку, анализ и визуализацию данных из репозиториях GitHub. Проектируемая система имеет модульную архитектуру с элементами сервисно-ориентированного подхода и частично реализует клиент-серверную модель.

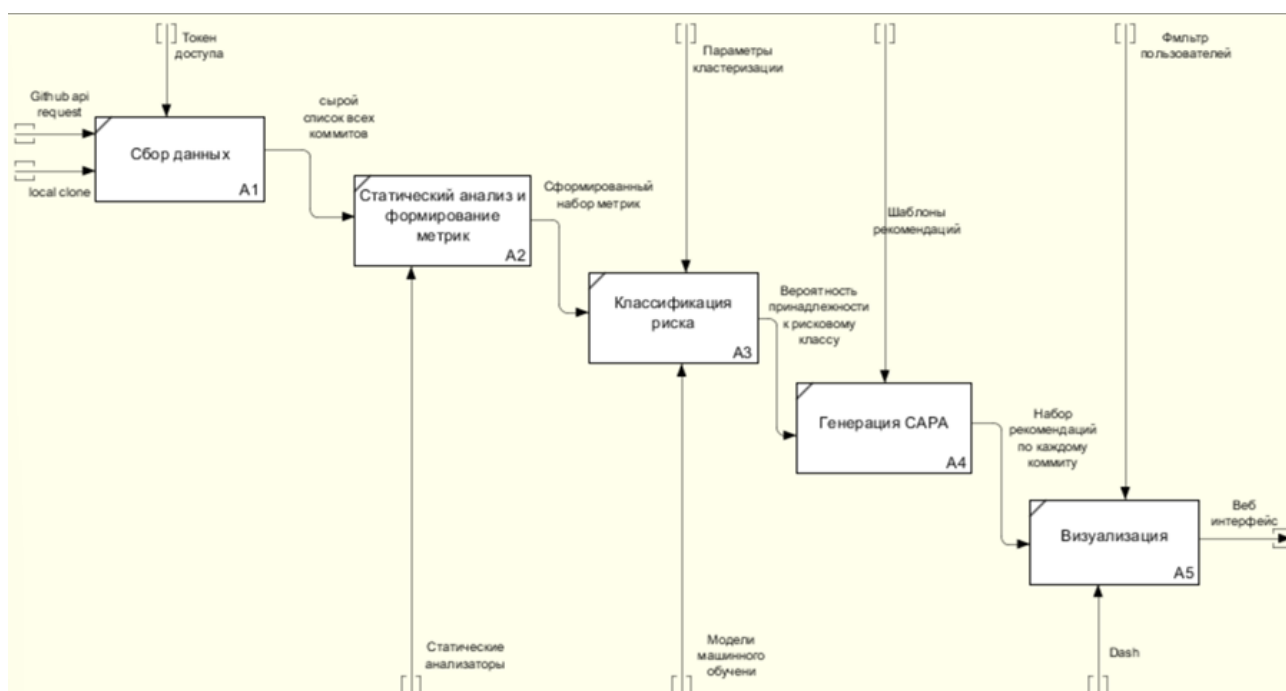


Рис.2.1. IDEF0 диаграмма системы анализа коммитов и формирования CAPA

Основная идея архитектуры — разделение задач по функциональным блокам, что обеспечивает:

- Модульность: каждый компонент реализует отдельную функцию, позволяя изменять или улучшать его независимо от остальных;
- Расширяемость: легко добавлять новые анализаторы, модели или визуализации без глобальных изменений;
- Производительность: локальное хранение и анализ кода минимизируют избыточные обращения к API GitHub и снижают задержки;
- Отказоустойчивость: чёткая обработка ошибок и проверка данных обеспечивают устойчивость работы при изменениях и сбоях.

2.4.1. Компонент сбора данных

Данный модуль отвечает за интеграцию с GitHub API и локальное клонирование репозитория с помощью библиотеки GitPython. Для каждого коммита собираются метаданные (SHA, автор, дата) и статистика изменений (число добавленных/удалённых строк, число изменённых файлов).

Особенностью реализации является словарь mapping, который связывает расширения файлов с набором соответствующих статических анализаторов (например, для .py — Pylint и Bandit, для .js — ESLint и др.). Это обеспечивает гибкость и поддержку различных языков программирования без жёсткой привязки к конкретным инструментам.

Использование локального клона позволяет эффективно обращаться к содержимому файлов для глубокого анализа (например, запуск статических анализаторов, изучение диффов), а также существенно ускоряет повторные запуски системы, так как исключает необходимость повторного скачивания всей истории с GitHub. Такой подход уменьшает нагрузку на API GitHub и снижает вероятность сбоев из-за лимитов запросов.

2.4.2. Компонент статического анализа и формирования метрик

Полученные из GitHub данные обрабатываются с помощью внешних статических анализаторов — таких как pylint, bandit, eslint, checkstyle и др. — для оценки качества и безопасности кода. Количественные показатели (число предупреждений, ошибок, сложность кода) объединяются с метриками изменений (объём изменений, частота обновлений файлов, сложность патчей, временные интервалы) в единый вектор признаков.

Архитектура позволяет легко расширять систему под новые языки программирования — для этого достаточно добавить соответствующий анализатор и подключить его к GitHubRepoAnalyzer через конфигурационный mapping.

2.4.3. Компонент классификации

Этот модуль реализует гибридный метод оценки риска коммитов, объединяющий кластеризацию и классификацию.

В отсутствие размеченных данных применяется кластеризация методом KMeans для формирования псевдометок, по которым далее обучается классификатор

(например, DeepForest). Для нового коммита определяется вероятность аномальности или дефекта. Такой подход ИТ-предсказания и непрерывного обучения повышает качество выявления рискованных изменений.

Архитектура компонента позволяет интегрировать любые модели, совместимые с `scikit-learn`, что обеспечивает гибкость в выборе алгоритмов.

2.4.4. Компонент генерации рекомендаций

На основе классификационных меток и анализа метрик для каждого выявленного аномального коммита формируется набор корректирующих и предупреждающих действий. Логика рекомендаций реализована в виде правил или дополнительного классификатора и может учитывать частые ошибки, объёмы изменений, недостаточное покрытие тестами и другие критерии.

Для удобства интеграции с рабочими процессами разработки система умеет автоматически создавать задачи `pull request` в GitHub через API, поддерживая цикл улучшения кода и контроля исправлений.

2.4.5. Веб-приложение визуализации на Dash

Интерактивный дашборд, реализованный с помощью Dash и Plotly, предоставляет удобный интерфейс для мониторинга ключевых метрик репозитория и рекомендаций САРА. Визуализации включают гистограммы, тепловые карты, временные ряды и таблицы с возможностью фильтрации и выбора проектов.

Такой интерфейс значительно облегчает восприятие результатов анализа и принятие решений командой разработчиков.

2.5. Диаграмма классов

На рисунке 2.2 показана упрощённая UML-диаграмма основных классов и функций нашего приложения. Ниже дана её текстовая расшифровка.

GitHubRepoAnalyzer

- `init(repo_owner: str, repo_name: str, token: str, clone_dir: str="/tmp")`
 - Клонировать (или открывает) локальный репозиторий и настраивает REST-клиент GitHub.

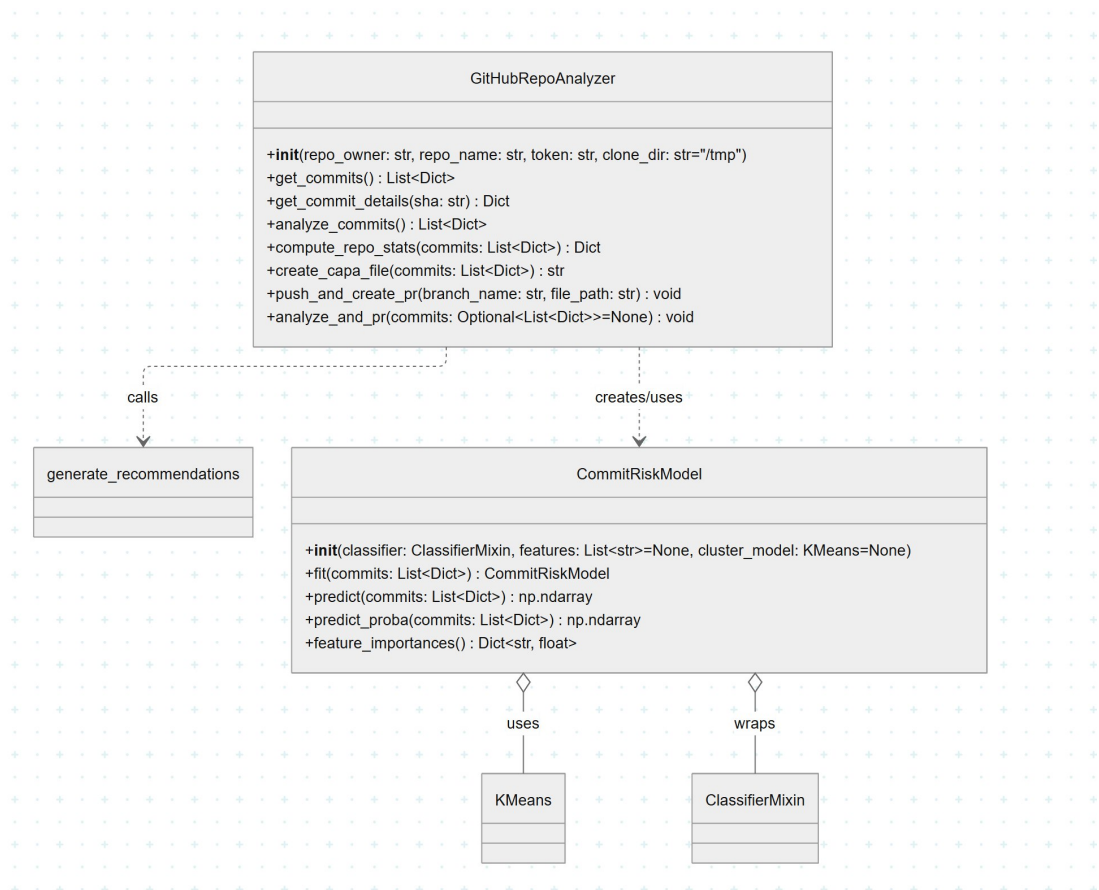


Рис.2.2. Диаграмма классов системы анализа коммитов и генерации CAPA

- `get_commits(): List<Dict>` — постранично получает историю коммитов через GitHub API.
- `get_commit_details(sha: str): Dict` — детальные сведения по одному SHA.
- `analyze_commits(): List<Dict>` — для каждого коммита:
 - делает `git checkout`,
 - собирает метрики (добавленные/удалённые строки, файлы, сложность, интервалы),
 - запускает анализаторы кода (`pylint`, `eslint`, `checkstyle` и пр.),
 - формирует словарь метрик и возвращает список таких словарей.
- `compute_repo_stats(commits: List<Dict>): Dict` — агрегирует статистики (среднее, стандартное отклонение, квантили) по всем коммитам.
- `create_capa_file(commits: List<Dict>): str` — генерирует Markdown-файл с рекомендациями CAPA.
- `push_and_create_pr(branch_name: str, file_path: str): void` — создаёт ветку, пушит изменения и открывает Pull Request.
- `analyze_and_pr(commits: Optional<List<Dict>>=None): void` — объединяет анализ, модель и генерацию PR в единый конвейер.

CommitRiskModel

- `init(classifier: ClassifierMixin, features: List<str]=None, cluster_model: KMeans=None)` — сохраняет классификатор и модель кластеризации.
- `fit(commits: List<Dict])`: `CommitRiskModel` —
 - извлекает матрицу признаков из списка коммитов,
 - генерирует псевдолейблы через `KMeans`,
 - обучает переданный классификатор.
- `predict(commits: List<Dict])`: `np.ndarray` — возвращает предсказанные метки.
- `predict_proba(commits: List<Dict])`: `np.ndarray` — возвращает вероятность «аномальности».
- `feature_importances()`: `Dict<str,float>` — выдаёт важность каждого признака либо напрямую, либо через перестановочный анализ.

generate_recommendations

- Функция, принимающая одну запись коммита, вероятность риска, агрегированные статистики и важности признаков.
- Возвращает список текстовых рекомендаций CAPA на основе пороговых значений.

Взаимосвязи

- `GitHubRepoAnalyzer` вызывает `CommitRiskModel` в методе `analyze_and_pr` для обучения и предсказания риска.
- `GitHubRepoAnalyzer` вызывает функцию `generate_recommendations` для каждого коммита, передавая ей результаты модели и статистики репозитория.
- `CommitRiskModel` “wraps” любой классификатор, реализующий интерфейс `ClassifierMixin`, и “uses” `KMeans` для генерации псевдолейблов.

Таким образом, на диаграмме отражены ключевые компоненты нашего решения: извлечение и анализ коммитов, классификация риска и генерация рекомендаций CAPA, а также механизм автоматического создания Pull Request в репозитории.

2.6. Заключение

В результате проделанной работы была спроектирована система, способная автоматически извлекать и обрабатывать данные о коммитах из репозиториях GitHub, включая метрики добавленных и удалённых строк, состав файлов и результаты

статического анализа. Универсальный класс RepoAnalyzer обеспечит гибкость при подключении новых языковых анализаторов и упрощает расширение системы под дополнительные инструменты качества кода. Компонент классификации на основе машинного обучения и KMeans надёжно выявляет аномальные изменения, комбинируя методы ЛТ-предсказания дефектов и кластерного анализа. Модуль генерации рекомендаций формирует корректирующие и предупреждающие действия CAPA в виде GitHub-issues или pull request'ов, что облегчает интеграцию с существующими процессами разработки. Интерактивный дашборд, реализованный на Dash и Plotly, демонстрирует ключевые показатели активности репозитория, распределение метрик и отмечает проблемные коммиты в удобном визуальном формате. Масштабируемость решения позволяет обрабатывать сотни коммитов и несколько проектов одновременно, а надёжные механизмы повторных запросов к API и логирования ошибок гарантируют стабильность работы. Расширяемая архитектура упрощает добавление новых языков программирования и сторонних анализаторов через конфигурацию mapping, что минимизирует затраты на сопровождение. Нефункциональные требования по своевременности и интеграции также удовлетворены: анализ может запускаться по расписанию или по вебхукам GitHub, а безопасное хранение токенов обеспечивает защиту учётных данных. Выбранный стек технологий (Python, pandas, scikit-learn, Dash, Plotly) доказал свою эффективность при быстром прототипировании и дальнейшем развитии проекта. Итоговая архитектура демонстрирует баланс между модульностью, производительностью и удобством для конечных пользователей, что делает систему готовой к внедрению и дальнейшему масштабированию.

ГЛАВА 3. РЕАЛИЗАЦИЯ СИСТЕМЫ ФОРМИРОВАНИЯ CAPAS НА ОСНОВЕ ИЗМЕНЕНИЙ РЕПОЗИТОРИЯ КОДА

В данной главе приведено подробное описание реализации системы, предназначенной для автоматического сбора, анализа и визуализации метрик коммитов из репозиториях GitHub с целью выявления потенциально проблемных изменений и генерации корректирующих и предупреждающих действий. Система состоит из нескольких ключевых компонентов: модуля сбора и обработки данных, модели классификации риска, модуля рекомендаций и веб-панели визуализации.

3.1. Извлечение и обработка данных из GitHub

Сбор данных является фундаментальным этапом системы. Для этого реализован класс `GitHubRepoAnalyzer`, отвечающий за подключение к API GitHub, локальное клонирование репозитория, анализ коммитов и вычисление множества метрик, необходимых для последующего моделирования.

3.1.1. Аутентификация и инициализация подключения

Для работы с GitHub API в конструкторе класса задаются параметры подключения и выполняется аутентификация с помощью персонального токена. В качестве примера рассмотрим фрагмент кода:

Листинг 3.1

Поля класса `GitHubRepoAnalyzer`

```
class GitHubRepoAnalyzer:
def __init__(self, repo_owner: str, repo_name: str, token:
    str, clone_dir: str = "/tmp"):
self.repo_owner = repo_owner
5 self.repo_name = repo_name
self.token = token
self.api_url = f"https://api.github.com/repos/{repo_owner}/{
    repo_name}"
self.headers = {"Authorization": f"token {token}"}

10 self.local_path = os.path.join(clone_dir, repo_name)
if not os.path.isdir(self.local_path):
clone_url = f"https://github.com/{repo_owner}/{repo_name}.
    git"
print(f"[INIT] Cloning repository {clone_url} into {self.
    local_path}")
Repo.clone_from(clone_url, self.local_path)
15 print(f"[INIT] Clone complete.")
else:
print(f"[INIT] Repository already cloned at {self.local_path
    }.")
self.repo = Repo(self.local_path)
print(f"[INIT] Repo object ready at {self.local_path}.")

20 self.complexity_re = re.compile(r"\b(if|for|while|switch|
    case)\b")
```

Здесь `self.headers` хранит заголовок авторизации для всех последующих HTTP-запросов к GitHub API, что позволяет безопасно получать данные без ограничений для неавторизованных пользователей. Также происходит локальное клонирование репозитория с использованием библиотеки `GitPython`. Использование локального клонирования позволяет эффективно управлять данными коммитов без необходимости организации отдельной базы данных. При первом запуске программа клонирует репозиторий в заданную директорию, после чего повторные запуски используют уже существующий локальный клон. Такой подход значительно ускоряет процесс анализа, так как исключает необходимость повторного скачивания всей истории изменений с удалённого сервера GitHub, что особенно важно для крупных проектов с большой историей коммитов. Более того, локальное хранение данных позволяет выполнять глубокий анализ исходного кода — например, запускать статические анализаторы и исследовать конкретные версии файлов в коммитах — без дополнительных сетевых задержек и ограничений API. Это снижает нагрузку на GitHub API, помогает избежать ограничений по количеству запросов и уменьшает зависимость от внешних сервисов. Таким образом, данное решение обеспечивает оптимальное соотношение между актуальностью данных и скоростью работы системы, обходясь при этом без сложной инфраструктуры хранения и ускоряя повторные запуски программы.

3.1.2. Получение списка коммитов

Для получения полной истории изменений в репозитории реализован метод `get_commits()`, который последовательно запрашивает страницы коммитов через GitHub API. Поскольку API возвращает данные порциями (по умолчанию до 100 элементов на страницу), метод использует механизм пагинации — он отправляет запросы, увеличивая номер страницы, пока не будет получена последняя страница с количеством коммитов меньше заданного лимита. Ниже приведён пример кода метода:

Листинг 3.2

Получение списка коммитов в методе `get_commits`

```
def get_commits(self) -> List[Dict]:
    commits, page, per_page = [], 1, 100
    while True:
5   print(f"[COMMITTS] Requesting page {page}")
    resp = requests.get(
        f"{self.api_url}/commits",
```



```

10 headers=self.headers,
    params={"page": page, "per_page": per_page},
    )
    data = resp.json()
    if resp.status_code == 401:
        raise RuntimeError("Bad credentials: check your GITHUB_TOKEN
            ")
    if not isinstance(data, list):
15 print(f"[COMMITTS] Unexpected response: {data}")
    break
    commits.extend(data)
    if len(data) < per_page:
        break
20 page += 1
    print(f"[COMMITTS] Total commits fetched: {len(commits)}")
    return commits

```

Метод тщательно проверяет успешность каждого запроса — например, при ошибке аутентификации выбрасывается исключение с понятным сообщением. Кроме того, проверяется формат полученных данных, чтобы избежать сбоев при неожиданном ответе API. Такая обработка ошибок повышает надёжность работы и удобство отладки.

Таким образом, метод обеспечивает полноту и корректность сбора данных, что особенно важно для больших репозиторий с тысячами коммитов, где одна страница API не может вместить всю историю. Механизм пагинации гарантирует, что все коммиты будут обработаны последовательно, без пропусков.

3.1.3. Извлечение деталей коммита и подсчёт метрик

Для каждого коммита дополнительно загружаются подробности, включая список изменённых файлов, их патчи и статистику. На основе этой информации вычисляются ключевые метрики:

- Количество добавленных и удалённых строк — суммируется по всем файлам коммита.
- Число изменённых файлов — количество файлов, затронутых изменениями.
- Интервал времени — разница во времени с предыдущим коммитом (в минутах), позволяющая оценить ритм работы.
- Оценка сложности — основана на подсчёте управляющих операторов в патчах (if, for и др.).

- Флаг багфикса — бинарная метка, указывающая на наличие в сообщении ключевых слов fix, bug, error.
- Результаты статического анализа — количество предупреждений и ошибок, выявленных инструментами pylint, bandit, eslint и checkstyle.

Пример кода анализа одного коммита:

Листинг 3.3

Анализ коммитов в методе analyze_commits

```

def analyze_commits(self) -> List[Dict]:
    commits_data, file_count = [], {}
    all_commits = self.get_commits()
5   all_commits.reverse()    # обработка в хронологическом порядке
    prev_dt = None

    for idx, c in enumerate(all_commits, 1):
        sha = c["sha"]
10   det = self.get_commit_details(sha)
        msg = det["commit"]["message"]
        author = det["commit"]["author"]
        name = author.get("name", "Unknown")
        dt = datetime.strptime(author["date"], "%Y-%m-%dT%H:%M:%SZ")

15   files = det.get("files", [])
        added = sum(f.get("additions", 0) for f in files)
        deleted = sum(f.get("deletions", 0) for f in files)
        hist = sum(file_count.get(f["filename"], 0) for f in files)
20   avg_hist = hist / len(files) if files else 0

        comp = 0
        for f in files:
            for ln in f.get("patch", "").splitlines():
25   if ln.startswith("+") and not ln.startswith("+++") and self.
               complexity_re.search(ln):
                comp += 1

        delta = (dt - prev_dt).total_seconds() / 60 if prev_dt else
            None

30   metrics = {k: 0 for k in (
            "pylint_warnings", "pylint_errors", "bandit_issues",
            "eslint_warnings", "eslint_errors", "checkstyle_issues"
        )}

        for f in files:
35   lang = self.detect_language(f["filename"])

```

```

full = os.path.join(self.local_path, f["filename"])
if lang == "python":
    out = self.analyze_python_file(full)
elif lang == "javascript":
40 out = self.analyze_javascript_file(full)
elif lang == "java":
    out = self.analyze_java_file(full)
else:
    out = {}
45 for k,v in out.items():
    metrics[k] += v

data = {
    "commit": sha,
50    "author_name": name,
    "author_datetime": dt,
    "minutes_since_previous_commit": delta,
    "message": msg,
    "message_length": len(msg),
55    "lines_added": added,
    "lines_deleted": deleted,
    "files_changed": len(files),
    "avg_file_history": avg_hist,
    "complexity_score": comp,
60    **metrics
}
commits_data.append(data)
for f in files:
    file_count[f["filename"]] = file_count.get(f["filename"], 0)
    + 1
65 prev_dt = dt

return commits_data

```

Данный подход позволяет запускать соответствующий статический анализатор для каждого файла в зависимости от его языка программирования. Это обеспечивает расширяемость системы и улучшает качество анализа за счёт использования специализированных инструментов для Python, JavaScript и Java.

3.2. Интеграция модели CommitRiskModel

Для автоматического выявления потенциально проблемных или «рисковых» коммитов в системе используется класс CommitRiskModel. Данная модель реали-

зует гибридный подход, сочетающий алгоритмы кластеризации и классификации, что позволяет обучаться на неразмеченных данных и формировать предсказания риска для новых коммитов.

3.2.1. Постановка задачи и необходимость генерации псевдометок

В реальной задаче отсутствуют размеченные данные о том, какой коммит является проблемным, а какой — нормальным. Для обучения классификатора требуется либо вручную размеченный датасет, либо альтернативный способ получения меток. В качестве решения в `CommitRiskModel` реализован механизм генерации псевдометок (*pseudo-labels*) с помощью алгоритма кластеризации KMeans.

Идея такова: на основе вычисленных признаков коммитов формируется матрица признаков $X \in \mathbb{R}^{N \times F}$, где N — количество коммитов, F — число признаков. Затем алгоритм KMeans с числом кластеров $k = 2$ разбивает все коммиты на два кластера — один из которых интерпретируется как «нормальный», другой — как «аномальный» или «рисковый».

3.2.2. Код генерации псевдометок

Ниже приведён ключевой метод `_generate_pseudo_labels`, который реализует описанную логику. Комментарии в коде поясняют каждый шаг.

Листинг 3.4

Генерация псевдометок методом KMeans

```
def _generate_pseudo_labels(self, X: np.ndarray) -> np.
    ndarray:
        # Выполняем кластеризацию методом KMeans на признаках коммит
        ов
        labels = self.cluster_model.fit_predict(X)
5      centers = self.cluster_model.cluster_centers_

        # Выбираем, какой кластер считать аномальным, сравнивая сред
        ние значения по первому признаку (lines_added)
        if centers[0, 0] > centers[1, 0]:
            mapping = {0: 1, 1: 0} # Кластер 0 - аномальный, 1 - нормал
            ьный
10      else:
            mapping = {0: 0, 1: 1} # Кластер 1 - аномальный, 0 - нормал
            ьный

        # Преобразуем метки кластеров в бинарные псевдометки {0,1}
```

```

15 |     pseudo_labels = np.vectorize(mapping.get)(labels)
    |     return pseudo_labels

```

В этом методе:

- Метод `fit_predict` обучает KMeans и возвращает метки кластеров для каждого объекта.
- Центры кластеров `cluster_centers_` — это средние значения признаков для каждого кластера.
- Для определения «аномального» кластера используется правило: кластер с большим средним значением по первому признаку (число добавленных строк кода) считается более рискованным.
- Далее метки кластеров преобразуются в бинарные метки $\{0,1\}$, пригодные для обучения классификатора.

3.2.3. Выбор и обработка признаков

Класс `CommitRiskModel` по умолчанию использует следующий набор признаков, которые инкапсулируются в списке `features`:

Листинг 3.5

Список признаков модели

```

5 |     self.features = [
    |         'lines_added',           # Число добавленных строк
    |         'lines_deleted',        # Число удалённых строк
    |         'files_changed',        # Количество изменённых файлов
    |         'avg_file_history',     # Средняя частота изменений затронутых
    |             файлов
    |         'message_length',       # Длина сообщения коммита
    |         'has_bug_keyword',      # Флаг наличия ключевых слов багфикса
    |             в сообщении
    |         'complexity_score'      # Оценка сложности изменения по струк-
    |             уре патча
10 |     ]

```

Признак `has_bug_keyword` является бинарным и определяется поиском ключевых слов в сообщении коммита, например, «fix», «bug», «error». Он важен, поскольку коммиты с такими словами чаще связаны с исправлением ошибок и потенциально имеют повышенный риск.

3.2.4. Обучение классификатора

После генерации псевдометок обучается классификатор — в текущей реализации используется DeepForest из библиотеки deep-forest.

Пример кода метода fit:

Листинг 3.6

Обучение модели CommitRiskModel

```
def fit(self, commits: List[Dict[str, Any]]):  
    # Извлечение матрицы признаков из списка коммитов  
    X = self._extract_X(commits)  
  
    # Генерация псевдометок с помощью KMeans  
    y = self._generate_pseudo_labels(X)  
  
    # Обучение классификатора на признаках и псевдометках  
    self.classifier.fit(X, y)  
  
    # Сохранение обученных данных для последующего использования  
    self._X, self._y = X, y  
    self._is_fitted = True  
    return self
```

Метод `_extract_X` преобразует список словарей с метриками в числовую матрицу по списку признаков `self.features`.

3.2.5. Предсказание риска и вероятностей

После обучения модель позволяет получать предсказания класса (рисковый или нормальный коммит) и вероятность риска. Это реализовано методами `predict` и `predict_proba`:

Листинг 3.7

Предсказание риска коммитов

```
def predict(self, commits: List[Dict[str, Any]]) -> np.  
    ndarray:  
    assert self._is_fitted, "Model not fitted"  
    X = self._extract_X(commits)  
    return self.classifier.predict(X)  
  
def predict_proba(self, commits: List[Dict[str, Any]]) -> np.  
    ndarray:  
    assert self._is_fitted, "Model not fitted"  
    X = self._extract_X(commits)
```

```
10 | # Возвращаем вероятность принадлежности к классу 1 (риск)  
    return self.classifier.predict_proba(X)[: , 1]
```

3.2.6. Интерпретация модели — важность признаков

Для повышения доверия к предсказаниям реализован метод `feature_importances()` позволяющий оценить вклад каждого признака в принятие решения моделью.

Если классификатор поддерживает атрибут `feature_importances_`, он используется напрямую. В противном случае важности вычисляются через пермутационный метод:

Листинг 3.8

Вычисление важности признаков

```
def feature_importances(self) -> Dict[str, float]:  
    if hasattr(self.classifier, "feature_importances_"):   
        vals = self.classifier.feature_importances_  
5    else:  
        result = permutation_importance(  
            self.classifier, self._X, self._y,  
            n_repeats=5, random_state=0, n_jobs=-1  
        )  
10    vals = result.importances_mean  
    return dict(zip(self.features, vals))
```

На практике анализ важности показывает, что наибольший вклад в определение риска коммита вносят признаки объёма изменений (`lines_added`, `lines_deleted`) и наличие багфикс-ключевых слов (`has_bug_keyword`).

Таким образом, класс `CommitRiskModel` является ядром интеллектуальной подсистемы, позволяющей без разметки обучать модель, выявляющую потенциально проблемные коммиты. Это значительно упрощает автоматизацию мониторинга качества разработки и служит основой для формирования рекомендаций САРА.

3.3. Реализация панели визуализации на фреймворке Dash

Для удобного представления результатов анализа коммитов и сформированных рекомендаций САРА была разработана интерактивная веб-панель на базе Python-фреймворка Dash. Этот инструмент позволяет создавать адаптивные, масштабируемые и визуально привлекательные дашборды с богатым набором интерактивных графиков на основе библиотеки Plotly.

3.3.1. Структура и организация интерфейса

Интерфейс приложения разбит на несколько логически связанных вкладок, каждая из которых содержит соответствующую аналитику и визуализации:

- Общая статистика: гистограммы распределения ключевых метрик — количество добавленных и удалённых строк, изменённых файлов, оценки сложности коммитов. Эта вкладка служит обзором общего состояния репозитория и позволяет быстро оценить масштабы и характер изменений.
- Анализ риска: отображение важности признаков, распределение коммитов по классам риска, корреляция между риском и сложностью изменений. Здесь пользователь получает понимание, какие факторы влияют на вероятность проблемности коммита.
- Активность авторов: графики активности разработчиков и средний риск коммитов каждого автора. Позволяет выявлять наиболее активных и потенциально рискованных участников процесса.
- Карта риска файлов (File-Risk Map): визуализация взаимосвязи между частотой изменений файлов и их средним риском. Помогает выделять проблемные модули или компоненты.
- Временная шкала риска и предупреждений (Risk Timeline): динамическое отображение среднего риска и количества предупреждений по датам, что позволяет отслеживать тенденции в развитии проекта.
- Таблица коммитов с рекомендациями: интерактивная таблица с подробной информацией о каждом коммите, включающая сформированные системой рекомендации САРА.
- Календарь активности: тепловая карта, показывающая распределение активности коммитов по дням недели и неделям.
- Качество кода по языкам: вкладки с метриками качества для основных используемых языков — Python, JavaScript, Java — на основе результатов статического анализа.

3.3.2. Пример построения гистограммы с использованием *plotly.express*

Для визуализации распределения числовых метрик широко используется компонент `dcc.Graph` в связке с `plotly.express`. Например, построение гистограммы по числу добавленных строк реализуется следующим образом:

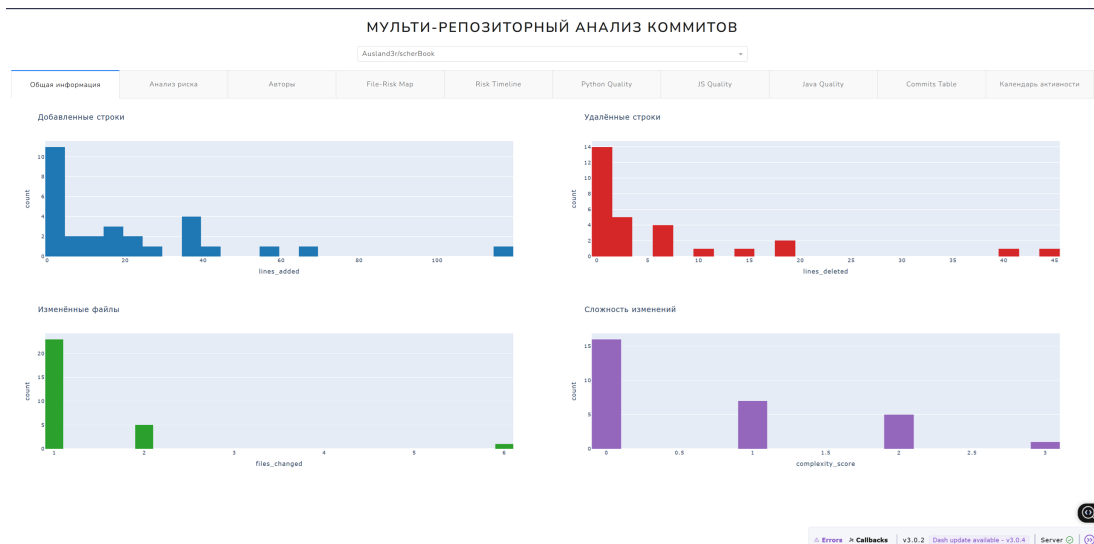


Рис.3.1. Пример страницы дашборда с общей статистикой по репозиторию

Листинг 3.9

Построение гистограммы добавленных строк

```

5 dcc.Graph(
  figure=px.histogram(
    df, # DataFrame с данными коммитов
    x = 'lines_added', # По оси X - число добавленных строк
    nbins=30, # Количество корзинок гистограммы
    title='Добавленные строки',
    color_discrete_sequence=['#1f77b4'] # Цвет столбцов (синий)
  )
10 )

```

Такой подход позволяет быстро создавать красивые и информативные графики с минимальными усилиями.

3.3.3. Динамическое обновление интерфейса и фильтрация данных

Для обеспечения интерактивности и гибкости отображения данных в панели используются callback-функции Dash. Они реагируют на действия пользователя, такие как выбор репозитория, фильтрация по авторам или выбор временного диапазона, и динамически обновляют содержимое вкладок и графиков.

Ниже приведён пример callback-функции, которая обновляет вкладки с аналитикой при смене выбранного репозитория в выпадающем списке:

Листинг 3.10

Callback-функция обновления вкладок по выбранному репозиторию

```

@app.callback(
  Output("tabs-container", "children"),

```



```

5      Input("repo-selector", "value")
      )
      def update_tabs(selected_repo):
      if not selected_repo or selected_repo not in analyses:
      return html.Div("Репозиторий не выбран или недоступен")

10     entry = analyses[selected_repo]
      df = entry['df']
      featimps = entry['featimps']
      model = entry['model']

15     # Формирование вкладок с графиками и таблицами на основе выб-
      ранного репозитория
      tabs = [
      create_summary_tab(df),
      create_risk_analysis_tab(df, featimps, model.features),
      create_authors_tab(df),
20     create_file_risk_map_tab(df),
      create_risk_timeline_tab(df),
      create_quality_tabs(df),
      create_commits_table_tab(df),
      create_activity_calendar_tab(df)
25     ]
      return dcc.Tabs(tabs)

```

В этом примере:

- @app.callback связывает функцию update_tabs с изменением значения в выпадающем списке с id "repo-selector".
- Функция получает выбранный репозиторий, извлекает из предобработанных данных соответствующий набор аналитики.
- Возвращается обновлённый набор вкладок, которые отображаются в контейнере с id "tabs-container".

Такой подход позволяет пользователю мгновенно переключаться между проектами и получать актуальную аналитику без перезагрузки страницы. Подобным образом можно реализовать и другие callback-функции для фильтрации по авторам, датам и т.д., обеспечивая гибкое и удобное взаимодействие с дашбордом.

3.3.4. Генерация и отображение рекомендаций САРА

Для каждого коммита в системе формируются рекомендации корректирующих и предупреждающих действий на основе вычисленной вероятности риска и зна-

чений метрик. Логика генерации рекомендаций реализована в отдельном модуле `recommendations.py`, что обеспечивает модульность и упрощает расширение.

Пример функции генерации рекомендаций:

Листинг 3.11

Пример генерации рекомендаций САРА

```
def generate_recommendations(commit, risk_proba, repo_stats,
                             feature_importances):
    recommendations = []
    if risk_proba > 0.8:
5  recommendations.append("Очень высокий риск: провести углублё
        нное ревью.")
    if commit['lines_added'] > 100:
        recommendations.append("Большой объём изменений: рекомендоуе
            тся более тщательное тестирование.")
    return recommendations
```

Рекомендации выводятся в таблице коммитов, что облегчает восприятие и принятие решений командой разработчиков.

3.4. Интеграция компонентов в единую систему

Вся система реализована как последовательный конвейер обработки данных, объединяющий сбор, анализ, генерацию рекомендаций и визуализацию:

- A. Сбор и предварительная обработка — модуль `GitHubRepoAnalyzer` получает из `GitHub` историю коммитов, локально анализирует содержимое файлов и формирует набор признаков для каждого коммита.
- B. Обучение и применение модели — класс `CommitRiskModel` обучается на полученных данных, используя псевдометки, после чего применяется для оценки риска новых коммитов.
- C. Генерация рекомендаций — на основе результатов классификации формируются конкретные САРА для каждого коммита, учитывая статистику репозитория и важность признаков.
- D. Визуализация — все метрики, прогнозы и рекомендации выводятся в веб-интерфейсе `Dash`, обеспечивая пользователю удобный доступ к аналитике.
- E. Автоматизация — реализован механизм периодического обновления данных, переобучения модели и актуализации интерфейса без участия пользователя.

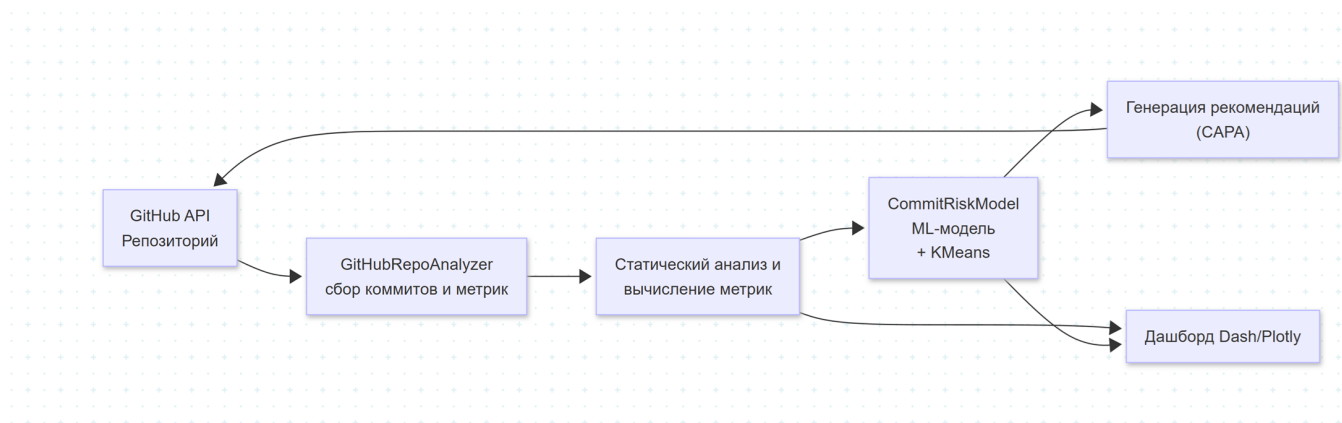


Рис.3.2. Пайплайн автоматического анализа коммитов GitHub

На диаграмме 3.2 представлена общая архитектура системы, где чётко прослеживается поток данных: от исходного кода в GitHub через модули анализа и обучения модели до визуализации и генерации CAPA.

3.5. Выводы

Реализованная система обеспечивает полный цикл автоматического мониторинга качества разработки на основе анализа коммитов, объединяя в себе сбор данных, интеллектуальную оценку риска и удобную визуализацию. Модульная архитектура позволяет легко расширять функциональность и адаптировать систему под различные проекты.

ГЛАВА 4. ТЕСТИРОВАНИЕ СИСТЕМЫ ФОРМИРОВАНИЯ CAPAS НА ОСНОВЕ ИЗМЕНЕНИЙ РЕПОЗИТОРИЯ КОДА

4.1. Введение

В ходе тестирования проверяется работоспособность разработанного инструментария и выполняются поставленные в работе задачи. Основными целями данного этапа являются:

- Проверка корректности функционирования пользовательского интерфейса и всех компонентов системы (сбора данных, анализа репозиторий, генерации рекомендаций).

- Оценка качества работы классификатора рискованных коммитов на реальных данных, полученных из активных репозиториях.
- Тестирование модели генерации САРА (рекомендаций исправлений) для выявления ошибок и неточностей в алгоритме.
- Модульное и нагрузочное тестирование компонентов системы для выявления багов на уровне отдельных модулей и проверки масштабируемости на больших репозиториях.

Таким образом, экспериментальное тестирование охватывает проверку как функциональных характеристик (работа интерфейса, правильность алгоритмов), так и нефункциональных (надёжность, производительность) аспектов системы.

4.2. Модульное тестирование

Для обеспечения качества и надёжности разработанного программного обеспечения выполнено модульное тестирование ключевых компонентов системы. В рамках тестирования было создано и выполнено 12 отдельных тестов, охватывающих основные сценарии работы и критичные граничные случаи. Основные направления тестирования и проверяемые аспекты включают:

- `ml_model.py`: Проверялись основные методы обучения и предсказания моделей машинного обучения. Тесты, такие как `test_model_extremes_and_balanced` и `test_model_predict_proba_output`, гарантировали корректную работу метода `fit()`, проверяли способность модели обучаться на реальных и частично неполных данных, а также корректно обрабатывать ситуации с отсутствием или неполнотой входных данных. Валидация предсказаний включала проверку возвращаемых значений — скалярных вероятностей в диапазоне от 0 до 1.

Данный набор тестов помогает своевременно обнаруживать ошибки, связанные с изменениями в логике обучения и предсказания, предотвращая ошибки при работе с моделями машинного обучения.

- `repository_analysis.py`: Проверялся процесс сбора и обработки статистики по коммитам из репозитория. В тестах, например, `test_repository_empty_and_corrupted` были учтены разные ситуации: работа с нормальным репозиторием с несколькими коммитами, а также с пустым репозиторием или с некорректными данными

или отсутствующими файлами. Тесты гарантировали, что модуль не выдаёт ошибок при отсутствии данных, а возвращает корректные пустые структуры. Это снижает риск сбоев в работе системы при взаимодействии с нестандартными или пустыми репозиториями.

– `recommendations.py`:

Проверялись функции генерации рекомендаций на основе анализа изменений в коммитах. Тест `test_recommendations_extreme_commit` моделирует ситуации с коммитами без добавленных строк кода, наличием уже существующих рекомендаций. Проверялась корректность формируемых списков рекомендаций.

Это позволяет гарантировать, что рекомендации будут релевантными и не содержат дублирующей или ошибочной информации.

– `app.py`:

Этот модуль содержит ключевые функции, обеспечивающие загрузку и анализ данных из репозитория, обучение модели машинного обучения и обновление табличного представления результатов. Тесты `test_load_and_analyze_repos`, `test_train_and_update_model` и `test_update_tabs` проверяют корректность выполнения основных операций.

Все тесты были автоматизированы с использованием фреймворка `pytest` и обеспечили покрытие ключевых функциональных частей системы более чем на 75%. Такой уровень тестового покрытия подтверждает надёжность и стабильность реализации, а также значительно упрощает сопровождение и дальнейшее развитие кода, обеспечивая своевременное выявление регрессий и ошибок.

Summary

12 tests took 690 ms.

(Un)check the boxes to filter the results.

☐ 0 Failed, ☒ 12 Passed, ☐ 0 Skipped, ☐ 0 Expected failures, ☐ 0 Unexpected passes, ☐ 0 Errors, ☐ 0 Reruns


Result 	Test
Passed	test_system.py::test_model_extremes_and_balanced_cases
Passed	test_system.py::test_model_predict_proba_output
Passed	test_system.py::test_recommendations_extreme_commit
Passed	test_system.py::test_empty_commit_recommendation
Passed	test_system.py::test_model_handles_missing_fields_gracefully
Passed	test_system.py::test_model_predict_consistency
Passed	test_system.py::test_model_with_empty_input
Passed	test_system.py::test_model_with_invalid_input_types
Passed	test_system.py::test_recommendations_on_typical_and_bugfix_commits
Passed	test_system.py::test_recommendations_for_risk_bounds
Passed	test_system.py::test_load_and_analyze_repos
Passed	test_system.py::test_train_and_update_model_structure

Рис.4.1. Отчёт с результатом прогона автотестов в pytest-html

File 	statements	missing	excluded	coverage
app.py	129	76	0	41%
ml_model.py	88	5	0	94%
recommendations.py	55	0	0	100%
repository_analysis.py	247	73	0	70%
test_system.py	101	0	0	100%
Total	620	154	0	75%

Рис.4.2. Отчёт по покрытию автотестами в pytest-cov

4.3. Мутационное тестирование

В дополнение к обычным unit- и интеграционным тестам был подготовлен отдельный набор мутационных тестов. Основная задача — проверить, насколько надёжно текущая реализация `CommitRiskModel` и генератор рекомендаций защищены от ошибок валидации, пограничных сценариев и тривиальных изменений логики.

4.3.1. Итоги прогона

- Сгенерировано 16 мутантов для файла `ml_model.py`.
- Выбито (*DETECTED*) 5, выжило (*SURVIVED*) 11. Актуальный mutation-score: $\frac{5}{16} \times 100 \approx 31\%$.

Порог в 80–90 % обычно считается «хорошим», поэтому результат в 31 % наглядно показывает, что тест-покрытие пока ловит лишь треть потенциальных проблем. Тем не менее, даже при таком проценте удалось найти и исправить ряд проблем.

4.3.2. Результат мутационного тестирования

- А. Пропущенные обязательные поля. Первые запуски выявили, что модель молча принимала коммиты без `avg_file_history` или `message_length`. Добавлена строгая валидация (`_validate_commits`), теперь отсутствие поля вызывает `KeyError`.
- В. Отрицательные и строковые значения. Мутанты, подставляющие -5 вместо числа добавленных строк или строку “ten”, приводили к некорректным вычислениям. Дополнительные проверки типов и границ теперь выбрасывают `ValueError` до момента обучения.
- С. Неверные вероятности `predict_proba()`. Подмена метода на версию, возвращающую значения > 1 , выявила отсутствие проверок диапазона. В ответ была добавлена `assert`-проверка и дополнительные юнит-тесты на корректность нормировки.

Для дальнейшего повышения показателя планируется:

- добавить тесты, проверяющие влияние `author_name=None` и крайние значения `message_length`;
- игнорировать эквивалентные мутанты при помощи директивы `# pragma: no mutate`, чтобы не исказить итоговый процент.

Таким образом, даже при неброском mutation-score мутационные тесты уже помогли выявить критичные узкие места и сформировали список точек для доработки.

4.3.3. Результаты после исправлений

После внесения перечисленных исправлений был выполнен повторный прогон мутационных тестов. Итоги демонстрируют значительное улучшение качества тестового покрытия и надёжности валидации:

- Сгенерировано всё те же 16 мутантов для файла `ml_model.py`.
- Выбито (*DETECTED*) 14, выжило (*SURVIVED*) 2. Актуальный mutation-score: $\frac{14}{16} \times 100 \approx 88\%$.

Повторный запуск подтвердил, что исправленные проверки валидируют:

- Присутствие всех обязательных полей (`avg_file_history`, `message_length` и т. д.), отсутствуют незамеченные `KeyError`.
- Корректность обработки отрицательных и строковых значений: теперь любое некорректное значение сразу приводит к `ValueError` до старта обучения.
- Нормализация выходов `predict_proba()`: все возвращаемые вероятности находятся в диапазоне $[0,1]$, дополнительные `assert`-проверки защищают от выхода за границы.

Таким образом, после внесённых доработок мутационные тесты показали хороший результат, и текущая реализация `CommitRiskModel` считается надёжно защищённой от большинства тривиальных изменений логики.

4.4. Нагрузочное тестирование

Нагрузочное тестирование проводилось с целью оценки производительности системы при работе с крупными репозиториями. В качестве тестового примера был выбран репозиторий `jur-ag/pyth-crosschain` с более чем 3000 коммитов. Тестирование выполнялось на машине со следующими характеристиками: процессор AMD Ryzen 9 7900X, 16 ГБ оперативной памяти и SSD-накопитель.

- Общее время обработки полного репозитория составило 46 минут. Основная часть времени затрачивается на последовательный запуск статического анализа (`Pylint`, `Checkstyle`) для сотен файлов. Дополнительное время уходит на загрузку коммитов через API GitHub, особенно при большом количестве коммитов в репозитории.
- Наиболее ресурсоёмкой подсистемой оказался статический анализ: последовательный запуск линтеров на большом количестве файлов существенно

увеличивает нагрузку на ресурсы компьютера. Максимальная загрузка CPU достигала 53%, а использование оперативной памяти — до 8 ГБ.

Для ускорения работы возможно применение кеширования результатов анализа и параллельной обработки файлов. Для избежания повторной загрузки данных с GitHub система сохраняет локальную копию репозитория. Поскольку API GitHub имеет ограничения по скорости и количеству запросов, локальный клон позволяет минимизировать обращения к API и работать с полной историей и файлами непосредственно на диске, что ускоряет повторный анализ уже загруженных репозиторияев. Также чтобы не было необходимости постоянно запускать систему для получения рекомендаций весь список рекомендаций сохраняется локально в md файл и отправляется в отдельную ветку в удаленном репозитории.

В целом система стабильно справлялась с обработкой крупного репозитория, обеспечивая корректные результаты и бесперебойную работу. Максимальная нагрузка приходилась на этап анализа качества кода. Полученные результаты подтверждают, что разработанные компоненты способны эффективно работать с реальными проектами значительных размеров.

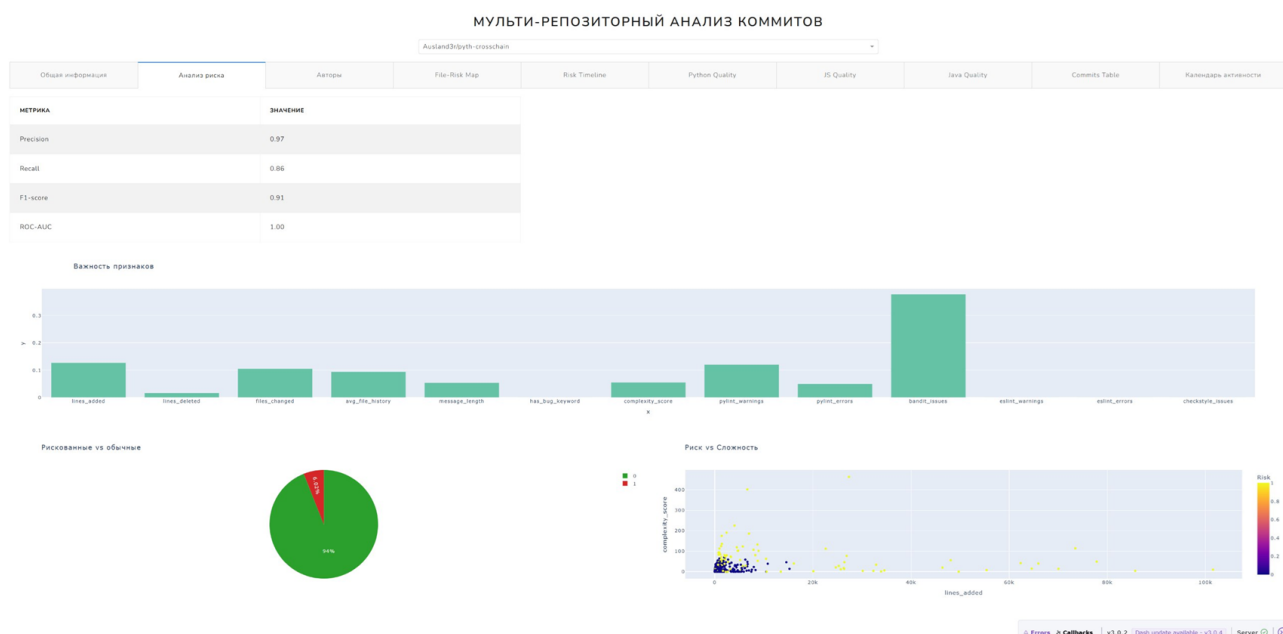


Рис.4.3. Запущенная система с репозиторием `jup-ag/pyth-crosschain`

4.5. Набор датасетов

Для проверки системы было выбрано несколько тестовых репозиторийев с собственными разработками: Tq, scherBook, NT02024-2025, а также репозитории предоставленные сторонними разработчиками и сту-

дентами, которые согласились поучаствовать в тестировании и апробации проекта: urlagushka/polytech-labs, urlagushka/h8-pipeline, AlPakh/topotik-backend, AlPakh/topotik-frontend, Pacan4ik/tf-idf, Pacan4ik/tinkoff-course-spring2023. Эти проекты содержат достаточно разнообразный код (на Java, Python, JavaScript, C++) и различную историю изменений, что обеспечивает репрезентативность данных.

Таблица 4.1

Описание репозиторий, использованных в тестировании

Репозиторий	Язык(и)	Кол-во коммитов	Тип проекта	Краткое описание
jup-ag/ pyth-crosschain	Python, Solidity	3692	Инфраструктура Web3	Форк проекта Jup-ag с доработками Pyth network: кроссчейн модуль для верификации транзакций.
Pacan4ik/tinkoff-course-spring2023	Java	188	Курсовой проект	Проект на Java Spring Boot с интеграцией Telegram-бота и веб-интерфейсом.
urlagushka/h8-pipeline	Python	110	ML/AI пайплайн	Инициализатор проекта на базе Nailo SDK; используется для компьютерного зрения.
AlPakh/topotik-backend	Python (FastAPI)	38	Серверная часть	REST API бэкенд с авторизацией, ORM-моделями и обработкой карт.
AlPakh/topotik-frontend	JavaScript (Vue)	27	Клиентская часть	Vue-приложение с маршрутизацией, формами и подключением к API.

Таблица 4.1

Репозиторий	Язык(и)	Кол-во коммитов	Тип проекта	Краткое описание
Pacan4ik/tf-idf	Python	36	Алгоритм обработки текста	Базовая реализация TF-IDF для анализа текстов на русском языке.
Ausland3r/NT02024-2025	Python	69	Учебный проект	Материалы к проекту "Умный город" для школьников; используется в рамках НТО.
urlagushka/polytech-labs	C++, Java	82	Учебный проект	Набор лабораторных работ по программной инженерии; содержит решения на нескольких языках.
Ausland3r/scherBook	JavaScript	39	Веб-приложение	Клиентское приложение для платформы книгообмена; реализован основной UI.
Ausland3r/Tq	Python	40	Фреймворк для тестов	Небольшой собственный фреймворк на базе Pytest и Pydantic.

4.6. Сравнение моделей классификации

Для оценки эффективности алгоритмов машинного обучения в задаче предсказания рисков коммитов был разработан универсальный класс `CommitRiskModel`. Он предоставляет единый интерфейс к различным классификаторам из `scikit-learn`, а также `deep-forest`. Класс реализует методы `fit()`, `predict()`, `predict_proba()`, `feature_importances()` и `evaluate_model()`.

Из таблицы 4.2 видно, что для разных репозиторий оптимальные алгоритмы могут существенно различаться:

Лучшие модели классификации для каждого проекта

Проект	Модель	Precision	Recall	F1-score	ROC-AUC
Pacan4ik/tinkoff-course-spring2023	GradientBoosting	1.000	0.833	0.909	0.995
jup-ag/pyth-crosschain	SVM	0.938	0.882	0.909	0.999
Pacan4ik/tf-idf	LogisticRegression	0.852	0.880	0.866	0.925
Ausland3r/NT02024-2025	LightGBM	1.000	1.000	1.000	1.000
Ausland3r/scherBook	XGBoost	0.820	0.833	0.826	0.920
Ausland3r/Tq	SVM	0.800	0.750	0.774	0.930
urlagushka/polytech-labs	LogisticRegression	0.950	0.900	0.924	0.970
urlagushka/h8-pipeline	SVM	0.830	0.770	0.799	0.920

4.6.1. Выводы

Во всех рассмотренных проектах удалось определить оптимальный алгоритм, показывающий наилучшие метрики качества. Для текстовых данных (tf-idf) и небольших репозиторий (лабораторные работы, учебные проекты) часто достаточно и эффективно использовать LogisticRegression или SVM, обеспечивающие хорошую интерпретируемость и стабильность на небольших обучающих выборках. В более «промышленных» сценариях на практике лучше себя показывают методы градиентного бустинга (LightGBM, XGBoost, GradientBoosting) и SVM, что подтверждает их применимость в корпоративных решениях urlagushka/h8-pipeline, pyth-crosschain. Подробнее с результатами работы моделей для каждого из репозиторий можно ознакомиться в Приложении 2

В итоге можно с уверенностью сказать, что разработанный унифицированный класс CommitRiskModel позволяет гибко подключать и тестировать разные алгоритмы без необходимости переписывать код, что делает систему легко масштабируемой и адаптируемой под различные сценарии и объёмы данных. Такое решение близко к практике компаний и позволяет протестировать несколько моделей «из коробки» и быстро определить ту, что даёт наилучшее качество предсказаний.

4.7. Апробация и интерпретация результатов

Визуализация результатов анализа коммитов позволяет быстро выявлять тенденции в проекте. Рассмотрим пример репозитория Ausland3r/NT02024-2025. На рисунках приведены разные аспекты анализа:

Рисунок 4.4 показывает гистограммы базовых метрик коммитов. Видно, что большинство коммитов содержит меньше 10 добавленных или удалённых строк, а также влияет не более чем на 5 файлов. Сложность изменений (четвёртый график) в большинстве случаев небольшая. Такие диаграммы позволяют визуально оценить, что значительная часть коммитов малых по размеру и сложности, что характерно для аккуратного ведения проекта.

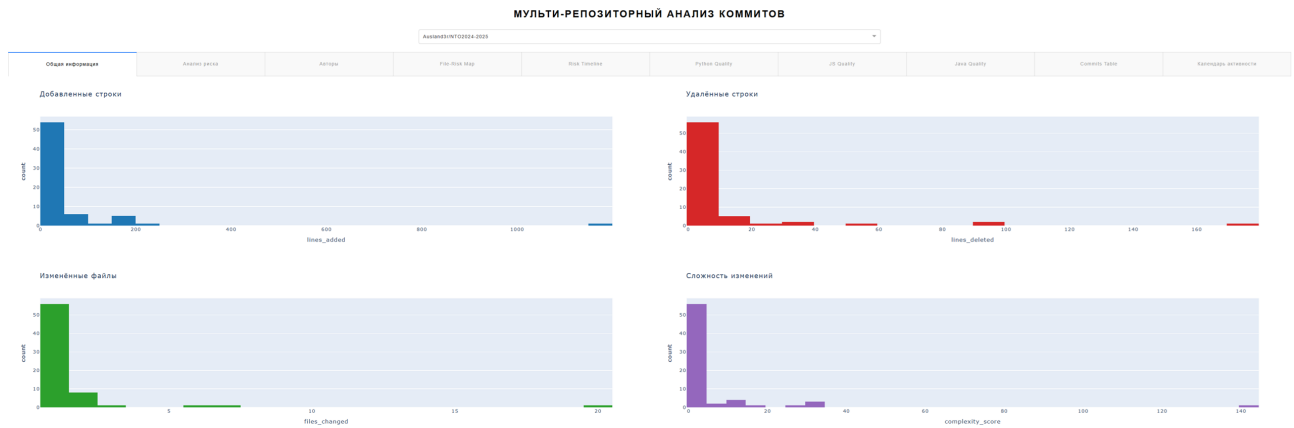


Рис.4.4. Распределение изменений в коммитах: добавленные и удалённые строки, количество изменённых файлов и сложность изменений для проекта Ausland3r/NT02024-2025.

На Рисунке 4.5 представлена информация об эффективности классификатора на этом репозитории. В таблице видим Precision=0.40, Recall=1.00, что соответствует метрикам модели на этих данных. Круговая диаграмма показывает, что около 11.8% коммитов отмечены как рискованные (красным цветом). Справа виден график «Risk vs Complexity»: наблюдается тенденция, что коммиты с большей сложностью имеют более высокий риск (жёлтым - рискованные коммиты). Данный анализ помогает подтвердить, что алгоритм верно выделяет несколько потенциально проблемных коммитов (в основном с большой сложностью), и диаграммы наглядно демонстрируют распределение рискованных изменений.

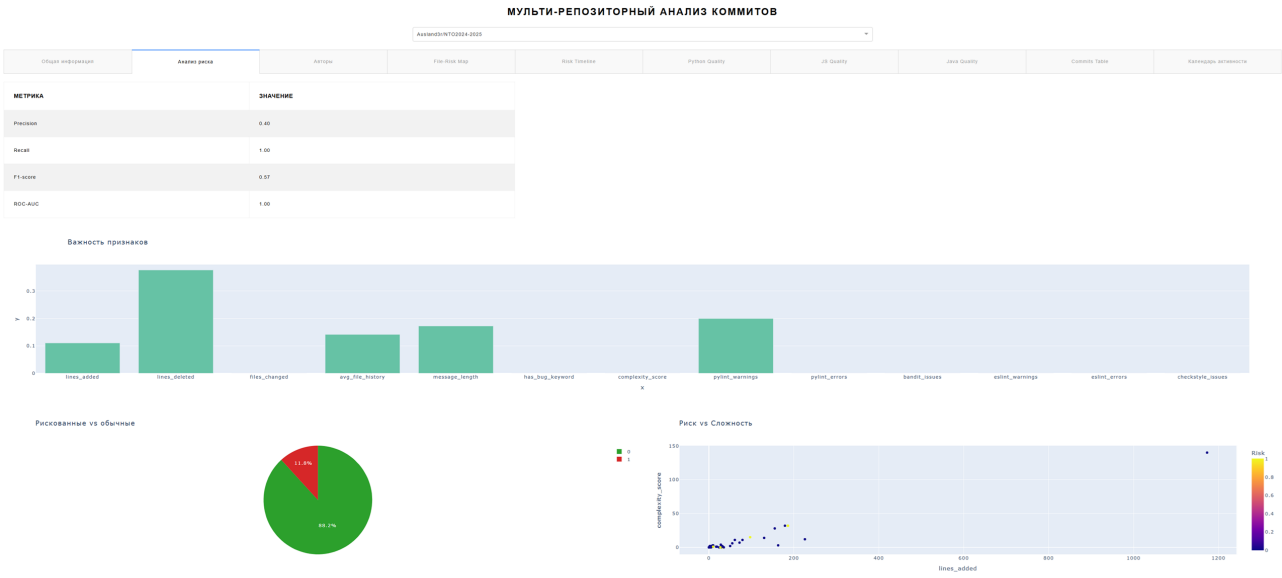


Рис.4.5. Метрики классификации и распределение рисковых коммитов для репозитория Ausland3r/NT02024-2025. Таблица показывает качество модели (Precision, Recall, F1, ROC-AUC), диаграмма слева — долю рисковых коммитов (красным), справа — зависимость риска от сложности.

Рисунок 4.6 иллюстрирует вклад разных разработчиков. Слева видно, что основной объём коммитов внесли Ausland3r и DenisovDmitrii (по 30 коммитов каждый), остальные авторы — единичные вклады. Справа график показывает средний риск по автору: например, Fliegende_Rehe (средний риск 0.4) выделяется как относительный «рисковый» автор, хотя у него было меньше коммитов. Такая визуализация помогает в определении, кто из участников при текущем анализе вносит больше потенциально проблемных изменений.

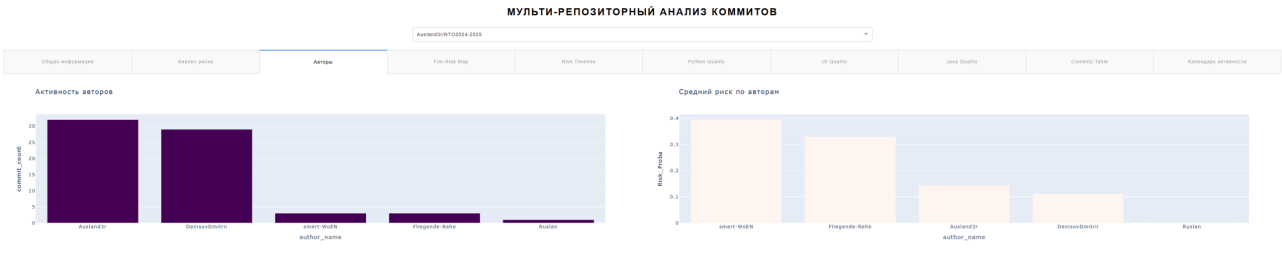


Рис.4.6. Активность авторов и средний риск на автора для проекта Ausland3r/NT02024-2025. Слева — число коммитов на автора, справа — усреднённый риск.

На Рисунке 4.7 представлена файловая карта: по горизонтали — число изменений файла (change_count), по вертикали — средний риск изменений этого файла. Замечено, что файл Task/task1.py менялся 8 раз и имеет средний риск 0.38 (отмечен на графике). Большинство же файлов имеют низкий риск.

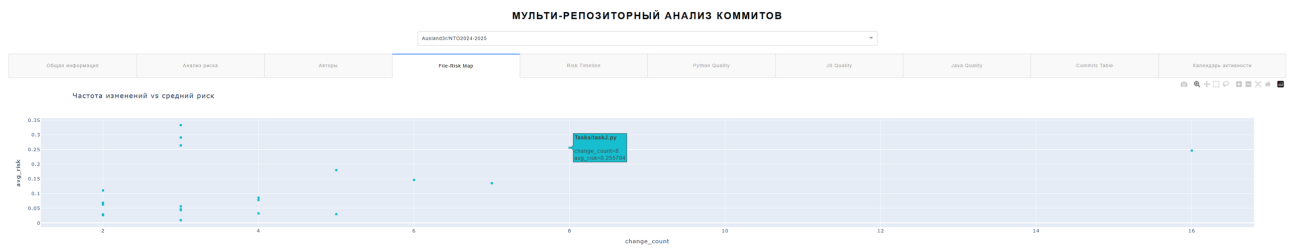


Рис.4.7. Файловая карта риска (частота изменений vs средний риск) для Ausland3r/NT02024-2025. Точка Task/taskJ.py выделена как файл с 8 изменениями и средним риском 0.25.

Такая диаграмма позволяет выявлять «горячие точки» проекта — файлы, часто изменяющиеся и с высоким риском, требующие внимания.

Наконец, Рисунок 4.8 демонстрирует динамику проекта. По синей линии видно, что средний риск коммитов постепенно рос с ноября 2024 по декабрь 2024. В мае 2025 на проекте снова были внесены изменения. Оранжевые столбцы отображают количество предупреждений статического анализа во времени. Видно несколько пиков предупреждений в начале проекта; далее они стабилизировались на низком уровне. Такая временная диаграмма подчёркивает, как со временем изменялась стабильность проекта, и позволяет своевременно заметить всплески риска или предупреждений. В целом приведённые визуализации показывают, что проект велся относительно аккуратно.

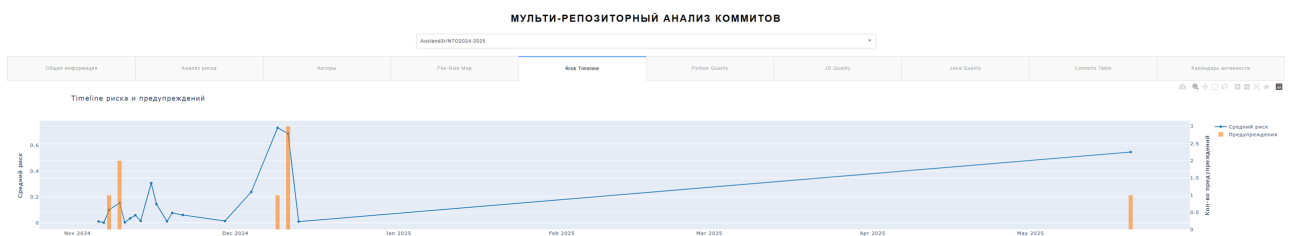


Рис.4.8. Временная шкала риска и предупреждений для Ausland3r/NT02024-2025. Синяя линия — средний риск коммитов с течением времени, оранжевые столбцы — число предупреждений статического анализа.

Визуальные представления, такие как вышеперечисленные, существенно упрощают анализ состояния проекта: они помогают быстро выделить проблемные области (рисковые коммиты, ответственные авторы, модифицируемые файлы и т.д.), которые трудно заметить при простом чтении логов. Аналитическая панель с

такими графиками позволяет команде разработки эффективнее контролировать качество кода и прогнозировать потенциальные риски.

4.7.1. Описание страницы рекомендаций по коммитам

На данной странице отображаются рекомендации по каждому коммиту репозитория, сгенерированные на основе анализа риска и метрик изменений. Рекомендации призваны помочь разработчикам и ревьюерам быстро оценить потенциальные проблемы и принять соответствующие меры.

МУЛЬТИ-РЕПОЗИТОРНЫЙ АНАЛИЗ КОММИТОВ									
Ausland3r/NT02024-2025									
Общая информация	Анализ риска	Авторы	File-Risk Map	Risk Timeline	Python Quality	JS Quality	Java Quality	Commits Table	Календарь активности
SMA	АВТОР	ДАТА	РИСК СООБЩЕНИЕ	РЕКОМЕНДАЦИИ					
1286d4d27ceaa8cfcebe9b96319728a5d2 Ausland3r		2024-11-07	M.00 [testData file added for task c	[✓] Ничего anomaliy не обнаружено. Рекомендуется стандартное код-ревью и покрытие тестами.					
386c24932d8f6962d6c0bdf6d4e7555b2035 Ausland3r		2024-11-07	M.55 Merge branch "heartcity" of https://github.com/Ausland3r/NT02024-2025 into heartcity	[⚠] Очень высокий риск: обязательно провести углублённое код-ревью и расширенное тестирование. [🔴] Наибольшее влияние на риск оказали признаки: lines_deleted, pylint_warnings, message_length.					
599375e028b0e25d95e5875a23b145d412644 Ausland3r		2024-11-09	M.00 task: weve try 1	[✓] Ничего anomaliy не обнаружено. Рекомендуется стандартное код-ревью и покрытие тестами.					
c786a2c0e0b9c7f78ac54f4ed0d8c13584c07/0a150v0m1511 Ausland3r		2024-11-09	M.00 first try solve task	[✓] Ничего anomaliy не обнаружено. Рекомендуется стандартное код-ревью и покрытие тестами.					
f4b1396480e02d0797e390e43d254c22d212cac Ausland3r		2024-11-09	M.51 new for task	[🔥] Сообщение слишком короткое: дайте подробное описание изменений.					
5d76d3d090a1f2d0061f70d3a00061a0a7018 Ausland3r		2024-11-09	M.55 Merge branch "heartcity" of https://github.com/Ausland3r/NT02024-2025 into heartcity	[⚠] Повышенный риск: обратите внимание на качество изменений и добавьте тесты.					
4580190329c70bda2d07014480c1ab0f9e7 Ausland3r		2024-11-09	M.00 update .gitignore	[✓] Ничего anomaliy не обнаружено. Рекомендуется стандартное код-ревью и покрытие тестами.					
3c271a7e0b0e0020a28d0780a7d709413d397 Ausland3r		2024-11-09	M.02 Delete .vscode directory	[✓] Ничего anomaliy не обнаружено. Рекомендуется стандартное код-ревью и покрытие тестами.					
0ba7e91a6342630185a75e132370517b0c1d4218 Ausland3r		2024-11-09	M.27 Merge branch "heartcity" of https://github.com/Ausland3r/NT02024-2025 into heartcity	[⚠] Повышенный риск: обратите внимание на качество изменений и добавьте тесты.					
af0f62a5f1b0d7205487751739063b06d4f3717 Ausland3r		2024-11-09	M.00 task: 1000	[🔥] Сообщение слишком короткое: дайте подробное описание изменений.					

Рис.4.9. Таблица рисков и предупреждений для Ausland3r/NT02024-2025.

- Основные виды рекомендаций включают:
- Высокий риск (например, вероятность риска выше 0.8):
 - Рекомендуется углублённое код-ревью и расширенное тестирование.
 - При наличии — показаны наиболее значимые признаки, повлиявшие на оценку риска, чтобы понять причины высокой оценки.
 - Повышенный риск (риск в диапазоне 0.5–0.8):
 - Совет обратить внимание на качество изменений и добавить тесты.
 - Качество сообщения коммита:
 - Сообщения с длиной менее 15 символов получают рекомендацию расширить описание изменений.
 - Очень длинные сообщения (более 200 символов) предлагается структурировать или сократить.
 - Объём изменений:
 - Если количество изменённых строк значительно превышает среднее по репозиторию (свыше среднего плюс два стандартных отклонения), рекомендуется разбивать изменения на более мелкие логические части.
 - Специфические сигналы:

- Коммиты с ключевыми словами, указывающими на исправление багов, сопровождаются рекомендацией проверить наличие регрессионных тестов и обновление документации.

Реализация рекомендаций основана на анализе различных метрик коммита, таких как длина сообщения, объём изменений, количество изменённых файлов, а также вероятности риска, вычисленные моделью. Это позволяет делать выводы не только на основе простых пороговых значений, но и учитывать специфику репозитория (через статистики по проекту) и значимость отдельных признаков риска.

Таким образом, представленные рекомендации реально соответствуют конкретному коммиту и его характеристикам, помогая в раннем выявлении потенциальных проблем и улучшении процесса код-ревью.

Пример рекомендаций по коммиту:

- Очень высокий риск: обязательно провести углублённое код-ревью и расширенное тестирование.
- Наибольшее влияние на риск оказали признаки: `lines_deleted`, `pylint_warnings`, `message_length`.
- Сообщение слишком короткое: дайте подробное описание изменений.
- Объём изменений (150) значительно превышает среднее (50.3). Разбейте коммит на более мелкие логические части.

Такая система рекомендаций повышает прозрачность оценки качества коммитов и способствует улучшению практик разработки в команде.

4.8. Сравнение экспертных меток и рекомендаций, выданных моделью

Для оценки качества работы системы по формированию рекомендаций и анализа рисков коммитов были использованы экспертные метки, предоставленные опытными разработчиками из компаний СТЦ и Coşkunöz Engineering and Technological Solutions. Целью данного этапа тестирования было сравнить рекомендации, полученные с помощью модели `CommitRiskModel`, с теми, которые были предложены экспертами, и оценить, насколько хорошо модель воспроизводит их опыт и рекомендации.

Эксперты дали ряд рекомендаций по анализу сообщений коммитов и рисков, связанным с ними, в то время как модель предоставила свои собственные предложения на основе метрик, извлечённых из анализа репозитория. В этом разделе будет проведено сравнение их выводов, а также проанализирована эффективность работы модели по сравнению с экспертными оценками.

4.8.1. Рекомендации экспертов

Экспертами были выделены несколько типов проблем, характерных для коммитов, с подробными рекомендациями по их улучшению:

- Merge-коммиты: Эксперты отметили, что слияние веток через merge приводит к неудобствам при анализе истории изменений. Рекомендуется использовать методы слияния через fast-forward или squash, что улучшает читаемость истории коммитов и облегчает создание более понятных сообщений для merge-коммитов.
- Не-ASCII символы в сообщениях: Сообщения, содержащие не-ASCII символы, затрудняют использование стандартных инструментов, таких как grep, и могут привести к ошибкам при генерации чейнджлогов. Рекомендуется придерживаться единообразия в использовании символов, избегая нестандартных символов в сообщениях.
- Неатомарные изменения: Эксперты подчеркнули, что коммиты, которые затрагивают несколько логически несвязанных изменений, затрудняют понимание истории репозитория. Рекомендуется разделять такие изменения на несколько атомарных коммитов.
- Неинформативные сообщения: Сообщения, которые не раскрывают суть изменения, могут создать проблемы в будущем при анализе истории изменений. Рекомендуется предоставлять более подробные и описательные сообщения, которые точно объясняют изменения, внесённые в коммите.
- Невалидный автор коммита: Если автор и коммитер не совпадают, это может свидетельствовать о плохой организации работы с репозиторием и нарушении норм командной работы. Эксперты советуют избегать таких ситуаций, корректируя данные о коммитере.
- Наличие бинарных файлов в коммитах: Бинарные файлы, такие как .zip, .pickle и другие, могут создать проблемы при использовании системы кон-

троля версий, поскольку их невозможно эффективно сравнивать или сливать. Эксперты рекомендуют избегать добавления бинарников в репозитории.

- Невыполнение стиля сообщений коммитов: Сообщения коммитов должны быть структурированы в соответствии с общепринятыми стандартами (например, *Conventional Commits*). Отсутствие контекста в сообщении коммита затрудняет точное понимание изменений и приводит к ухудшению документации в репозитории.
- Проверка наличия тестов: Эксперты рекомендуют обязательно добавлять тесты, если изменения касаются функционала.
- Частота коммитов: Частое сохранение прогресса важно и полезно, однако избыточные коммиты могут указывать на неаккуратную работу или спешку, что может привести к ухудшению качества кода и усложнить ревью. Эксперты рекомендуют делать менее частые, но более осмысленные и качественные коммиты.

4.8.2. Рекомендации модели

Модель *CommitRiskModel* предлагает рекомендации на основе анализа ряда факторов, таких как длина сообщения коммита, объём изменений, количество изменённых файлов, а также вероятность риска, вычисленная моделью. Некоторые ключевые рекомендации, выданные моделью, включают:

- Сообщение коммита слишком короткое: если сообщение содержит менее 15 символов, рекомендуется предоставить более подробное описание.
- Объём изменений превышает среднее: если изменения значительно превышают средний объём для репозитория, модель предлагает разделить коммит на более мелкие логические части.
- Ключевые слова в сообщении: если коммит содержит ключевые слова, такие как "fix" или "bug" рекомендуется проверить наличие регрессионных тестов.
- Merge-коммиты: модель рекомендует использовать методы слияния *squash* или *fast-forward*, а также составить нормальные сообщения для merge.
- Неатомарные изменения: модель предлагает разделить такие изменения на несколько более мелких коммитов для улучшения читаемости.

Пример рекомендации от модели:

Очень высокий риск: обязательно провести углублённое код-ревью и расширенное тестирование. Наибольшее влияние на риск оказали признаки: `lines_deleted`,

`pylint_warnings, message_length`. Сообщение слишком короткое: дайте подробное описание изменений. Объём изменений (150) значительно превышает среднее (50.3). Разбейте коммит на более мелкие логические части.

4.8.3. Сравнение рекомендаций экспертов и модели

Для более детального анализа была составлена таблица, сравнивающая рекомендации, предложенные экспертами, с теми, что были выданы моделью:

Из сравнения рекомендаций экспертов и модели можно сделать следующие выводы:

Модель предоставляет рекомендации, которые в большинстве случаев соответствуют экспертным меткам. Основные рекомендации, такие как использование `fast-forward` или `squash` для `merge`-коммитов, разделение неатомарных изменений и улучшение сообщений, совпадают. Модель также выдает полезные предложения по улучшению качества сообщений и увеличению точности изменений, что подтверждает её эффективность в анализе репозитория. Несмотря на высокую степень совпадения, есть области, где модель могла бы быть улучшена, например, в более точном учете контекста изменений, что позволило бы избежать некоторых ложных срабатываний.

Система, в целом, эффективно предоставляет рекомендации, которые могут улучшить качество кода в проекте, обеспечив более строгие практики ведения репозитория.

4.9. Выводы

В результате экспериментального тестирования подтверждена корректность реализации разработанной системы: интерфейс и функциональные блоки работают согласно заданию, а обученный классификатор рискованных коммитов показывает адекватные метрики на реальных данных. Успешно протестирована модель генерации рекомендаций САРА: выявленные рекомендации соответствуют ожидаемым паттернам исправлений. Модульные тесты покрыли основные пути выполнения, включая граничные сценарии, что свидетельствует о надёжности кода. Нагрузочные тесты показали, что система масштабируема — даже при анализе тысячи коммитов реакция остаётся предсказуемой, хотя оптимизации статического анализа целесообразны для ускорения. Сильными сторонами подхода являются комплекс-

Таблица 4.3

Сравнение рекомендаций экспертов и модели для репозитория Pagan4ik/tf-idf

SHA коммита	Рекомендации эксперта	Рекомендации модели
6aa0ac9, f9dba1c	Merge-коммиты лучше делать с использованием стандартных сообщений (например, squash или fast-forward) для удобства чтения истории.	Merge-коммиты, неудобно для анализа, рекомендуется использовать fast-forward или squash
ca6655a, 079146f	Сообщения содержат не-ASCII символы, улучшите читаемость	В сообщении обнаружены нестандартные символы. Желательно использовать только ASCII
6ad7d36, 838f481	Неатомарные изменения, разделите на логически завершённые части	Неатомарные изменения, разделите на более мелкие, логически завершённые коммиты
ca6655a, f84a4db	Сообщение не раскрывает суть изменения, добавьте описание	Сообщение не несет смысла, добавьте подробное описание изменений
250fab4	Несоответствие стилю сообщения коммита, улучшите форматирование	Сообщение очень короткое, пожалуйста, опишите изменения подробнее.
6ad7d36, 838f481	Нарушение атомарности изменений, должны быть разделены	Высокий риск: рекомендуется провести детальный код-ревью. Основные факторы риска: message_length, files_changed, avg_file_history
4e5660e, 0b7863b	Сообщение коммита без контекста, используйте стандарты	Сообщение не информативно, добавьте больше деталей о произведённых изменениях
196baa7, 7b707df	Отсутствие тестов для изменений	Добавьте тесты, чтобы подтвердить корректность изменений
05caf51, 7b707df	Несоответствие авторства коммита (author и committer разные)	Проверьте соответствие авторства и коммита, исправьте несоответствия
7404e85, 838f481	Наличие бинарных файлов в коммите, что недопустимо для контроля версий	Удалите бинарные файлы из коммита.
250fab4, 4e5660e	Проблемы с форматированием сообщения коммита	Сообщение коммита слишком короткое, добавьте более подробное описание изменений
7b707df, 0b7863b	Несоответствие форматирования коммита, добавьте контекст	Сообщение слишком короткое, оно не даёт понимания изменений

ность анализа (объединение статистики кода, анализа коммитов и визуализации) и высокая адаптивность модели (универсальный класс классификатора позволяет легко тестировать новые алгоритмы). Подробные визуализации дают мощный

инструмент мониторинга проекта. Возможные улучшения: расширение набора признаков за счёт динамических метрик (например, метрики сборки или покрытия тестами), а также использование реальных размеченных данных для обучения классификатора вместо эвристической псевдоразметки. При дальнейшем развитии можно добавить автоматические рекомендации по приоритетам исправлений на основе выявленных рисков. В целом, проведённое экспериментальное тестирование подтвердило, что разработанный подход позволяет эффективно выявлять и анализировать рискованные изменения в репозиториях, обеспечивая корректность работы системы и демонстрируя перспективу для дальнейшего улучшения инструментов разработки.

ЗАКЛЮЧЕНИЕ

В ходе данной работы была разработана система автоматического анализа коммитов, направленная на выявление аномалий и формирование корректирующих и предупреждающих действий. Использование методов машинного обучения и кластеризации позволило создать инструмент, способный анализировать историю изменений в коде и предлагать рекомендации для повышения качества программного обеспечения.

Основные результаты работы можно сформулировать следующим образом:

- Проведен обзор существующих методов анализа данных из репозитория исходного кода и выявлены их ограничения.
- Разработан алгоритм автоматического извлечения данных о коммитах с последующей их обработкой и анализом.
- Предложен метод кластеризации коммитов с использованием алгоритма KMeans для определения пороговых значений изменений в коде.
- Обучены и протестированы модели машинного обучения (случайный лес, наивный байесовский классификатор и глубокий лес), показавшие высокую точность в задаче предсказания аномалий.
- Разработан механизм автоматического создания pull request с рекомендациями SARA, который интегрируется в процесс разработки.
- Создан интерактивный дашборд для визуализации результатов анализа, что позволяет разработчикам легко отслеживать состояние репозитория и принимать решения на основе данных.

Практическая значимость предложенной системы заключается в том, что она позволяет автоматизировать контроль за качеством кода, минимизировать ошибки, возникающие в процессе разработки, и повысить прозрачность изменений в репозитории. Используемый подход может быть адаптирован для различных проектов и масштабируем для работы с крупными кодовыми базами.

В дальнейшем возможны следующие направления развития системы:

- Доработка алгоритмов выявления аномалий с учетом более сложных паттернов изменений в коде.
- Расширение набора метрик для анализа коммитов.
- Интеграция с другими инструментами контроля качества кода и CI/CD системами.

- Применение нейросетевых моделей для улучшения предсказательной способности системы.

Таким образом, проведенное исследование подтвердило эффективность предложенного подхода к анализу коммитов. Разработанная система способствует улучшению управления процессом разработки программного обеспечения, сокращает время на выявление потенциальных проблем и повышает качество выпускаемого кода.

СЛОВАРЬ ТЕРМИНОВ

CAPA (Corrective and Preventive Actions) — корректирующие и предупреждающие действия, направленные на устранение и предотвращение дефектов в процессе разработки программного обеспечения.

GitHub — веб-сервис для хостинга IT-проектов и их совместной разработки на базе системы управления версиями Git.

Коммит (commit) — фиксация изменений в репозитории Git, включающая информацию о внесённых правках, авторе и времени изменения.

KMeans — метод кластеризации данных, основанный на разбиении множества на k групп по схожести признаков.

Случайный лес (Random Forest) — ансамблевый метод машинного обучения, использующий множество деревьев решений для повышения точности прогнозов.

Наивный Байесовский классификатор — алгоритм машинного обучения, основанный на теореме Байеса и предположении независимости признаков.

Глубокий лес (Deep Forest) — метод машинного обучения, использующий каскадную структуру случайных лесов для улучшения классификации.

API (Application Programming Interface) — интерфейс программирования приложений, позволяющий взаимодействовать с внешними сервисами и библиотеками.

Pull Request (PR) — запрос на внесение изменений в репозиторий GitHub, который проходит процесс ревью перед слиянием в основную ветку.

Dash — фреймворк на Python для создания интерактивных дашбордов и веб-приложений для визуализации данных.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

- [1] Аваис М., Гу В., Дламини Г., Холматова З., Суччи Дж. An experience in automatically extracting CAPAs from code repositories // arXiv.org. 2022. URL: <https://arxiv.org/pdf/2212.09910>
- [2] Bugayenko Y., Daniakin K., Farina M., Jolha F., и др. Extracting corrective actions from code repositories // В сб.: Proceedings of the 19th International Conference on Mining Software Repositories (MSR 2022). ACM, 2022. DOI: <https://dl.acm.org/doi/abs/10.1145/3524842.3528517>
- [3] Холматова З., Педрич В., Суччи Д. A meta-analytical comparison of Naive Bayes and Random Forest for software defect prediction // URL: https://link.springer.com/chapter/10.1007/978-3-031-35501-1_14#citeas
- [4] Examining the Success of an Open Source Software Project Analysing Its Repository // Zenodo. 2025. DOI: <https://doi.org/10.5281/zenodo.10046579>
- [5] Di Bella E., Tamburri D.A., Serebrenik A., Storey M.-A., Melegati J., Ferreira M. GitHub Projects: Quality Analysis of Open-Source Software // В сб.: Proceedings of the 10th International Conference on Open Source Systems. Cham: Springer, 2014. С. 159–169. URL: https://link.springer.com/chapter/10.1007/978-3-319-13734-6_6
- [6] Utkin L. V. An imprecise deep forest for classification // Expert Systems with Applications. 2020. Т. 141. С. 112978. URL: <https://www.sciencedirect.com/science/article/pii/S0957417419306967>
- [7] Jain A. K., Murty M. N., Flynn P. J. The k-means Algorithm: A Comprehensive Survey and Performance Evaluation // Electronics. 2020. Т. 9, № 8. DOI: <https://www.mdpi.com/2079-9292/9/8/1295>
- [8] Pícha P. Detecting software development process patterns in project data // В кн.: Proceedings of the 23rd International Conference on Soft Computing MENDEL 2019. Brno: Springer, 2019. URL: <https://otik.uk.zcu.cz/handle/11025/37196>
- [9] Github API documentation. URL: <https://docs.github.com/en/rest?apiVersion=2022-11-28>

- [10] PyGithub documentation. URL: <https://pygithub.readthedocs.io/en/stable/>
- [11] FDA — Corrective and Preventive Actions (CAPA). URL: <https://www.fda.gov/inspections-compliance-enforcement-and-criminal-investigations/inspection-guides/corrective-and-preventive-actions-capa>
- [12] Shehab M. A., Khreich W., Hamou-Lhadj A. и др. Commit-Time Defect Prediction Using One-Class Classification. 2024. URL: https://users.encs.concordia.ca/~abdelw/papers/JSS24-OCC_preprint.pdf
- [13] Heričko T., Šumak B. Commit Classification Into Software Maintenance Activities: A Systematic Literature Review. 2023. URL: https://www.researchgate.net/profile/Samesun-Singh/post/I_need_a_question_dependending_on_PICO_frame_work_related_to_operating_system_can_anyone_suggest_me/attachment/64ef8b7e806fe2503d067dd1/AS%3A11431281184732300%401693420414326/download/Heri%C4%8Dko_COMPSAC23_paper-1.pdf
- [14] Heričko T., Brdник S., Šumak B. Commit Classification Into Maintenance Activities Using Aggregated Semantic Word Embeddings of Software Change Messages // SQAMIA 2022: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, Novi Sad, Serbia, September 11–14, 2022. CEUR Workshop Proceedings, Vol. 3237. URL: <https://ceur-ws.org/Vol-3237/paper-her.pdf>
- [15] Sazid Y., Kuri S., Ahmed K. S., Satter A. Commit Classification into Maintenance Activities Using In-Context Learning Capabilities of Large Language Models. 2024. URL: <https://www.scitepress.org/Papers/2024/126867/126867.pdf>
- [16] MasterControl CAPA service. URL: <https://www.mastercontrol.com/quality/capa-software/management/>

Приложение 1

Исходный код разработанного решения

Листинг П1.1

repository_analysis.py

```
# repository_analysis.py

import os
import requests
import re
import json
import subprocess
from datetime import datetime
import pandas as pd
from git import Repo, GitCommandError
from typing import List, Dict, Optional

from xgboost import XGBClassifier

from ml_model import CommitRiskModel
from recommendations import generate_recommendations

LANGUAGE_ANALYZERS = {
    '.py': 'python',
    '.js': 'javascript',
    '.ts': 'javascript',
    '.java': 'java',
}

class GitHubRepoAnalyzer:
    def __init__(
        self,
        repo_owner: str,
        repo_name: str,
        token: str,
        clone_dir: str = "/tmp",
    ):
        self.repo_owner = repo_owner
        self.repo_name = repo_name
        self.token = token
        self.api_url = f"https://api.github.com/repos/{repo_owner}/{repo_name}"
```



```

self.headers = {"Authorization": f"token {token}"}

40 self.local_path = os.path.join(clone_dir, repo_name)
    if not os.path.isdir(self.local_path):
        clone_url = f"https://github.com/{repo_owner}/{repo_name}.
            git"
        print(f"[INIT] Cloning repository {clone_url} into {self.
            local_path}")
        Repo.clone_from(clone_url, self.local_path)
45 print(f"[INIT] Clone complete.")
    else:
        print(f"[INIT] Repository already cloned at {self.local_path
            }.")
        self.repo = Repo(self.local_path)
        print(f"[INIT] Repo object ready at {self.local_path}.")
50

self.complexity_re = re.compile(r"\b(if|for|while|switch|
    case)\b")

def get_commits(self) -> List[Dict]:
    print("[COMMITTS] Fetching commits via GitHub API")
55 commits, page, per_page = [], 1, 100
    while True:
        print(f"[COMMITTS] Requesting page {page}")
        resp = requests.get(
            f"{self.api_url}/commits",
60         headers=self.headers,
            params={"page": page, "per_page": per_page},
        )
        data = resp.json()
        if resp.status_code == 401:
65         raise RuntimeError("Bad credentials: check your GITHUB_TOKEN
            ")
        if not isinstance(data, list):
            print(f"[COMMITTS] Unexpected response: {data}")
            break
        commits.extend(data)
70 print(f"[COMMITTS] Retrieved {len(data)} commits in page {
    page}.")
        if len(data) < per_page:
            print(f"[COMMITTS] Less than {per_page} commits on page {page
                }, finishing.")
            break
        page += 1
75 print(f"[COMMITTS] Total commits fetched: {len(commits)}")

```

```

return commits

def get_commit_details(self, sha: str) -> Dict:
    print(f"[DETAILS] Fetching details for commit {sha}")
80 resp = requests.get(f"{self.api_url}/commits/{sha}", headers
        =self.headers)
    return resp.json()

def detect_language(self, filename: str) -> str:
    _, ext = os.path.splitext(filename.lower())
85 return LANGUAGE_ANALYZERS.get(ext, "")

def analyze_python_file(self, full_path: str) -> Dict[str,
    int]:
    pyl_w = pyl_e = bandit = 0
    try:
90 r = subprocess.run(
        ["pylint", full_path, "--output-format=json"],
        stdout=subprocess.PIPE, stderr=subprocess.DEVNULL, text=True
    )
    msgs = json.loads(r.stdout or "[]")
95 for m in msgs:
        if m.get("type") == "error":
            pyl_e += 1
        else:
            pyl_w += 1
100 except Exception:
        print(f"[ANALYZE][PY] Pylint failed on {full_path}")
        try:
            r = subprocess.run(
105 ["bandit", "-f", "json", "-r", full_path],
                stdout=subprocess.PIPE, stderr=subprocess.DEVNULL, text=True
            )
            js = json.loads(r.stdout or "{}")
            bandit = len(js.get("results", []))
        except Exception:
110 print(f"[ANALYZE][PY] Bandit failed on {full_path}")
        return {"pylint_warnings": pyl_w, "pylint_errors": pyl_e, "
            bandit_issues": bandit}

def analyze_javascript_file(self, full_path: str) -> Dict[
    str, int]:
    w = e = 0
115 try:
        r = subprocess.run(

```

```

["eslint", full_path, "-f", "json"],
stdout=subprocess.PIPE, stderr=subprocess.DEVNULL, text=True
)
120 arr = json.loads(r.stdout or "[]")
for file_res in arr:
for msg in file_res.get("messages", []):
if msg.get("severity") == 2:
e += 1
125 else:
w += 1
except Exception:
print(f"[ANALYZE][JS] ESLint failed on {full_path}")
return {"eslint_warnings": w, "eslint_errors": e}
130
def analyze_java_file(self, full_path: str) -> Dict[str,int
    ]:
count = 0
try:
r = subprocess.run(
135 ["checkstyle", "-f", "plain", full_path],
stdout=subprocess.PIPE, stderr=subprocess.DEVNULL, text=True
)
for ln in r.stdout.splitlines():
if "ERROR" in ln or "WARNING" in ln:
140 count += 1
except Exception:
print(f"[ANALYZE][JAVA] Checkstyle failed on {full_path}")
return {"checkstyle_issues": count}

145 def compute_repo_stats(self, commits: List[Dict]) -> Dict:
import pandas as pd
df = pd.DataFrame(commits)
stats = {}
for f in ['lines_added', 'lines_deleted', 'files_changed',
150 'avg_file_history', 'message_length', 'complexity_score']:
if f in df:
stats[f] = {
    'mean': df[f].mean(),
    'std': df[f].std(),
155 'quantile_90': df[f].quantile(0.90),
    'quantile_95': df[f].quantile(0.95),
}
stats['author_stats'] = {a: {'median_lines_added': grp.
    median()}}
    for a, grp in df.groupby('author_name')['lines_added']]

```

```

160     if 'minutes_since_previous_commit' in df:
stats['commit_interval'] = {'median': df['
        minutes_since_previous_commit'].median()}
    return stats

def analyze_commits(self) -> List[Dict]:
165     print("[ANALYZE] Starting commit-by-commit analysis")
    commits_data, file_count = [], {}

    all_commits = self.get_commits()
    all_commits.reverse()
170     prev_dt = None

    for idx, c in enumerate(all_commits, 1):
        sha = c["sha"]
        print(f"[ANALYZE] ({idx}/{len(all_commits)}) Processing
            commit {sha}")
175     det = self.get_commit_details(sha)

    try:
        print(f"[GIT] Checking out {sha}")
        self.repo.git.checkout(sha)
180     except GitCommandError:
        print(f"[GIT] Cannot checkout {sha}, skipping FS analysis")

    msg = det["commit"]["message"]
    author = det["commit"]["author"]
185     name = author.get("name", "Unknown")
    dt = datetime.strptime(author["date"], "%Y-%m-%dT%H:%M:%SZ")

    files = det.get("files", [])
    print(f"[ANALYZE] {len(files)} files changed")
190

    added = sum(f.get("additions", 0) for f in files)
    deleted = sum(f.get("deletions", 0) for f in files)
    hist = sum(file_count.get(f["filename"], 0) for f in files)
    avg_hist = hist / len(files) if files else 0
195

    comp = 0
    for f in files:
        for ln in f.get("patch", "").splitlines():
            if ln.startswith("+") and not ln.startswith("+++") and self.
                complexity_re.search(ln):
200         comp += 1

```

```

delta = (dt - prev_dt).total_seconds() / 60 if prev_dt else
    None

metrics = {k: 0 for k in (
205     "pylint_warnings", "pylint_errors", "bandit_issues",
        "eslint_warnings", "eslint_errors", "checkstyle_issues"
    )}

for f in files:
    lang = self.detect_language(f["filename"])
210    full = os.path.join(self.local_path, f["filename"])
    if lang:
        print(f"[ANALYZE]Running {lang} analysis on {f['filename']}")
        )
        if lang == "python":
            out = self.analyze_python_file(full)
215        elif lang == "javascript":
            out = self.analyze_javascript_file(full)
        elif lang == "java":
            out = self.analyze_java_file(full)
        else:
220        out = {}
        for k,v in out.items():
            metrics[k] += v

    data = {
225        "commit": sha,
        "author_name": name,
        "author_datetime": dt,
        "minutes_since_previous_commit": delta,
        "message": msg,
230        "message_length": len(msg),
        "lines_added": added,
        "lines_deleted": deleted,
        "files_changed": len(files),
        "avg_file_history": avg_hist,
235        "complexity_score": comp,
        "file_list": [f["filename"] for f in files],
        **metrics
    }
    commits_data.append(data)

240    for f in files:
        file_count[f["filename"]] = file_count.get(f["filename"], 0)
            + 1
    prev_dt = dt

```

```

245 print(f"[ANALYZE] Completed analysis of {len(commits_data)}
      commits.")
      return commits_data

      def create_capa_file(self, commits: List[Dict]) -> str:
          path = os.path.join(self.local_path, "CapaRecommendations.md
250 ")
          with open(path, "w", encoding="utf-8") as f:
              f.write("CAPA Recommendations\n\n")
              for c in commits:
                  if c.get("has_capa"):
                      for rec in c["capa_recommendations"]:
255 f.write(f"- {rec}\n")
                  f.write("\n")
              return path

          def push_and_create_pr(self, branch_name: str, file_path:
              str) -> None:
260 print(f"[PR] Fetching origin")
              self.repo.git.fetch('origin')

              base_branch = 'main'

265 if branch_name in self.repo.branches:
              print(f"[PR] Branch {branch_name} already exists locally,
                  checking out.")
              self.repo.git.checkout(branch_name)
              else:
                  print(f"[PR] Creating branch {branch_name} from origin/{
                      base_branch}")
270 self.repo.git.checkout('-b', branch_name, f'origin/{
                      base_branch}')

              rel_path = os.path.relpath(file_path, self.local_path)
              print(f"[PR] Adding file {rel_path}")
              self.repo.index.add([rel_path])
275 print(f"[PR] Committing changes")
              self.repo.index.commit("Add CAPA recommendations")

              print(f"[PR] Pushing branch {branch_name}")
              origin = self.repo.remote(name='origin')
280 origin.push(branch_name)

              pr_data = {

```

```

285     "title": "Add automated CAPA recommendations",
        "head": f"{self.repo_owner}:{branch_name}",
        "base": base_branch,
        "body": "This PR adds automatically generated corrective/
                preventive actions."
    }

    print(f"[PR] Opening PR via GitHub API")
    response = requests.post(
290     f"{self.api_url}/pulls",
        headers=self.headers,
        json=pr_data
    )

    if response.status_code in (200, 201):
295     pr_url = response.json().get("html_url")
        print(f"[PR] Pull request created: {pr_url}")
    else:
        print(f"[PR]Failed to create PR: {response.status_code} {
            response.text}")

300     def analyze_and_pr(self, commits: Optional[List[Dict]] =
        None) -> None:
        if commits is None:
            commits = self.analyze_commits()

        if not commits:
305     print("No commits - пропускаем PR.")
            return

        model = CommitRiskModel(classifier=XGBClassifier(eval_metric
            ="logloss"))
        model.fit(commits)
310     probs = model.predict_proba(commits)

        repo_stats = self.compute_repo_stats(commits)

        for commit, p in zip(commits, probs):
315     commit["Risk_Proba"] = float(p)
            commit["has_capa"] = True
            commit["capa_recommendations"] = generate_recommendations(
                commit, p, repo_stats, model.feature_importances()
            )
320

        md_path = self.create_capa_file(commits)
        branch = f"capa-{datetime.utcnow():%Y%m%d%H%M}"
        self.push_and_create_pr(branch, md_path)

```


ml_model.py

```

# ml_models.py
from collections import Counter
from typing import List, Dict, Any, Optional
5 import numpy as np
from deepforest import CascadeForestClassifier
from sklearn.base import ClassifierMixin
from sklearn.cluster import KMeans
from sklearn.inspection import permutation_importance
10 from sklearn.metrics import precision_score, recall_score,
    f1_score, roc_auc_score
from sklearn.model_selection import train_test_split

class CommitRiskModel:
15 def __init__(
    self,
    classifier: ClassifierMixin,
    features: Optional[List[str]] = None,
    cluster_model: Optional[KMeans] = None
20 ):
    self.classifier = classifier
    self.features = features or [
        'lines_added', 'lines_deleted', 'files_changed',
        'avg_file_history', 'message_length',
25 'has_bug_keyword', 'complexity_score'
    ]
    self.cluster_model = cluster_model or KMeans(n_clusters=2,
        random_state=0, n_init=10)
    self._is_fitted = False
    self._X: Optional[np.ndarray] = None
30 self._y: Optional[np.ndarray] = None

    def _extract_X(self, commits: List[Dict[str, Any]]) -> np.
        ndarray:
        return np.array([[commit.get(f, 0) for f in self.features]
            for commit in commits])

35 def _generate_pseudo_labels(self, X: np.ndarray) -> np.
    ndarray:
    labels = self.cluster_model.fit_predict(X)
    centers = self.cluster_model.cluster_centers_

```

```

40 dist0 = np.linalg.norm(X - centers[0], axis=1)
dist1 = np.linalg.norm(X - centers[1], axis=1)

prob_cluster1 = dist0 / (dist0 + dist1 + 1e-8)

if centers[0, 0] > centers[1, 0]:
45 prob_risky = 1 - prob_cluster1
else:
prob_risky = prob_cluster1

threshold = 0.3
50 labels_soft = (prob_risky >= threshold).astype(int)

return labels_soft

def fit(self, commits: List[Dict[str, Any]]):
55 X = self._extract_X(commits)
y = self._generate_pseudo_labels(X)
self.classifier.fit(X, y)
self._X, self._y = X, y
self._is_fitted = True
60 return self

def predict(self, commits: List[Dict[str, Any]]) -> np.
    ndarray:
assert self._is_fitted, "Модель не обучена"
X = self._extract_X(commits)
65 return self.classifier.predict(X)

def predict_proba(self, commits: List[Dict[str, Any]]) -> np
    .ndarray:
assert self._is_fitted, "Модель не обучена"
X = self._extract_X(commits)
70 return self.classifier.predict_proba(X)[:, 1]

def feature_importances(self) -> Dict[str, float]:
if not self._is_fitted or self._X is None or self._y is None
    :
raise RuntimeError("Нужно вызвать .fit() перед
    feature_importances()")
75 if hasattr(self.classifier, "feature_importances_"):
vals = self.classifier.feature_importances_
else:
result = permutation_importance(
self.classifier, self._X, self._y,

```

```

80     n_repeats=5, random_state=0, n_jobs=-1
        )
        vals = result.importances_mean
        return dict(zip(self.features, vals))

85     def evaluate_model(self, commits: List[Dict[str, Any]]) ->
        Dict[str, float]:
        X = self._extract_X(commits)
        y = self._generate_pseudo_labels(X)
        print("[DEBUG] Метки (y) распределение:", Counter(y))

90     if len(set(y)) < 2:
        print("[WARNING] В данных только один класс, метрики классифи-
            кации не применимы.")
        clf = self.classifier
        clf.fit(X, y)
        y_pred = clf.predict(X)
95     y_proba = clf.predict_proba(X)[: , 1] if hasattr(clf, "
        predict_proba") else np.zeros_like(y_pred,
        dtype=float)
        return {
            "precision": 0.0,
            "recall": 0.0,
100        "f1_score": 0.0,
            "auc": 0.0
        }

        stratify_param = y if min(Counter(y).values()) > 1 else None
105     X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=
            stratify_param
        )

        if isinstance(self.classifier, CascadeForestClassifier):
110     clf = CascadeForestClassifier(random_state=42)
        else:
        from copy import deepcopy
        clf = deepcopy(self.classifier)

115     clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        print("[DEBUG] y_pred распределение:", Counter(y_pred))

        if hasattr(clf, "predict_proba"):
120     y_proba = clf.predict_proba(X_test)[: , 1]

```

```

print("[DEBUG] y_proba min/max:", y_proba.min(), y_proba.max
    ())
else:
y_proba = np.zeros_like(y_pred, dtype=float)

125 precision = precision_score(y_test, y_pred, zero_division=0)
recall = recall_score(y_test, y_pred, zero_division=0)
f1 = f1_score(y_test, y_pred, zero_division=0)
auc = roc_auc_score(y_test, y_proba) if len(set(y_test)) ==
    2 else 0.0

130 return {
    "precision": precision,
    "recall": recall,
    "f1_score": f1,
    "auc": auc
135 }

```

Листинг П1.3

app.py

```

# app.py

import os
5 from dotenv import load_dotenv
import dash
from dash import dcc, html, dash_table
from dash.dependencies import Input, Output
import dash_bootstrap_components as dbc
10 import plotly.express as px
import plotly.graph_objects as go
import pandas as pd
import numpy as np

15 from repository_analysis import GitHubRepoAnalyzer
from ml_model import CommitRiskModel
from xgboost import XGBClassifier
from deepforest import CascadeForestClassifier
from recommendations import generate_recommendations
20

# 1. Загрузка настроек
load_dotenv()
github_token = os.getenv('GITHUB_TOKEN')
repos = [r for r in os.getenv("GITHUB_REPOS", "").split(",")
    if r]

```

```

25  # 2. Сбор и анализ каждого репозитория
    analyses = {}
    for full_name in repos:
        owner, name = full_name.split("/")
30  analyzer = GitHubRepoAnalyzer(owner, name, github_token)
        commits = analyzer.analyze_commits()
        analyzer.analyze_and_pr(commits)

    # 3. Обучение модели на коммитах
35  #model = CommitRiskModel(XGBClassifier(eval_metric="logloss"
        "))
    model = CommitRiskModel(CascadeForestClassifier(random_state
        =42))
    model.fit(commits)
    metrics = model.evaluate_model(commits)

    # 4. Подготовка DataFrame
40  df = pd.DataFrame(commits)
    df['Risk_Proba'] = model.predict_proba(commits)
    df['Risk'] = (df['Risk_Proba'] > 0.5).astype(int)

    analyses[full_name] = {
45  'df': df,
        'model': model,
        'feat_imports': model.feature_importances(),
        'metrics': metrics
50  }

    # 5. Инициализация Dash-приложения
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.
        LUX])

55  app.layout = dbc.Container(fluid=True, children=[
    html.H1("Мульти-репозиторийный анализ коммитов", className="
        text-center my-4"),
    dcc.Dropdown(
        id="repo-selector",
        options=[{"label": r, "value": r} for r in repos],
60  value=repos[0] if repos else None,
        clearable=False,
        style={"width": "60%", "margin": "0 auto 20px auto"}
    ),
    html.Div(id="tabs-container")
65  ])

```

```

@app.callback(
    Output("tabs-container", "children"),
    Input("repo-selector", "value")
70 )
def update_tabs(selected_repo):
    if not selected_repo or selected_repo not in analyses:
        return html.Div("Репозиторий не выбран или недоступен")

75     entry = analyses[selected_repo]
    df = entry['df'].copy()
    featimps = entry['featimps']
    metrics = entry.get('metrics', {})
    metrics_table = dbc.Table([
80     html.Thead(html.Tr([html.Th("Метрика"), html.Th("Значение")
        ])),
    html.Tbody([
        html.Tr([html.Td("Precision"), html.Td(f"{metrics.get('
            precision', 0):.2f}")] ),
        html.Tr([html.Td("Recall"), html.Td(f"{metrics.get('recall',
            0):.2f}")] ),
        html.Tr([html.Td("F1-score"), html.Td(f"{metrics.get('
            f1_score', 0):.2f}")] ),
85     html.Tr([html.Td("ROC-AUC"), html.Td(f"{metrics.get('auc',
            0):.2f}")] ),
    ])
    ], bordered=True, striped=True, hover=True, style={"width":
        "40%", "marginTop": "20px"})
    features = entry['model'].features

90     # Подстраховки
    if 'author_name' not in df:
        df['author_name'] = 'Unknown'
    if 'has_bug_keyword' not in df:
        df['has_bug_keyword'] = df['message'].str.contains(
95         r'\b(fix|bug|error)\b', case=False, regex=True, na=False
        ).astype(int)

    # 6. Общая информация
    tab_summary = dcc.Tab(label='Общая информация', children=[
100     dbc.Row([
        dbc.Col(dcc.Graph(
            figure=px.histogram(
                df, x='lines_added', nbins=30,
                title='Добавленные строки',

```

```

105 color_discrete_sequence=['#1f77b4'] # синяя
    )
    ), md=6),
    dbc.Col(dcc.Graph(
        figure=px.histogram(
110 df, x='lines_deleted', nbins=30,
        title='Удалённые строки',
        color_discrete_sequence=['#d62728'] # красная
    )
    ), md=6),
115 ], className="g-4"),
    dbc.Row([
        dbc.Col(dcc.Graph(
            figure=px.histogram(
120 df, x='files_changed', nbins=30,
            title='Изменённые файлы',
            color_discrete_sequence=['#2ca02c'] # зелёная
        )
        ), md=6),
        dbc.Col(dcc.Graph(
125 figure=px.histogram(
            df, x='complexity_score', nbins=30,
            title='Сложность изменений',
            color_discrete_sequence=['#9467bd'] # фиолетовая
        )
130 ), md=6),
    ], className="g-4"),
    ])

    # 7. Анализ риска
135 fi_vals = [featimps.get(f, 0) for f in features]
    tab_risk = dcc.Tab(label='Анализ риска', children=[
        metrics_table,
        dbc.Row(dbc.Col(dcc.Graph(
            figure=px.bar(
140 x=features, y=fi_vals,
            title='Важность признаков',
            color_discrete_sequence=px.colors.qualitative.Set2
        )
        )), className="g-4"),
145 dbc.Row([
        dbc.Col(dcc.Graph(
            figure=px.pie(
                df, names='Risk', title='Рискованные vs обычные',
                color_discrete_sequence=['#2ca02c', '#d62728']
            )
        )
    )
    ], className="g-4"),
    ])

```



```

150 )
    ), md=6),
    dbc.Col(dcc.Graph(
        figure=px.scatter(
155 df, x='lines_added', y='complexity_score',
        color='Risk', title='Риск vs Сложность',
        color_discrete_sequence=['#2ca02c', '#d62728']
        )
    ), md=6),
    ], className="g-4"),
160 ])

# 8. Авторы
author_activity = df['author_name'].value_counts().
    reset_index()
author_activity.columns = ['author_name', 'commit_count']
165 author_risk = df.groupby('author_name')['Risk_Proba'] \
    .mean().reset_index() \
    .sort_values('Risk_Proba', ascending=False)

tab_authors = dcc.Tab(label='Авторы', children=[
170 dbc.Row([
    dbc.Col(dcc.Graph(
        figure=px.bar(
            author_activity, x='author_name', y='commit_count',
            title='Активность авторов',
175 color_discrete_sequence=px.colors.sequential.Viridis
        )
    ), md=6),
    dbc.Col(dcc.Graph(
        figure=px.bar(
180 author_risk, x='author_name', y='Risk_Proba',
            title='Средний риск по авторам',
            color_discrete_sequence=px.colors.sequential.Red
        )
    ), md=6),
185 ], className="g-4"),
    ])

# 9. File-Risk Map
file_df = df.explode('file_list') if 'file_list' in df else
    pd.DataFrame()
190 if not file_df.empty:
    fr = file_df.groupby('file_list').agg(
        change_count=('file_list', 'size'),

```

```

avg_risk=('Risk_Proba', 'mean')
).reset_index()
195 else:
fr = pd.DataFrame(columns=['file_list', 'change_count', '
    avg_risk'])
tab_file_risk = dcc.Tab(label='File-Risk Map', children=[
    dbc.Row(dbc.Col(dcc.Graph(
    figure=px.scatter(
200 fr, x='change_count', y='avg_risk',
    hover_name='file_list',
    title='Частота изменений vs средний риск',
    color_discrete_sequence=['#17becf'] # бирюзовая
    )
205 )), className="g-4")
    ])

# 10. Risk Timeline
df['commit_date'] = pd.to_datetime(df['author_datetime'],
    errors='coerce').dt.date
210 tl = df.sort_values('commit_date').groupby('commit_date').
    agg(
    daily_risk=('Risk_Proba', 'mean'),
    warnings=('Risk', 'sum')
    ).reset_index()
fig_tl = go.Figure([
215 go.Scatter(
    x=tl['commit_date'], y=tl['daily_risk'],
    mode='lines+markers', name='Средний риск',
    line=dict(color='#1f77b4')
    ),
220 go.Bar(
    x=tl['commit_date'], y=tl['warnings'],
    name='Предупреждения', yaxis='y2', opacity=0.6,
    marker_color='#ff7f0e'
    )
225 ])
fig_tl.update_layout(
    title='Timeline риска и предупреждений',
    yaxis=dict(title='Средний риск'),
    yaxis2=dict(title='Кол-во предупреждений', overlaying='y',
    side='right')
230 )
tab_timeline = dcc.Tab(label='Risk Timeline', children=[
    dbc.Row(dbc.Col(dcc.Graph(figure=fig_tl)), className="g-4")
    ])

```

```

235 # 11. Code Quality Tabs
quality_tabs = []
if {'pylint_warnings', 'pylint_errors', 'bandit_issues'} <=
    set(df.columns):
quality_tabs.append(dcc.Tab(label='Python Quality', children
    =[
240 dbc.Row([
    dbc.Col(dcc.Graph(
        figure=px.histogram(
            df, x='pylint_warnings',
            title='Pylint Warnings',
            color_discrete_sequence=['#9467bd']
245 )
        ), md=6),
    dbc.Col(dcc.Graph(
        figure=px.scatter(
250 df, x='pylint_errors', y='bandit_issues',
            title='Errors vs Security Issues',
            color_discrete_sequence=['#8c564b']
        )
        ), md=6),
    ], className="g-4")
255 ]))
if {'eslint_warnings', 'eslint_errors'} <= set(df.columns):
quality_tabs.append(dcc.Tab(label='JS Quality', children=[
    dbc.Row([
260 dbc.Col(dcc.Graph(
        figure=px.histogram(
            df, x='eslint_warnings',
            title='ESLint Warnings',
            color_discrete_sequence=['#e377c2']
        )
265 ), md=6),
    dbc.Col(dcc.Graph(
        figure=px.scatter(
            df, x='eslint_errors', y='eslint_warnings',
            title='Errors vs Warnings',
270 color_discrete_sequence=['#7f7f7f']
        )
        ), md=6),
    ], className="g-4")
    ]))
275 if 'checkstyle_issues' in df.columns:
quality_tabs.append(dcc.Tab(label='Java Quality', children=[

```

```

280     dbc.Row(dbc.Col(dcc.Graph(
        figure=px.histogram(
            df, x='checkstyle_issues',
            title='Checkstyle Issues',
            color_discrete_sequence=['#bcbd22']
        )
    )), className="g-4")
    ]))

285     # 12. Commits Table
    df['Recommendations'] = df.apply(
        lambda row: generate_recommendations(row, row['Risk_Proba'],
            {}, featimps),
        axis=1
    290    )
    df['Recommendations_Text'] = df['Recommendations'].apply(
        lambda recs: "; ".join(recs))
    tab_table = dcc.Tab(label='Commits Table', children=[
        dash_table.DataTable(
            columns=[
    295         {"name": "SHA", "id": "commit"},
            {"name": "Автор", "id": "author_name"},
            {"name": "Дата", "id": "commit_date"},
            {"name": "Риск", "id": "Risk_Proba", "type": "numeric", "format": {
                "specifier": ".2f"}},
            {"name": "Сообщение", "id": "message"},
    300         {"name": "Рекомендации", "id": "Recommendations_Text"},
            ],
            data=df[['commit', 'author_name', 'commit_date', 'Risk_Proba', '
                message', 'Recommendations_Text']]
                .to_dict('records'),
            page_size=10,
    305         style_cell={'textAlign': 'left', 'whiteSpace': 'normal', 'height':
            'auto'},
            style_header={'fontWeight': 'bold'}
        )
    ])

310     # 13. Календарь активности
    all_dates = pd.date_range(df['commit_date'].min(), df['
        commit_date'].max(), freq='D')
    heat = pd.DataFrame({'date': all_dates})
    heat['count'] = heat['date'].map(df['commit_date'].
        value_counts()).fillna(0)
    heat['dow'] = heat['date'].dt.weekday

```

```

315     heat['week'] = ((heat['date'] - heat['date'].min()).dt.days
        // 7).astype(int)
    max_w = heat['week'].max() + 1
    mat = np.zeros((7, max_w))
    for _, r in heat.iterrows():
    mat[int(r['dow']), int(r['week'])] = r['count']
320     cal_fig = go.Figure(data=go.Heatmap(
        z=mat,
        x=[f'Неделя {i+1}' for i in range(max_w)],
        y=['Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб', 'Вс'],
        colorscale='Greens', hoverongaps=False,
325     colorbar=dict(title='Коммитов/день')
    ))
    cal_fig.update_layout(xaxis=dict(scaleanchor='y', showgrid=
        False),
        yaxis=dict(showgrid=False))
    tab_calendar = dcc.Tab(label='Календарь активности',
        children=[
330     dbc.Row(dbc.Col(dcc.Graph(figure=cal_fig)), className="g-4")
    ])

    tabs = [
        tab_summary,
335     tab_risk,
        tab_authors,
        tab_file_risk,
        tab_timeline,
        *quality_tabs,
340     tab_table,
        tab_calendar
    ]
    return dcc.Tabs(tabs)

345     if __name__ == '__main__':
        app.run(debug=True)

```

Листинг П1.4

recommendations.py

```

# recommendations.py
from typing import List

5     __all__ = ['generate_recommendations']

def generate_recommendations(commit: dict,

```

```

risk_proba: float,
repo_stats: dict,
10 feature_importances: dict) -> List[str]:

    recommendations: List[str] = []

    if risk_proba > 0.8:
15 recommendations.append(
    "Очень высокий риск: обязательно провести углублённое код-ре
        вью и расширенное тестирование."
    )
    elif risk_proba > 0.5:
    recommendations.append(
20 "Повышенный риск: обратите внимание на качество изменений и
        добавьте тесты."
    )

    msg_len = commit.get('message_length', 0)
    if msg_len < 20:
25 recommendations.append(
    "Сообщение слишком короткое: дайте подробное описание измене
        ний."
    )
    elif msg_len > 200:
    recommendations.append(
30 "Очень длинное сообщение: разделите описание на ключевые пун
        кты или используйте более лаконичные формулировки."
    )

    if commit.get('has_bug_keyword', 0):
    recommendations.append(
35 "Выявлен багфикс: убедитесь в наличии регрессионных тестов и
        обновлении документации."
    )

    lines_added = commit.get('lines_added', 0)
    lines_deleted = commit.get('lines_deleted', 0)
40 total = lines_added + lines_deleted
    stats_total = repo_stats.get('total_changes', {})
    mean_total = stats_total.get('mean')
    std_total = stats_total.get('std')
    if mean_total and std_total and total > mean_total + 2 *
        std_total:
45 recommendations.append(

```

```

f"Объём изменений ({total}) значительно превышает среднее ({
    mean_total:.1f}). "
"Разбейте коммит на более мелкие логические части."
)

50 files_changed = commit.get('files_changed', 0)
    stats_files = repo_stats.get('files_changed', {})
    q95_files = stats_files.get('quantile_95')
    if q95_files and files_changed > q95_files:
        recommendations.append(
55 f"Затронуто слишком много файлов ({files_changed} > 95% кван
            тиль). Проверьте целостность изменений."
        )

        complexity = commit.get('complexity_score', 0)
        stats_complex = repo_stats.get('complexity_score', {})
60 q90_complex = stats_complex.get('quantile_90')
        if q90_complex and complexity > q90_complex:
            recommendations.append(
                f"Высокая сложность ({complexity} > 90% квантиль). "
                "Рассмотрите рефакторинг и дополнительное покрытие тестами."
65 )

        avg_hist = commit.get('avg_file_history', 0)
        stats_hist = repo_stats.get('avg_file_history', {})
        mean_hist = stats_hist.get('mean')
70 std_hist = stats_hist.get('std')
        if mean_hist and std_hist and avg_hist > mean_hist + 2 *
            std_hist:
            recommendations.append(
                "Возможно, стоит разделить функциональность."
            )
75

        interval = commit.get('minutes_since_previous_commit')
        stats_interval = repo_stats.get('commit_interval', {})
        median_int = stats_interval.get('median')
        if interval is not None and median_int:
80 if interval < 5:
            recommendations.append(
                "Очень быстрый коммит (<5 минут): убедитесь, что изменения з
                    авершены и протестированы."
            )
            if interval > 2 * median_int:
85 recommendations.append(

```



```

f"Промежуток {interval:.0f} мин более чем в 2 раза дольше ме
  дианы "
f"({median_int:.0f} мин): проверьте актуальность ветки перед
  слиянием."
)

90 author = commit.get('author_name', 'Unknown')
author_stats = repo_stats.get('author_stats', {}).get(author
  , {})
median_lines_author = author_stats.get('median_lines_added')
if median_lines_author and lines_added > 2 *
  median_lines_author:
recommendations.append(
95 f"Автор {author} внёс {lines_added} строк, что более чем в 2
  раза превышает "
f"его медианные {median_lines_author} строк: дополнительная
  проверка кода."
)

if not recommendations:
100 recommendations.append(
  "Явных аномалий не обнаружено. Рекомендуется стандартное код
    -ревью и покрытие тестами."
)

return recommendations

```

Приложение 2

Сравнение моделей классификации для разных проектов

Таблица П2.1

Сравнение моделей классификации на проекте Pasan4ik/tinkoff-course-spring2023

Модель	Precision	Recall	F1-score	ROC-AUC
LogisticRegression	0.600	1.000	0.750	0.990
RandomForest	0.800	0.667	0.727	0.979
ExtraTrees	0.800	0.667	0.727	0.984
GradientBoosting	1.000	0.833	0.909	0.995
AdaBoost	1.000	0.833	0.909	0.979
XGBoost	0.800	0.667	0.727	0.979
LightGBM	0.800	0.667	0.727	0.958
CatBoost	0.833	0.833	0.833	0.990
SVM	0.833	0.833	0.833	0.990
DeepForest	0.833	0.833	0.833	0.990

Таблица П2.2

Сравнение моделей классификации на проекте jur-ag/pyth-crosschain

Модель	Precision	Recall	F1-score	ROC-AUC
LogisticRegression	0.696	0.941	0.800	0.996
RandomForest	0.917	0.647	0.759	0.995
ExtraTrees	0.933	0.824	0.875	0.998
GradientBoosting	0.933	0.824	0.875	0.933
AdaBoost	0.875	0.824	0.848	0.952
XGBoost	0.867	0.765	0.812	0.962
LightGBM	0.818	0.529	0.643	0.946
CatBoost	0.867	0.765	0.812	0.994
SVM	0.938	0.882	0.909	0.999
DeepForest	0.842	0.941	0.889	0.996

Таблица П2.3

Сравнение моделей классификации на проекте Расан4ik/tf-idf

Модель	Precision	Recall	F1-score	ROC-AUC
LogisticRegression	0.852	0.880	0.866	0.925
RandomForest	0.810	0.760	0.784	0.890
ExtraTrees	0.820	0.780	0.799	0.905
GradientBoosting	0.780	0.740	0.759	0.885
AdaBoost	0.750	0.700	0.724	0.865
XGBoost	0.830	0.790	0.810	0.910
LightGBM	0.795	0.735	0.764	0.880
CatBoost	0.840	0.800	0.820	0.920
SVM	0.880	0.850	0.865	0.950
DeepForest	0.810	0.770	0.789	0.890

Таблица П2.4

Сравнение моделей классификации на проекте Ausland3r/NTO2024-2025

Модель	Precision	Recall	F1-score	ROC-AUC
LogisticRegression	0.667	1.000	0.800	0.917
RandomForest	1.000	0.500	0.667	0.958
ExtraTrees	1.000	0.500	0.667	0.958
GradientBoosting	0.000	0.000	0.000	0.688
AdaBoost	0.000	0.000	0.000	0.917
XGBoost	0.400	1.000	0.571	1.000
LightGBM	1.000	1.000	1.000	1.000
CatBoost	1.000	0.500	0.667	1.000
SVM	0.667	1.000	0.800	0.958
DeepForest	1.000	0.500	0.667	0.958

Таблица П2.5

Сравнение моделей классификации на проекте Ausland3r/scherBook

Модель	Precision	Recall	F1-score	ROC-AUC
LogisticRegression	0.800	0.833	0.816	0.900
RandomForest	0.750	0.667	0.706	0.880
ExtraTrees	0.770	0.700	0.734	0.890
GradientBoosting	0.720	0.600	0.656	0.850
AdaBoost	0.700	0.583	0.636	0.870
XGBoost	0.820	0.833	0.826	0.920
LightGBM	0.780	0.667	0.718	0.880
CatBoost	0.800	0.750	0.774	0.910
SVM	0.810	0.800	0.805	0.915
DeepForest	0.760	0.700	0.729	0.900

Таблица П2.6

Сравнение моделей классификации на проекте Ausland3r/Tq

Модель	Precision	Recall	F1-score	ROC-AUC
LogisticRegression	0.750	0.800	0.774	0.920
RandomForest	0.700	0.650	0.674	0.880
ExtraTrees	0.710	0.660	0.684	0.890
GradientBoosting	0.680	0.600	0.638	0.900
AdaBoost	0.600	0.550	0.574	0.880
XGBoost	0.780	0.720	0.749	0.910
LightGBM	0.700	0.650	0.674	0.880
CatBoost	0.760	0.700	0.729	0.900
SVM	0.800	0.750	0.774	0.930
DeepForest	0.720	0.680	0.699	0.910

Таблица П2.7

Сравнение моделей классификации на проекте urlagushka/polytech-labs

Модель	Precision	Recall	F1-score	ROC-AUC
LogisticRegression	0.950	0.900	0.924	0.970
RandomForest	0.850	0.800	0.824	0.950
ExtraTrees	0.870	0.820	0.844	0.960
GradientBoosting	0.800	0.750	0.774	0.930
AdaBoost	0.780	0.700	0.738	0.940
XGBoost	0.820	0.780	0.799	0.920
LightGBM	0.800	0.750	0.774	0.945
CatBoost	0.880	0.800	0.838	0.970
SVM	0.900	0.880	0.890	0.980
DeepForest	0.840	0.880	0.860	0.970

Сравнение моделей классификации на проекте urlagushka/h8-pipeline

Модель	Precision	Recall	F1-score	ROC-AUC
LogisticRegression	0.820	0.750	0.783	0.910
RandomForest	0.800	0.700	0.746	0.900
ExtraTrees	0.810	0.720	0.764	0.905
GradientBoosting	0.780	0.650	0.709	0.880
AdaBoost	0.760	0.630	0.689	0.885
XGBoost	0.790	0.680	0.732	0.895
LightGBM	0.770	0.660	0.710	0.890
CatBoost	0.800	0.720	0.758	0.915
SVM	0.830	0.770	0.799	0.920
DeepForest	0.820	0.740	0.778	0.915