ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
Факультет Санкт-Петербургская школа физико-математических и компьютерных наук

Лобанов Артём Владиславович

РЕШЕНИЕ ЗАДАЧ ПРОГРАММНОЙ ИНЖЕНЕРИИ НА ОСНОВЕ ОБЪЕДИНЕНИЯ ИНФОРМАЦИИ ИЗ РАЗНЫХ ИСТОЧНИКОВ

Выпускная квалификационная работа
по направлению подготовки 01.04.02 Прикладная математика и информатика
образовательная программа «Программирование и анализ данных»

Рецензент
к.ф.-м.н.,
_____
А.А. Фильченков

Научный руководитель
к.т.н,
_____
Т.А. Брыксин

Санкт-Петербург 2021

Оглавление

Введение

Программирование проникло во все сферы нашей жизни. Для понимания масштаба этого явления можно обратиться к статистике сервиса GitHub[1]. За 2020 год 56 миллионов активных пользователей создали более 60 миллионов новых проектов и совершили почти два миллиарда коммитов. Разработка, тестирование и поддержка программного кода — очень трудоёмкие и дорогие процессы. Оптимизацией этих процессов занимается *программная инженерия*. В рамках этой дисциплины изучается огромное количество прикладных задач, таких как автодополнение кода [95, 103], поиск ошибкоопасных мест [98], оценка тестового покрытия [13] и т.д. Применение методов машинного обучения позволило приблизиться к решению некоторых из перечисленных задач. Однако при применении моделей машинного обучения возникает ряд проблем, специфичных для данной области.

Несмотря на внешнее сходство исходного кода и естественного языка, эти типы данных имеют важные структурные отличия. С одной стороны, исходный код имеет намного более строгую формальную структуру, нежели текст. Это даёт простор для применения сложных моделей, анализирующих абстрактные синтаксические деревья или графы потока управления. С другой стороны, особенности именования методов и переменных в исходном коде приводят к намного бóльшему количеству в нём уникальных токенов [11], что приводит к проблемам с размером словаря и несловарными токенами в некоторых моделях машинного обучения.

Ещё одна специфичная сложность сложность заключается в гетерогенности данных. В разных задачах алгоритмы должны работать с исходным кодом, текстом на естественном языке, изменениями кода или текста, результатами запуска тестов, логов выполнения программ и т.п. Поэтому важной задачей является изыскание способов комбинировать информацию из разных по форме источников. В данной работе эта задача будет рассмотрена на двух конкретных примерах из разных областей программной инженерии.

[1]Статистика GitHub за 2020 год: `https://octoverse.github.com/`

Постановка задачи

Целью данной работы исследование применимости ансамблей для объединения информации из разных источников в задачах программной инженерии.

Для достижения цели необходимо решить следующие задачи:

1. Выделить основные типы информации в задачах программной инженерии.
2. Выбрать архитектуру решения, позволяющую эффективно объединять информацию из разных источников.
3. Выбрать модели, специфичные для выделенных типов информации.
4. Провести апробацию предложенной архитектуры на проблеме предсказания тегов задач спортивного программирования.
5. Провести апробацию предложенной архитектуры на задаче определения коммитов, исправляющих ошибки.

1 Обзор литературы

1.1 Программная инженерия

В середине двадцатого века стремительный рост вычислительных мощностей и сложности решаемых с помощью компьютеров задач привели к *кризису программного обеспечения*. Так назвали совокупность проблем, связанных с трудностью реализации больших программных проектов с поддержанием высокого уровня качества и надёжности кода. Широкую огласку эта ситуация получила благодаря конференции *NATO Software Engineering Conference* [72] в 1968. На тот момент программирование было достаточно новой отраслью, и ещё не было устоявшихся принципов разработки продуктов. Однако, во многих аспектах процесс разработки программ схож с разработкой сложных электронных устройств, а опыт в работе над последними у человечества был уже достаточно обширный. Вопрос переиспользования и переосмысления этого опыта активно обсуждался на конференции. Например, в докладе *Mass-produced software components* [69] обращалось внимание на то, что в электронике каждый элемент разрабатывается отдельно и потом активно применяется в разных ситуациях. Эта идея модульности легла в основу компонентно-ориентированной парадигмы программирования — идеи о разделении программы на независимые переиспользуемые компоненты с уникальной функциональностью. Такой подход позволяет существенно снизить сложность проектирования и разработки. Важным итогом конференции стало появление новой дисциплины [87], посвящённой изучению, систематизации и оптимизации процессов проектирования, документирования, разработки, тестирования и внедрения программного обеспечения. Она получила название *программная инженерия*.

С тех пор многое изменилось. Появилась целая плеяда парадигм программирования (например, функциональное программирование, объектно-ориентированное программирование и другие), было предложено множество методологий разработки (например, водопадная или Agile подходы), разработаны спецификации диаграмм для описания разных аспектов проектов (диаграммы управления, диаграммы классов и так далее), реализованы си-

стемы контроля версий с поддержкой удалённых репозиториев и интеграцией с системами непрерывной сборки, разработаны специальные инструменты для одновременной работы с кодовой базой и различные решения, позволяющие отслеживать статус различных задач и сообщать об ошибках. В результате вокруг кода появилось большое количество дополнительных данных.

1.2 Системы контроля версий

Системы контроля версий — это обобщённое название инструментов, позволяющих отслеживать изменения файлов и возвращаться к их предыдущим версиям. Прежде чем углубиться в историю развития подобных систем, введём определения специфичных для этой темы терминов. **Репозиторий** (repository) — хранилище данных, необходимых для работы системы контроля версий. В нём хранятся файлы проекта, а также информация о всех предыдущих версиях. Для экономии памяти применяются различные алгоритмы сжатия данных. Репозиторий может быть локальным (т. е. храниться на компьютере пользователя) или удалённым (т. е. храниться на удалённом сервере). **Коммит** (commit) — сохранённое в какой-то момент состояние файлов проекта. Каждый коммит хранит информацию о предыдущем коммите. Таким образом, коммиты образуют ориентированный ациклический граф. Также коммит содержит написанное автором текстовое описание смысла внесённых в проект изменений. На практике под коммитом часто поимают не версию проекта, а её отличие от предыдущей, так как основную смысловую нагрузку имеют именно внесённые изменения. Именно такое понимание термина будет использоваться нами в дальнейшем. **Ветка** — это альтернативная версия истории проекта. Ветки не влияют друг на друга, а потому позволяют свободно вести параллельную работу. После окончания работы над какой-то функциональностью пользователь может слить свою ветку с основной, тем самым скопировав в неё все проделанные изменения. Если при этом выяснилось, что в основной и альтернативной ветке были противоречащие друг другу модификации файлов, то такая ситуация называется **конфликт**. **Пул-реквест** (pull request) — запрос на слияние веток

репозитория. Владелец репозитория может его принять или отклонить. Пулреквесты позволяют удобно проводить **ревью кода** (code review) — процесс проверки написанного кода на соответствие стандартам качества и стилистике проекта.

Исторически первой [93] реализацией системы контроля версий с открытым исходным кодом было решение от Bell Labs под названием *Source Code Control System* [90]. К сожалению, оно имело довольно сложный и запутанный интерфейс, для выполнения простых действий приходилось выполнять множество команд. Существенно проще в использовании была сменившая её разработка *Revision Control System* [104] от университета Пёрдью. Она автоматизировала множество побочных процессов, связанных с сохранением и извлечением данных, логированием действий и идентификацией авторов изменений, предоставляя пользователям более высокоуровневый интерфейс. К сожалению, RCS была слабо приспособлена для коллективной работы над проектом. Прежде чем изменять какой-то файл, пользователь должен был заблокировать его для редактирования другими пользователями. Попытки решить проблему конкурентных модификаций привели к созданию *Concurrent Versioning System* [108]. CVS частично базировалась на RCS — использовала такой же формат хранения файлов и алгоритмы сжатия данных. Однако, в отличие от всех предыдущих вариантов, CVS имела клиент-серверную архитектуру. Такое решение позволяло множеству пользователей получать от сервера локальные копии проектов, независимо их модифицировать и отправлять изменения на сервер. В случае, если сервер обнаруживал конфликтующие изменения, он отклонял последнюю попытку загрузки локальных изменений и просил пользователя исправить конфликты с актуальной версией. Впоследствии CVS была вытеснена *Subversion* [82]. В ней было исправлено множество недочётов CVS: улучшена работа с директориями, добавлена поддержка эффективных алгоритмов хранения истории изменений для бинарных файлов, оптимизирован трафик между клиентом и сервером и т.д. Наконец, в 2005 Линукс Торвальдс представил миру новую систему контроля версия — git [49]. Сейчас существует множество сервисов, позволяющих создавать удалённые git-репозитории, одним из самых попу-

лярных из них является GitHub[2].

Повсеместное использование систем контроля версий породило множество задач, в которых применяется анализ кода, обработка естественного языка и выделение изменений. Примерами таких задач служат классификация коммитов [66], генерация описаний коммитов [74], разметка пулреквестов [89] и т.п.

1.3 Обработка естественного языка

В лингвистике языки, использующиеся для общения людей, называют естественными. Например, русский и английский являются естественными, а Java или язык математических выкладок — нет. Данные на естественных языках окружают нас повсюду — переписки и посты в социальных сетях, книги, информация на сайтах и т. д. Для решения многих задач, связанных с естественным языком, используют машинной обучение. Направление машинного обучения, изучающее проблемы распознавания и синтеза речи, анализа и генерации текстов на естественных языках, называют обработкой естественного языка (Natural Language Processing или NLP). В рамках этого направления решаются такие важные задачи, как машинный перевод, анализ тональности текста, построение вопросно-ответных систем, информационный поиск, автоматическое реферирование и т.д.

Классификация текстов

Многие прикладные задачи в области обработки естественного языка формулируются как задачи классификации, то есть требуется предложенный текст отнести к одной из заранее известных категорий. Такая постановка используется при анализе эмоциональной окраски отзывов, выявлении оскорблений в социальных сетях, определении намерений пользователя в диалогах с чат-ботами. Существует множество различных подходов к классификации, но прежде чем их применять, необходимо как-то представить текст в численном виде, с которым работают модели машинного обучения, то есть векторизовать.

[2]GitHub: `https://github.com/`

Самым простым способом векторизации текста является так называемый метод “мешок слов”. В рамках этого подхода составляется словарь токенов, в котором каждому токену сопоставляется уникальный идентификатор, а затем для текста генерируется вектор, в котором i-ый элемент обозначает количество токенов с идентификатором i в тексте. Размер итогового вектора будет совпадать с размером словаря. Тут есть некоторая проблема, связанная с существованием множества форм слов. Например, при простой токенизации текста слова *груша* и *грушу* будут считаться различными. Это сильно увеличивает размер словаря и усложняет классификацию. Решить эту проблему помогает нормализация текста, то есть приведение всех форм одного слова к некоторому общему виду. Для этого существует два классических метода.

Первый метод называется стемминг (stemming) и обозначает выделение основы слова. Обычно алгоритмы стемминга пытаются с помощью эвристических правил отбросить окончание, иногда похожим образом пытаются удалять и приставки. В приведённом выше примере слова *груша* и *грушу* будут сведены к одинаковой основе *груш*. Подобные решения отличаются высокой скоростью работы, но, к сожалению, точность полученных основ оставляет желать лучшего. Иногда формы одного слова не удаётся привести к общей форме. Например, токены *идёт* и *шёл* являются различными формами слова *идти*, однако выделить из них общий корень с помощью стемминга не получится. Бывают и обратные ситуации. Например, существительные *пастъ* и *паста* будут усечены до общей формы *паст*, хотя несут совершенно разный смысл.

Второй метод называется лемматизация и обозначает приведение слов к нормальной форме. Например, для имён существительных нормальной считается форма, соответствующая единственному числу именительного падежа (для слов *груша* и *грушу* нормальной формой будет слово *груша*). Алгоритмы лемматизации сначала анализируют текст и пытаются определить, к каким частям речи относятся отдельные слова в тексте, а затем, в зависимости от части речи, применяют различные методы восстановления нормальной формы. Такой подход намного точнее стемминга, но требует значительно больше вычислений и, как следствие, времени.

Вместе с тем, нормализация токенов — не единственная проблема мешка слов. Если честно посчитать количество слов для текста, то самыми частыми окажутся предлоги, местоимения и прочие частотные слова, не несущие почти никакой информации о сути текста. Самым простым решением является удаление слов из выборки. Это можно делать либо с помощью отсечения по частоте встречаемости, либо с помощью специальных словарей стоп-слов. Однако есть и более гибкие подходы. Например, можно учитывать слова с различными весами. Для частых слов вес будет маленький, а для редких — большой. Эта идея реализована в подходе TF-IDF (term frequency / inverse document frequency), в котором количество использований токена в тексте делится на количество документов, в которых этот токен встречается.

И, наконец, при подсчёте количества использований слов никак не учитывается их порядок. Таким образом, векторное представление фразы «Я люблю груши и ненавижу яблоки» будет совпадать с векторным представлением фразы «Я люблю яблоки и ненавижу груши», хотя смысл этих предложений диаметрально противоположен. К сожалению, это является фундаментальной проблемой данного подхода. Частично её можно решить с помощью добавления в словарь N-грамм — кортежей из N последовательных слов. Например, можно дополнительно считать количество словосочетаний «люблю яблоки» и «люблю груши». Это позволит различать фразы из примера выше, но сильно увеличит размер словаря (в худшем случае из N уникальных слов мы получим $O(N^2)$ словосочетаний). Но даже в этом случае полностью восстановить смысл фразы по нашему векторному представлению не получится.

Для учёта информации о порядке слов применяются рекуррентные нейронные сети (*Recurrent neural network*, или RNN). RNN спроектированы для работы с последовательностями. На вход такая модель принимает вектор состояния и очередной элемент входной последовательности, а на выход даёт новый вектор состояния. Таким образом, последовательно передавая элементы, можно обработать последовательность любой длинны. Самыми распространёнными вариантами RNN являются *Long Short-Term Memory* (LSTM) и *Gated Recurrent Units* (GRU). Данные модели активно применялись для решения таких задач как построение языковых моделей [101], определения

фейковых новостей [42], определения эмоциональной окраски текста [106] и т. д. Для улучшения результатов классификации вместо конечного вектора состояния или выходного значения модели после обработки последнего элемента последовательности используют [3] агрегированные с помощью механизма внимания вектора состояний или выходы моделей после каждого из элементов. Идея механизма внимания довольно простая: если имеется много векторов, а требуется получить один итоговый, то следует просто просуммировать все вектора с некоторыми весами, показывающими важность каждого конкретного элемента.

Механизм внимания оказался очень удачной идеей, заметно улучшившей результаты работы рекуррентных нейронных сетей. Более того, в 2017 году Vaswani et al. [4] показали, что этот механизм самодостаточный. Предложенная ими архитектура получила название Transformer и состояла из нескольких слоёв механизма внимания, перемежающихся полносвязными слоями. Однако именно в такой формулировке мы возвращаемся к тем же проблемам, которые были у мешка слов, — отсутствие учёта порядка слов и необходимость нормализации токенов.

Первую проблему авторы решили с помощью *кодирования позиций*. Этот подход подразумевает добавление к каждому вектору токенов специального вектора, зависящего от позиции токена в тексте. В работе Vaswani et al. [4] использовался вектор из значений синуса и косинуса с разными периодами, посчитанных в точках, зависящих от абсолютной позиции конкретного токена в тексте. Также существуют вариации с помощью обучаемых векторов позиций [25]. Эти вектора учитываются при использовании механизма внимания, позволяя модели учитывать абсолютные и относительные позиции токенов.

Вторая проблема, связанная с нормализацией слов, решается с помощью специальных методов разбиения токенов на фрагменты. Популярными решениями являются WordPieces [94, 44] и Byte Pair Encoding (BPE) [96, 39]. Их идея состоит в построении словаря фрагментов на основе информации о частоте встречаемости сочетаний. Словарь инициализируется отдельными буквами, а затем на каждой итерации находится пара фрагментов из словаря, которые чаще всего встречаются в тексте подряд и их конкатенация

тоже добавляется в словарь. В итоге получается словарь, содержащий часто используемые слова и морфемы и позволяющий разбить любой текст на словарные фрагменты. Это с одной стороны позволяет ограничить размер словаря, а с другой эффективно справляться с проблемой несловарных токенов.

Однако модели на основе трансформеров довольно большие, и для их обучения нужно много данных, а выборки примеров для конкретных прикладных задач обычно не очень большие. Для решения этой проблемы можно воспользоваться подходом [56], который называется *тонкой настройкой*, или *дообучением*. Именно эта идея использовалась в популярной модели BERT [9]. Модель обучалась на двух синтетических задачах: восстановления пропущенных слов в тексте и определения, являются ли два предложения последовательными. Затем полученную модель дообучили на относительно небольших выборках для популярных в области обработки естественного языка задачах и показали, что BERT превосходит существующие аналоги в каждой из них.

1.4 Анализ исходного кода

Применение методов машинного обучения может существенно упростить решение многих задач программной инженерии. Однако, это довольно специфичная область — большую часть данных составляет исходных код проектов. Как можно его анализировать?

Самой простой вариант — просто воспринимать исходный код как текст и применять подходы из области обработки естественного языка. Такой подход действительно применяется [48, 10, 109], однако в этом случае игнорируются знания о структуре кода. Разработка специализированных подходов была вопросом времени.

Первый специфичный для программного кода подход — это использование метрик. Метрики кода появились на заре становления программной инженерии [36] и изначально проектировались для оценки различных аспектов кода. Однако, они нашли довольно широкое применение в машинном обучении. Их использовали для поиска ошибкоопасных мест [54, 20, 85, 105],

определения запахов кода [6, 24, 58], выявления содержащих ошибки коммитов [73, 92, 59], рекомендации рефакторнигов [88] и т.д. У метрик есть несколько несомненных преимуществ. Т. к. все метрики придуманы людьми, то, с одной стороны, они извлекают из кода какую-то действительно важную с точки зрения человека информацию, а с другой — имеют понятный смысл и легко интерпретируются. К сожалению, практика показывает, что информации, извлекаемой метриками, оказывается недостаточно для конкуренции с более современными методами. В исследовании Wei et al. [28] было проведено сравнение нескольких подходов к задаче поиска ошибкоопасных мест и показано, что современные подходы к векторизации кода превосходят подходы, основанные на использовании метрик. В задаче выявления ошибкоопасных коммитов лучшие результаты также показывают модели глубокого обучения [27].

Однако, код отличается от текста наличием строгой формальной структуры, что позволяет строить по нему *абстрактное синтаксическое дерево*. Абстрактное синтаксическое дерево (Abstract Syntax Tree, или AST) — это представление программного кода в виде дерева, содержащее полную информацию о семантике кода, но опускающее лишние подробности (форматирование, скобки, запятые). Интерпретируя код как простую последовательность токенов, мы игнорируем это априорное знание. Для работы с AST иногда используют TreeLSTM [7]. Идея этого подхода в применении LSTM к линеаризации AST. Чтобы задать какую-нибудь последовательность вершинам AST, обычно используют обход в ширину. Для каждой вершины на вход ячейке LSTM подаётся информация о самой вершине, усреднённые или сконкатенированные вектора её детей и вектор состояния. Недостатком такого подходя является чрезвычайная сложность в реализации и обучении.

Существуют и другие способы использования AST. code2vec [111] — это подход к векторизации кода, основанный на извлечении путей между листьями из абстрактного синтаксического дерева кода. Каждый путь характеризуется тройкой из токена первого листа дерева, последовательности типов вершин пути и токена второго листа. Каждому уникальному элементу из этой тройки сопоставляется собственный обучаемый вектор. Вектора элементов тройки конкатенируются и пропускаются через полносвязный слой

для получения векторного представления пути. Вектора путей в дальнейшем агрегируются с помощью механизма внимания для получения векторного представления кода в целом. Подход code2vec успешно применялся для поиска ошибок в коде [8], предсказания имён методов [111, 33] и поиска кода по текстовому описанию [2]. Одной из проблем данного подхода является большое разнообразие возможных путей в графе. Даже с учётом ограничений на максимальную длину количество потенциальных вариантов путей исчисляется миллиардами. Разумеется, в словарь попадёт лишь малая часть самых частых представителей. Существует вариация описанной архитектуры под названием code2seq [110], использующая рекуррентные нейронные сети для получения векторного представления путей из последовательности типов вершин, однако она не получила такого широкого распространения.

Извлечь больше структурной информации из кода позволяют методы, использующие графовое представление данных. Graph Neural Network [47] — это архитектура моделей глубокого обучения, которая предназначена для анализа графовых структур. Идея данного подхода состоит в итеративном обмене информацией между соседними вершинами графа. Более подробно процесс обучения выглядит следующим образом: сначала с каждой вершиной в графе ассоциируется некоторый числовой вектор её текущего состояния, а затем на каждой итерации обмена сообщениями для каждого исходящего ребра по вектору состояния генерируется вектор, называемый *сообщением*, затем агрегируются сообщения, полученные по входящим рёбрам, и обновляется вектор состояния вершины. Генерируемые сообщения зависят от метки вершины, её текущего состояния и типа связующего ребра. Gated Graph Neural Network (GGNN) – это особая модификация GNN, которая использует архитектуру Gated Recurrent Unit [63] для обновления векторов состояний. Архитектуры GNN применяются для предсказания трафика [64], восстановления повреждённых данных от сенсоров [51], диагностирования рака [40] и предсказания свойств молекул [32, 113]. Своё применение GNN нашли и в области анализа кода. В недавних работах [107, 1, 41] GGNN показала хорошие результаты в ряде работ, связанных с анализом семантики кода. Для получения графового представления кода обычно [107] берут AST и дополняют его рёбрами, отображающими какие-то логические связи,

а также потоки управления и передачи информации.

Архитектуры на основе трансформеров, отлично зарекомендовавшие себя в области обработки естественного языка, также активно применяются для анализа исходного кода. Примерами таких моделей могут служить CodeBERT [23] и CuBERT [84]. Они также используют идею предобучения на большом датасете и дальнейшее дообучение на небольших выборках для решения прикладных задач. Mastropaolo et al. [99] пошли ещё дальше и адаптировали для работы с кодом подход T5 из статьи [34]. Ключевой идеей данного подхода является формулирование всех задач в формат генерации текста по тексту и предобучении модели на множестве различных прикладных задач. Такой подход упрощает и унифицирует процесс дообучения модели для новых практических задач.

1.5 Представление изменений кода

Отдельной существенной проблемой применения машинного обучения в области программной инженерии является представление изменений программного кода. Один из возможных вариантов — анализ построчной разницы (добавленные и удалённые строки). В работе Kim et al. [59] определяли наличие ошибок в коммитах, используя мешок слов для добавленных строк, мешок слов для удалённых строк, а также метаданные коммита и различные метрики кода. Такой подход имеет ряд недостатков, например, чувствительность к изменению форматирования.

Альтернативой могут служить подходы, выделяющие структурные изменения AST. Список модификаицй, преобразующих одно дерево в другое, называют сценарием редактирования. Существующие исследования [37, 19, 18] в области поиска сценариев редактирования AST в основном имеют схожую структуру и состоят из двух фаз: сначала происходит сопоставление похожих вершин деревьев, а потом на основе получившегося сопоставления генерируется список изменений. Поиск сценария редактирования минимальной длины — NP-трудная задача [12], но для второй фазы существует оптимальный алгоритм [18], работающий за $O(n^2)$. Таким образом, ключевое отличие подходов кроется в первой фазе. Например, библиотека GumTree [37] срав-

нивает два дерева (AST кода до изменений и AST кода после) и генерирует список произошедших изменений. Всего выделяется четыре типа изменений: добавление вершины, перемещение вершины, удаление вершины и изменение метки вершины.

В библиотеке GumTree [37] сопоставление вершин двух деревьев происходит в два этапа. На первом шаге с помощью жадного поиска сверху вниз находятся наибольшие по высоте изоморфные поддеревья, называемые якорными сопоставлениями. На втором шаге сопоставляются вершины, имеющие в поддеревьях большое количество якорных сопоставлений. Получающиеся сценарии редактирования всегда корректны, то есть действительно преобразуют заданное AST в целевое, но могут быть неоптимальными по количеству действий. Исследователи провели сравнение и показали, что GumTree в среднем генерирует более короткие сценарии редактирования, чем аналогичная библиотека ChangeDistiller [19].

Ещё одним способом представлять изменения в коде является построение графов изменений. Например, этот способ используется в популярном инструменте CPatMiner [46], созданном для поиска частых шаблонов изменений кода. Сначала для кода до и кода после строится графовое представление, основанное на AST и дополненное рёбрами, отображающими потоки управления и передачу данных; затем вершины двух AST сопоставляются с помощью библиотеки GumTree, и два графа объединяются в один с помощью рёбер, соединяющих сопоставленные вершины. Получившийся граф содержит очень много информации, не относящейся непосредственно к произошедшим изменениям, поэтому из графа удаляются вершины, которые остались без изменений и при этом не связаны с изменившимися вершинами прямыми рёбрами. Авторы статьи выложили в открытый доступ реализацию для работы с кодом на языке Java. Также существует реализация, поддерживающая работу с Python[3].

[3]CodeChangeMiner: `https://github.com/JetBrains-Research/code-change-miner`

1.6 Объединение информации

В ряде задач при применении машинного обучения мы сталкиваемся с разнородностью данных. Например, при анализе коммитов имеются данные об изменении кода проекта и оставленное автором их текстовое описание. В работе Levin et al. [66], посвящённой классификации коммитов по цели изменений (добавление функциональности, исправление ошибки, улучшение качества кода), для объединения информации просто конкатенируются вектора мешков слов для описаний коммитов и для изменённого кода. Далее полученные вектора использовались для обучения классических моделей машинного обучения (Random Forest [53], Decidion Tree [68] и Grddient Boosting [38]). В данном исследовании применение комбинированной модели позволило увеличить точность классификации коммитов с 60% у лучшей модели, использовавшей только текст, до 75% у комбинированной модели.

В работе Hoang et al. [27] исследовался вопрос определения наличия ошибки в коммите. Для векторизации изменений кода и текстового описания применялись свёточные нейронные сети (Convolutional Neural Network, или CNN [60]). Сравнение полученной модели с предыдущими разработками проводилось на выборках коммитов из двух проектов: Qt[4] и openstack[5]. В качестве целевой метрики использовалась ROC-AUC [55]. В данной работе также было продемонстрировано превосходство комбинированных моделей: использование всей информации позволило добиться значения метрик $0,768$ на QT и $0,751$ на openstack против $0,641$ и $0,689$ при использовании только текста, и $0,738$ и $0,729$ при использовании только изменений кода.

1.7 Предсказание тегов

Рассмотрим подробнее одну из задач, в которой также есть потенциал для применения комбинированных методов. Спортивное программирование — это интеллектуальное соревнование, в котором участники должны писать программы, решающие заданные алгоритмические задачи и укладывающиеся в ограничения по используемому времени и оперативной памяти. Уча-

[4]Qt: `https://www.qt.io/`

[5]openstack: `https://www.openstack.org/`

стие в подобных соревнованиях мотивирует учащихся к изучению программирования, алгоритмов и структур данных, а также помогает проходить технические интервью в IT компаниях [15].

Онлайн-платформы для проведения соревнований по спортивному программированию, такие как Codeforces[6] и CodeChef[7], позволяют использовать архивы задач для тренировки или в качестве практических заданий к теоретическим курсам по алгоритмам и структурам данных. К сожалению, задачи в подобных архивах не систематизированны, что усложняет поиск заданий по конкретной теме. Для решения этой проблемы некоторые платформы используют системы тегов. Тег (tag) — это специальная метка, обозначающая принадлежность задачи некоторой группе. Теги могут описывать как общую тематику задач (*математика*, *геометрия*, *графы*, *строки*), так и конкретные способы их решения (*динамическое программирования*, *полный перебор*, *бинарный поиск*). Каждая задача может иметь произвольное число тегов. Ручная расстановка тегов отнимает много времени и сил, а разметка получается субъективной и неоднородной. Автоматическая система тегирования может решить эти проблемы. Существующие попытки [16, 5, 21, 97, 71, 22] создания такой системы можно разделить на два принципиально разных типа.

Часть авторов [16, 97, 71, 22] использовала методы обработки естественного языка для анализа текста условий задач. Такой подход показывает хорошие результаты при определении общей тематики задачи (*math*, *graphs*), но для определения конкретных способов решения (*brute force*, *binary search*) по тексту условия в некотором смысле необходимо решить задачу, а это вызывает сложности даже у людей.

Извлечь информацию о конкретных алгоритмах решения задачи можно из присланных участниками программ. Это приводит нас к анализу исходного кода. Подобный подход был реализован в нескольких предыдущих работах [5, 21], где авторы интерпретировали определение тегов как задачу распознавания алгоритмов, т. е. пытались каждому фрагменту кода сопоставить алгоритм, который он реализует. Однако и такой подход не лишён

[6]Онлайн-платформа Codeforces: `https://codeforces.com/`

[7]Онлайн-платформа CodeChef: `https://www.codechef.com/`

недостатков. Например, в указанных работах разметка для задач экстраполируется на их решения, что может порождать дополнительный шум в том случае, если задача допускает вариативность решения.

Исследования применимости комбинированных моделей в данной задаче не проводились.

1.7.1 Предсказание тегов по условию задач

Условия задач написаны на естественном языке, а потому применение хорошо зарекомендовавших себя методов из NLP выглядит разумной идеей.

Bora et al. [16] предлагали использовать архитектуру LSTM. Для кодирования токенов авторы применяли как предобученные вектора из модели word2vec [29], так и унитарное кодирование. Выборка условий задач была собрана с платформы Codeforces — популярного сервиса для проведения соревнований по спортивному программированию. Для каждой задачи учитывался только первый тег. В экспериментах авторов только Random Forest [53] (один из классических алгоритмов классификации, использовавшийся в качестве референсного решения) превзошёл наивный классификатор, всегда выбирающий самый популярный класс.

Athavale et al. [97] использовали свёрточные нейронные сети для анализа текста. Они ставили задачу предсказания тегов и как многоклассовую классификацию (каждому элементу сопоставляется ровно один из множества классов), и как классификацию с несколькими метками (каждому элементу сопоставляется несколько классов). Как и в предыдущей работе, выборка задач собиралась с платформы Codeforces. В двух сериях экспериментов авторы пробовали предсказывать 10 и 20 самых частых тегов. Лучшие результаты показал ансамбль из CNN. Далее авторы попросили решить схожую задачу нескольких людей, имеющих опыт в спортивном программировании. Хотя эксперты предсказывали теги точнее лучшей из рассматривавшихся моделей, их результаты всё ещё сложно назвать впечатляющими: в среднем метрика F1-macro получилась на уровне $0,43$ на задаче определения релевантности 20 самых частых тегов.

Похожая задача исследовалась в работе Iancu et al. [71]. Авторы собрали

выборку условий задач с платформ Codeforces и TopCoder[8], и сгруппировали их теги в 9 классов. Было рассмотрено несколько моделей: Doc2Vec [62] и LSTM, где токены векторизовались с помощью word2vec или унитрного кодирования. Максимальное значение метрики F1 было достигнуто моделью LSTM в комбинации с унитарным кодированием токенов, но LSTM с векторами из word2vec оказалась лучше по метрике Weighted Hamming Score (взвешенная версия расстояния Хэмминга [50]).

Немного иной подход рассматривался в исследовании Intisar et al. [22]. Авторы использовали алгоритмы тематического моделирования (LDA [14] и NMF [65]) для векторизации условий, а потом применяли к полученым векторам классические алгоритмы классификации: kNN [61], Random Forest (RF), Multinomial Naive Bayes (MNB) [102] и Multilayer Perceptron [52]. В качестве альтернативного представления данных использовался простой TF-IDF [86]. Хотя использование неявных признаков заметно повлияло на работу отдельных алгоритмов (положительно в случае kNN и MNB, отрицательно в случае RF), максимальная точность классификации почти не улучшилась — $0,86$ против $0,88$.

Однако, такие подходы слабо подходят для определения конкретных алгоритмов решения (*динамическое программмирование*, *dfs* и т.д.) Другим значительным недостатком анализа условий является относительно маленькие размеры выборок. Это может приводить к перебучению — явлению, при котором модель вместо вывода обобщающих правил явно запоминает ответы для примеров из обучающей выборки.

1.7.2 Предсказание тегов по коду решений

Кроме условия задачи на естественном языке для анализа доступен и исходный код присланных решений. Shalaby et al. [5] использовали метрики исходного кода для векторизации решений. Они взяли выборку решений задач с Codeforces и посчитали для них 30 простых метрик, таких как количество переменных определённого типа (например, *int*, *string*), количество строк кода, число циклов, число вложенных циклов и так далее. Авторы про-

[8]Онлайн-платформа TopCoder: `https://www.topcoder.com/`

верили, как хорошо такой способ представления исходного кода позволяет различать категории задач и определять конкретные алгоритмы решения. Их эксперименты показали, что классические алгоритмы классификации и метрики исходного кода могут успешно применяться для классификации решений.

Sudha et al. [21] использовали посимвольные CNN, т. е. интерпретировали каждое решение как последовательность символов. Авторы использовали выборку задач с Codeforces и обучили модель для классификации решений на 4 класса. Они также предложили агрегировать информацию от всех присланных решений для одной задачи путём выбора самого частого класса. В их экспериментах предложенная модель смогла правильно определить категорию задачи в 61% случаев.

Одним из важных недостатков подобных подходов является необходимость экстраполировать теги задач на каждое из их решений, так как обычно изначальная разметка имеется именно для задач. Но многие задачи допускают вариативность решения, из чего следует, что для части решений некоторые теги могут быть нерелевантными.

1.8 Поиск bug-fix коммитов

Рассмотрим ещё одну задачу, где есть возможность использовать информацию из разных источников. Выше уже упоминалась задача классификации коммитов по цели изменений. Частным случаем этой задачи является определение коммитов, исправляющих какие-то ошибки в коде (далее — bug-fix коммиты).

Поиск ошибок в коде — весьма времязатратный и трудоёмкий процесс. Применение методов машинного обучения может существенно ускорить и удешевить разработку ПО. Для этих целей существует несколько классических постановок задачи машинного обучения. Поиск ошибкоопасных мест подразумевает определение единицы кода, которая наиболее вероятно содержит ошибку. В качестве единицы кода обычно выступает метод или класс. Другой постановкой является определение ошибкоопасных коммитов. В этой постановке предлагается по коммиту определить, вносит ли он новую ошиб-

ку в проект. Для обучения подобных моделей сначала необходимо собрать выборку подобных коммитов. Сама по себе это задача нетривиальная, однако существуют алгоритмы (например, SZZ [91]), позволяющие по bug-fix коммиту найти коммит, ответственный за внесение ошибки в кодовую базу. Собрать же выборку коммитов, исправляющих ошибки, уже несколько проще. Часто для этого просто фильтруют коммиты по ключевым словам в описании коммита или используют системы баг-трекинга крупных проектов. Однако, существуют свидетельства [114], что такие выборки получаются нерепрезентативными. Использование методов машинного обучения для сбора выборки может если не добиться репрезентативности, то по крайней мере увеличить размеры выборки (за счёт автоматизации по сравнению с использованием баг-трекеров) и разнообразность ошибок (по сравнению с фильтрацией по ключевым словам в описании).

Другим прикладным применением данной задачи является автоматизация дублирования исправлений ошибок в стабильную версию. Обычно у каждого крупного проекта есть одна ветка, в которой лежит версия, предназначенная для конечных пользователей. Новая функциональность попадает в неё после долгого тестирования и других проверок качества. Однако, если в какой-то другой ветке была найдена и исправлена какая-то критическая ошибка, то такое исправление следует непромедлительно продублировать и в стабильную версию.

Большинство исследований при решении этой задачи используют только текстовое описание коммитов, однако существуют исследования, применяющие комбинированные подходы.

1.8.1 Анализ описаний коммитов

Самый простой способ выявить bug-fix коммиты – проверить наличие ключевых слов в описании коммита. Обычно используется небольшой список из ключевых слов (не более 10), включая *bug*, *fix*, *issue*, *error*, *incorrect* и т. д. Такой подход используется для сбора таких наборов данных повсеместно [70, 77, 76, 57, 75, 112]. Плюсом таких подходов является простота реализации, скорость работы, а также хорошая точность работы (т. е. почти все

коммиты, содержащие ключевые слова в описании, действительно исправляют ошибки). Однако полнота и репрезентативность полученной выборки вызывает вопросы [114].

1.8.2 Комбинированные подходы

Hoang et al. [78] для анализа описаний коммитов и изменений кода в проекте PatchNet применяют CNN [60]. CNN независимо применяется для удалённого и добавленного кода, а также для текстовых описаний. Полученные вектора конкатенируются и пропускаются через полносвязный слой для получения итогового предсказания. В работе использовалась выборка коммитов из проекта Linux Kernel[9]. Авторы сравнивали свой подход с классическими алгоритмами, обученными на мешке слов из описаний коммитов, и показали превосходство комбинированного подхода – $0,860$ по метрике ROC-AUC против $0,809$ у ближайшего конкурента. В дальнейшем авторы продолжили работу над проектом и улучшили работу с изменениями кода.

В новом исследовании [17] для векторизации изменений использовался подход CC2Vec. Этот подход использует несколько слоёв механизма внимания. Сначала строчки обрабатываются с помощью GRU для получения векторного представления токенов. Далее вектора слов агрегируются с помощью первого слоя механизма внимания в векторное представление строки. После этого аналогичная схема с применением GRU и механизма внимания используется для агрегации информации из нескольких строк в векторное представление блока подряд идущих изменений. На последнем шаге таким же образом информация из разных блоков объединяется в итоговое представление изменений кода. CC2Vec тестировалась на нескольких задачах, включая определение bug-fix коммитов. Для этого полученное от CC2Vec итоговое векторное представление изменений объединялось с полученным от PatchNet векторным представлением текстового описания коммита. В результате удалось улучшить результат с $0,860$ до $0,916$.

Недостатком данных подходов можно считать работу с кодом на уровне токенов (без использования структуры кода) и апробация на выборке из все-

[9]Linux Kernel: `https://github.com/torvalds/linux`

го одного проекта, что ставит под сомнение воспроизводимость результатов на других проектах.

1.9 Выводы

Применение методов машинного обучения для решений практических задач программной инженерии — активно развивающаяся область. Во многих задачах есть данные в разной форме — код, изменения кода или текст. Практика показывает, что объединение информации из разных источников позволяет улучшить результаты работы моделей. Однако, не для всех задач пока используются комбинированные подходы. Но даже там, где комбинированные подходы применяются, работа с кодом зачастую устроена как работа с текстом — на уровне токенов. Опыт из области анализа кода показывает, что использование специфичных для кода моделей позволяет получить лучшие результаты.

2 Предлагаемый подход

Обзор существующих решений показал, что не для всех задач существуют комбинированные подходы. Но даже если комбинированные решения предлагаются, зачастую они не используют специфичные для кода модели. В данной работе предлагается архитектура, позволяющая быстро и просто получать комбинированные решения для задач программной инженерии.

2.1 Общая архитектура

Важным требованием к архитектуре решения является возможность объединять информацию из нескольких разных по форме источников. Для этого предлагается отдельно обучать специфичные для разных данных модели, а потом объединять их в ансамбль. Такой подход обладает несколькими важными качествами. Во-первых, он чрезвычайно гибкий. Такая архитектура позволяет, например, комбинировать методы классического машинного обучения (скажем, Random Forest) с нейросетевыми подходами. Это позволит при необходимости включать в архитектуру решения передовые для конкретной задачи подходы. Во-вторых, за счёт независимости обучения такая архитектура меньше страдает от переобучения. Общая схема предлагаемого подхода показана на Рисунке 2.1.

Далее описывается архитектура предлагаемых моделей для трёх наиболее часто встречающихся типов данных: кода, изменений кода и текста на естественном языке.

2.2 Анализ текста

В недавних исследованиях [100, 67, 81] подход Bidirectional Encoder Representations from Transformers (BERT) [9] – передовой метод NLP – показал лучшие результаты для нескольких популярных задач. Одной из важных особенностей данного подхода является разделение процесса применения на предобучение на обширной выборке данных на нескольких стандартных задачах и дообучение на отдельной выборке для целевой задачи. Благодаря знаниям о структуре естественного языка, полученным во время предо-

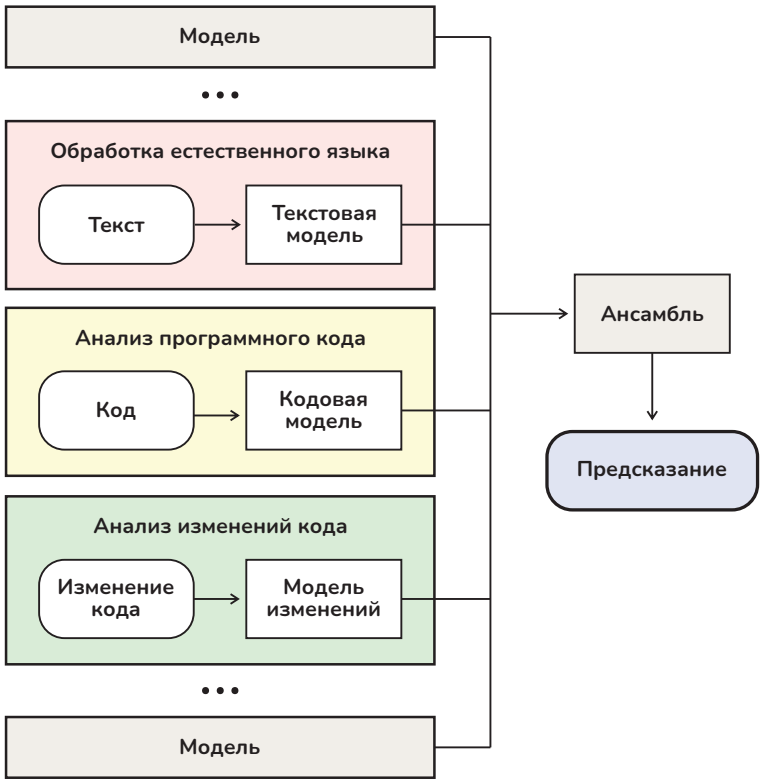

Рис. 2.1: Общая схема предлагаемого подхода.

бучения, BERT показывает хорошие результаты работы даже на небольших выборках. Именно из-за этой особенности BERT кажется хорошим выбором для нашей задачи. Для токенизации текста должна применяться техника WordPieces [44] с тем же словарём фрагментов, который был получен при обучении модели.

2.3 Анализ кода

Для анализа кода была выбрана модель GGNN, так как она показала лучшие результаты в ряде недавних работ [107, 1, 41]. Графовое представление исходного кода базируется на AST фрагмента, дополненном специальными рёбрами. Для построения AST использовалась библиотека для генерации парсеров Tree-sitter[10].

[10]Библиотека Tree-sitter: `https://tree-sitter.github.io/`

Набор дополнительных рёбер использовался такой же, как и в статье Allamanis et al. [1]. Для получения векторного представления всего графа, а не отдельных его вершин, использовался механизм внимания. Для регуляризации к полученному вектору применяется метод исключения [31]. Далее векторное представление кода подаётся на вход полносвязному слою, размер выходного слоя которого зависит от конкретной задачи.

2.4 Анализ изменений кода

Изменения кода предлагается представлять в виде графа изменений, так как именно такое представление сохраняет больше всего информации о структуре изменений. Графовое представление изменений в данной работе получалось с помощью библиотеки CodeChangeMiner[11]. Граф содержит рёбра восьми типов: рёбра *control* соединяют вершину с темы вершинами, чьё выполнение зависит от неё; рёбра *def* соединяет переименую и её новое значение в конструкциях присваивания; рёбра *ref* добавляют связи между разными вершинами, соответствующими одной переменной; *para* соединяют вызов метода с передаваемыми параметрами; *cond* соединяют условные операторы с их условиями; *qual* соединяет части составных имён,; *map* соединяет соответствующие друг другу вершины из графов для кода до и после изменений. Более подробное описание рёбер графов изменений можно найти в работе Nguyen et al. [46]. Для векторизации графов изменений использовалась архитектура GGNN.

Так же, как и в случае анализа кода, итоговое векторное представление получалось из векторов вершин с помощью механизма внимания, а затем пропускалсь через метод исключения [31] и подавалось на вход полносвязному слою.

[11] CodeChangeMiner: `https://github.com/JetBrains-Research/code-change-miner`

3 Предсказание тегов

Эта глава посвящена применению предложенного подхода для решения задачи определения тегов задач спортивного программирования. Задача формулируется как классификация с пересекающимися классами. У задачи из выборки есть условие на естественном языке и список присланных пользователями решений на одном из языков программирования. Каждой задаче присвоено некоторое множество релевантных тегов. Требуется для каждой задачи и каждого тега определить, является ли выбранный тег релевантным для конкретной задачи.

3.1 Архитектура

В целом архитектура решения совпадает с предложенной в Главе 2, однако в данном случае у нас не один фрагмент кода, а несколько. Решено было обучать использующую код решений модель в несколько этапов. Сначала модель GGNN тренируется предсказывать теги для отдельных решений, а затем обученная модель используется для генерации векторных представлений всех решений в выборке. Далее на полученных данных обучается ещё одна модель, агрегирующая вектора решений отдельной задачи с помощью механизма внимания и использующая получившееся векторное представление для предсказания тегов задачи в целом. Для дополнительной регуляризации во время обучения на каждой итерации использовалось лишь 50 случайно выбранных решений для каждой задачи, а к их векторным представлениям применялся метод исключения [31]. Итоговая схема архитектуры показана на Рисунке 3.1.

3.2 Набор данных

В этом исследовании использовалась выборка задач с Codeforces — платформы проведения соревнований по спортивному программированию. Каждая задача имеет некоторый (возможно, пустой) набор релевантных тегов, условие на английском языке в формате LaTex и список присланных и успешно прошедших все тесты решений на языке *C++*. Задачи, не имевшие условий

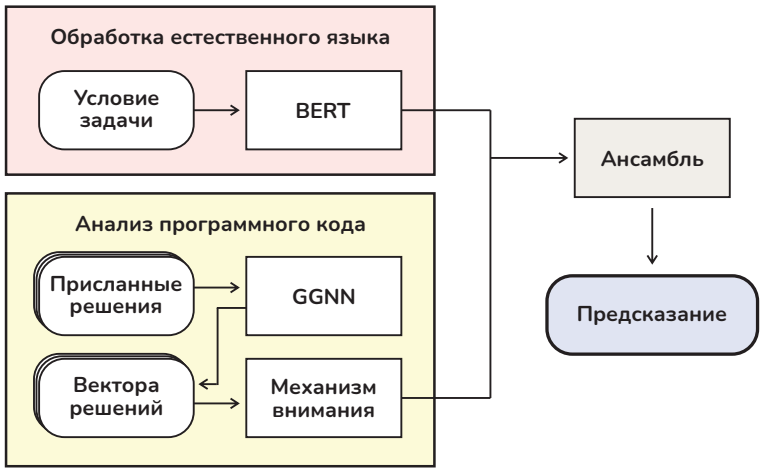

Рис. 3.1: Архитектура предагаемого алгоритма для предсказывания тегов.

на английском языке, и решения, написанные на других языках, игнорировались. В итоге выборка состояла из 1 065 контестов, 5 981 задач и 14 257 576 решений. Решения задач были распределены неравномерно: популярные задачи могли иметь тысячи решений, а самые сложные — лишь считанные единицы. Всего в выборке было представлено 35 различных тегов, но 5 из них встречались слишком редко, поэтому мы убрали их из рассмотрения.

3.2.1 Удаление дубликатов

Решения в выборке оказались неуникальными. Мы использовали алгоритм SHA-256 [43] для подсчёта хешей решений, а потом из каждой группы решений с одинаковым хешем оставили только самое раннее по моменту посылки. Однако оказалось, что дублируются не только решения, но и условия задач. Почти идентичные условия часто возникают в ситуации, когда имеется несколько версий одной задачи с разным уровнем сложности — отличаться может ограничение на размер входных данных, максимальное время исполнения или потребляемая оперативная память и т.д. Например, упрощённая версия задачи может быть решена полным перебором, а продвинутая версия потребует реализации сложных алгоритмов. Можно было удалить или объединить задачи с похожими условиями, но чтобы не вносить лишний шум в теги задач и не терять данные, мы решили ограничется проверкой, что

никакая задача из тестовой выборки не была использована во время обучения моделей или оптимизации гиперпараметров. Мы считали условия задач дубликатами, если их сходство по мере Жаккара [45] превышало $0,9$. Всего на этой стадии было удалено 18 задач и $2,5\%$ решений.

3.2.2 Разделение на выборки

При решении задач машинного обучения выборка обычно разбивается на 3 части: обучающую, валидационную и тестовую. Первая используется для обучения модели, вторая для подбора гиперпараметров (размеры скрытых слоёв, количество эпох обучения и т.п.), а третья для финальной независимой оценки качества.

В подобных нашему исследованиях, где выборка состоит из данных, собранных за значительный промежуток времени, крайне важно правильно разделить выборку. Первый контест в нашем датасете был опубликован в 2010 году, а последний в 2019. Множество изменений произошло за эти годы (новые стандарты добавляли в языки новые конструкции, росла численность аудитории платформы и т.д.), поэтому выборка получилась неоднородной, что может влиять на оценку качества моделей. Это можно наглядно наблюдать в работе Shalaby et al. [5]. Авторы сравнили качество моделей при случайном и при хронологическом разделении выборки и обнаружили существенной падение наблюдаемых метрик качества во втором случае.

Для получения надёжной оценки качества мы разделили датасет хронологически. Для гарантии корректности разбиения мы удалили все решения из обучающей выборки, которые были присланы после самого первого решения валидационной выборки. Аналогичная процедура была проделана для валидационной и тестовой выборок. Наконец, для уменьшения дисбаланса данных мы ограничили максимальное количество решений для одной задачи. Если количество решений для какой-то задачи превышало 1 000, мы случайным образом выбирали из них 1 000 представителей, а остальные решения удаляли. В Таблице 3.1 показана информация о размере выборок после выполнения всех описанных процедур.

Таблица 3.1: Информация об итоговом размере выборок

Выборка	**Контесты**	**Задачи**	**Решения**
Обучающая	716	3 837	2 493 745
Валидационная	169	1 020	664 578
Тестовая	159	1 046	734 612
Всего	1 044	5 903	3 892 935

3.2.3 Предобработка

Анализ исходного кода на *C++* чрезвычайно сложен из-за богатой грамматики языка. Для упрощения задачи все директивы *#include* были удалены для избежания подстановки кода из стандартной библиотеки языка, а потом был запущен препроцессор для подстановки всех пользовательских шаблонов и удаления комментариев. Условия задачи также требовали обработки. Изначально условия хранились в формате *LaTeX*, который весьма далёк от естественного языка. Была использована программа Pandoc [30] для конвертации документов в обычный текст, а затем применён предобученный токенизатор WordPieces [44] для получения текстового представления, подходящего для применения модели BERT [9].

3.3 Апробация

3.3.1 Целевая метрика

Метрики Precision и Recall [83] часто используются для оценки качества бинарных классификаторов. Они оценивают качество их работы по двум аспектам: точность и полнота соответственно. Но современные модели обычно сообщают не только предсказанный класс, но и степень уверенности в ответе. Это даёт нам дополнительную степень свободы: увеличивая пороговое значение классификации, мы уменьшаем Recall и обычно увеличиваем Precision. Выбор порогового значения всегда приводит к проблеме балансирования между этими метриками и конкретный его выбор зависит от условия применения модели. В силу этих причин, эти метрики обычно не используются для сравнения качества работы моделей.

Метрика F1 — гармоническое среднее Precision и Recall — лучше подходит для сравнения, однако всё ещё зависит от выбора порогового значения. Чтобы избавится от необходимости выбора, перейдём от точечных оценок к анализу кривой Precision-Recall, полученной как график зависимости Precision от Recall при изменений порога. Если посчитать площадь под этой кривой, то можно получить метрику Area Under Precision-Recall Curve (PR-AUC) [26] — популярную метрику качества классификации. Кроме того, PR-AUC устойчива к сильно дисбалансным датасетам, что является крайне важным свойствам в нашем исследовании, так как процент задач, имеющих заданный тег, обычно невелик.

Мы выбрали PR-AUC в качестве целевой метрики, но для полноты картины предоставляем также значения метрик F1, Precision and Recall (пороговое значение подбиралось для максимизации метрики F1 на валидационной выборке).

3.3.2 Текстовые модели

В первой серии экспериментов изучалась полезность условий задач для предсказания тегов. Для этой цели на условиях всех задач из обучающей выборки было обучено несколько текстовых моделей предсказывать их теги. Вот полный список рассматривавшихся подходов:

1. Bora et al. [16]: LSTM с унарным кодированием, LSTM с векторами из word2vec и Logistic Regression (LR) поверх мешка слов. Следуя оригинальной статье, никаких методов предобработки текста не применялось (стемминг, лемматизация, удаление стоп-слов и т.п.). Модели LSTM предобучались на задаче определения сложности задачи. На платформе Codeforces нет строгого деления задач по сложности, поэтому в качестве меток мы использовали разделение контестов на дивизионы.

2. Iancu et al. [71]: LSTM с унарным кодированием, LSTM с векторами из word2vec и деревья решений поверх TF-IDF. Для удаления стоп-слов использовалась библиотека Natural Language Toolkit[12].

[12]NLTK: `https://www.nltk.org/`

3. Athavale et al. [97]: CNN с обучаемыми векторами слов (TWE + CNN), CNN с предобученными векторами из GloVe [80] (GloVe + CNN) и ансамбль из CNN. Использовалась открытая реализация подходов из оригинальной статьи.

4. BERT [9]. В данной работе использовалась преобученная модель BERT, дообученная для нашей задачи.

Таблица 3.2: Результаты предсказания тегов задач по тексту условий

Подход	**PR-AUC**	**F1**	**P**	**R**
td-idf + DT [71]	0.250	0.207	0.219	0.203
one-hot encoding + LSTM [71]	0.203	0.259	0.212	0.409
word2vec + LSTM [71]	0.210	0.280	0.252	0.400
BoW + LR [16]	0.227	0.252	0.252	0.306
one-hot encoding + LSTM [16]	0.177	0.236	0.188	0.385
word2vec + LSTM [16]	0.185	0.255	0.252	0.362
TWE + CNN [97]	0.268	0.303	0.295	0.372
GloVe + CNN [97]	0.276	0.292	0.263	0.388
CNN Ensemble [97]	**0.278**	0.289	0.296	0.338
BERT (данная работа)	0.273	**0.308**	0.268	0.417

Результаты этих экспериментов показаны в Таблице 3.2. Стоит отметить, что классические алгоритмы классификации (например, решающие деревья) показывают относительно неплохие результаты, что согласуется с результатами Bora et al. [16]. Возможная причина кроется в малом количестве параметров, что позволяет им меньше переобучаться на маленьких выборках.

Все рекуррентные сети обучились немного хуже. Одной из частых проблем рекуррентных сетей в задачах классификации является исчезание градиента — негативного эффекта обратного распространения ошибки в глубоких сетях. Свёрточные сети такой проблемы обычно не имеют. Ансамбль из CNN ещё и более устойчив к шумам, поэтому неудивительно, что он показал лучший результат, составивший $0,278$ PR-AUC. BERT незначительно проиграл ему по метрике PR-AUC ($0,278$ против $0,273$), но оказался немного

лучше по метрике F1 ($0,289$ против $0,308$). Главным преимуществом BERT является предобучение на большом корпусе текста.

Этот эксперимент демонстрирует, что предложенная модель по качеству сопоставима с лучшими из ранее предложенных текстовых моделей.

3.3.3 Кодовые модели

Во второй серии экспериментов сравнивались подходы, использующие решения задач для определения тегов. Эти эксперименты проводятся в два этапа. На первом шаге модели сравниваются на задаче предсказания тегов для отдельных решений. К сожалению, у нас нет отдельной разметки для решений, поэтому в качестве меток используются теги соответствующих задач. На втором шаге модели анализируют все решения задачи и предсказывают её теги с учётом всей доступной информации. Вот полный список рассматривавшихся подходов:

1. Sudha et al. [21]: посимвольный CNN.
2. Shalaby et al. [5]: AdaBoost, Random Forest (RF), and Support Vector Machine (SVM) из библиотеки sklearn[13] с метриками кода в качестве признаков.
3. GGNN для анализа и векторизации отдельных решений, GGNN с механизмом внимания для итогового предсказания тегов задач.

Таблица 3.3: Результаты предсказания тегов решений

Подход	**PR-AUC**	**F1**	**P**	**R**
character-wise CNN [21]	0.249	0.289	0.306	0.323
metrics+RF [5]	0.254	0.303	0.282	0.376
metrics+AdaBoost [5]	0.225	0.270	0.208	0.474
metrics+SVM [5]	0.225	0.272	0.245	0.495
GGNN (данная работа)	**0.393**	**0.430**	0.467	0.446

[13]sklearn: `https://scikit-learn.org/`

Результаты этапа экспериментов показаны в Таблице 3.3. Несмотря на значительную потерю информации при перобразовании исходного кода в набор метрик, использовавшие такое представление информации классические алгоритмы классификации показали достойные результаты. Как и в исследовании Bora et al. [16], лучший результат ($0,254$ PR-AUC) показала модель Random Forest. Посимвольный CNN использовал потенциально более полную информацию о коде, но это не помогло ему предвзойти Random Forest ($0,249$ против $0,254$). Слабым местом этой модели является полное игнориование априорной информации о структуре кода. Возможно, именно активное использование этой информации помогло модели GGNN значительно превзойти референсные решения, показав результат $0,393$ по метрике PR-AUC.

На следующем этапе сравнивалось качество предсказания тегов для задач по всем их решениям. Похожая идея использовалась в работе Sudha et al. [21], где модель обучалась на предсказании тегов для отдельных решений, а предсказание для задачи выполнялось путём определения самого частого результата классификации её решений. Мы сохранили такой подход обобщения работы референсных моделей на уровень задач, с той лишь разницей, что самый частый результат в нашей работе считается отдельно по каждому тегу, так как в отличие от оригинальной статьи, в данной работе решалась задача классификации с несколькими метками. Предлагается применять другой подход к агрегации информации о решениях: модель GGNN, обученная на предыдущем шаге, используется для генерации векторного представления решений, которые затем суммируются с помощью механизма внимания и используются для предсказания тегов на уровне задач.

Результаты второго этапа экспериментов показаны в Таблице 3.4. Как и следовало ожидать, качество предсказания тегов для задач сильно выше качества предсказания тегов для отдельных решений за счёт использования большего количества информации, что позволяет учесть несколько возможных алгоритмов решения и уменьшить влияние шума. Подход, основанный на GGNN, уверенно занимает первое место по качеству предсказания — $0,514$ по метрике PR-AUC против $0,388$ у ближайшего конкурента (Random Forest).

Таблица 3.4: Результаты предсказания тегов задач по коду решений

Подход	**PR-AUC**	**F1**	**P**	**R**
character-wise CNN [21]	0.372	0.360	0.377	0.414
metrics + RF [5]	0.388	0.389	0.386	0.467
metrics + AdaBoost [5]	0.325	0.356	0.341	0.452
metrics + SVM [5]	0.363	0.339	0.297	0.564
GGNN + Attention (данная работа)	**0.514**	**0.517**	0.487	0.594

3.3.4 Ансамбль и итоги

Выше мы рассмотрели две диаметрально противоположных точки зрения на задачу предсказания тегов. На первый взгляд может показаться, что анализ кода даёт больше информации, чем анализ текста. Но такая иллюзия возникает из-за усреднения результатов по тегам. На самом деле ситауция несколько сложнее. Для иллюстрации неоднородности качества предсказания мы выбрали несколько моделей разных типов и вычислили для каждой модели вектор, каждый элемент в котором является значением метрики качества предсказания отдельного тега, а затем посчитали коэффициент корреляции Пирсона [79] для этих векторов. Высокое значение коэффициента корреляции означает, что элементы векторов отклоняются от среднего значения синхронно: если одна модель особенно хорошо предсказывает какой-то тег, то и вторая модель должна успешно с ним справляться. Результаты показаны на Рисунке 3.2. Использовавшие код решений модели выделены жёлтым, использовавшие текст условий — красным. Можно видеть, что корреляция между моделями одного типа заметно выше, чем между моделями разных типов, что наглядно иллюстрирует глобальные отличия между подходыми: текстовые модели хороши для предсказания одних тегов, а кодовые — для предсказания других. Именно поэтому объединение разных подходов может увеличить точность классификации.

Как упоминалось в Главе 2, предлагается взять GGNN и BERT как представителей разных подходов и объединить их в ансамбль. Сравнение полученной модели с другими рассматриваемыми подходами приведено в Таблице 3.5. Предложенный подход превзошёл все остальные модели, включая

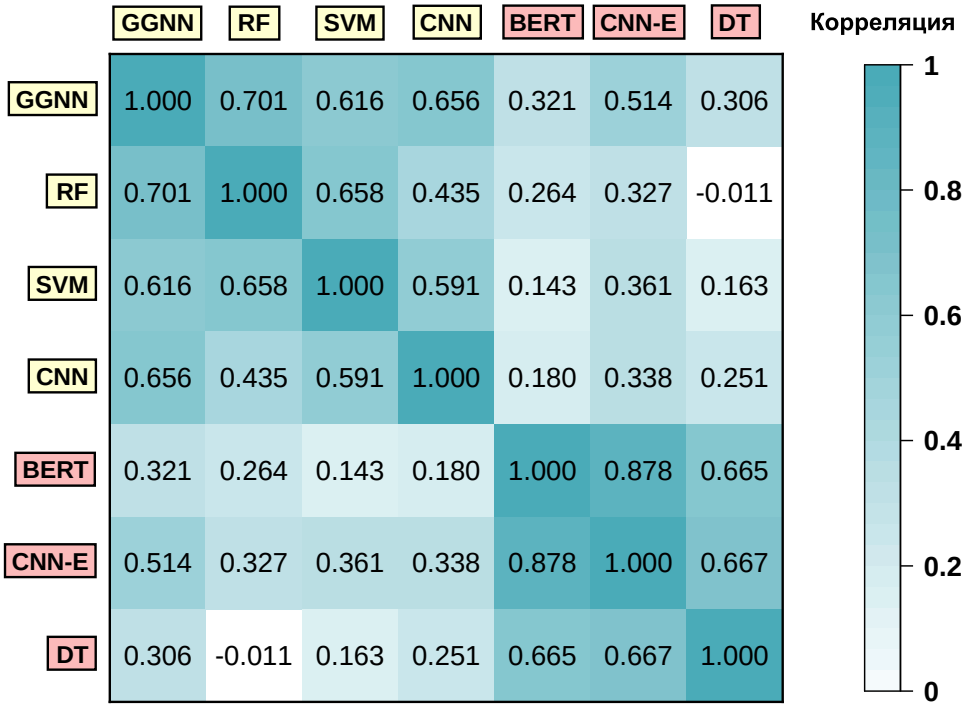

Рис. 3.2: Корреляция списков значений PR-AUC для разных моделей. Жёлтый цвет означает подходы, работающие с кодом, а красный — подходы, работающие с текстом.

GGNN и BERT по отдельности.

3.4 Выводы

В данной главе проверялась применимость предложенной архитектуры для решения проблемы определения тегов задач спортивного программирования. Для сравнения были реализовали разнообразные подходы из предыдущих исследований и сравнено качество их работы на обширном датасете задач с Codeforces. Анализ корреляции качества работы моделей показывает, что ориентированные на текст модели показывают лучшие результаты на других тегах, нежели модели, ориентированные на анализ кода. Это свидетельствует о том, что комбинирование этих моделей может улучшить общую точность предсказания. В частности, использовавшийся здесь комбинированный подход превосходит лучшую текстовую модель на $0,264$ и лучшую

Таблица 3.5: Сводная таблица результатов лучших моделей

Подход	**PR-AUC**	**F1**	**P**	**R**
CNN Ensemble [97]	0.278	0.268	0.271	0.327
BoW + DT [16]	0.257	0.236	0.233	0.249
metrics + RF [5]	0.388	0.389	0.386	0.467
Code+CNN [21]	0.367	0.392	0.429	0.419
BERT (данная работа)	0.273	0.308	0.268	0.417
GGNN + Attention (данная работа)	0.514	0.517	0.487	0.594
BERT + GGNN (данная работа)	**0.542**	**0.532**	0.489	0.618

кодовую модель на $0,154$ по метрике PR-AUC.

Хотя о полной автоматизации разметки задач спортивного программирования говорить пока не приходится в силу недостаточной точности предсказания, полученная модель может существенно упростить ручную разметку путём рекомендации тегов либо ранжирования их по релевантности, а также помочь в поиске ошибок уже расставленных тегов.

4 Выявление bug-fix коммитов

Эта глава посвящена выявлению bug-fix коммитов из истории проектов. Проблема формулируется как классическая задача бинарной классификации. Каждому коммиту соответствует изменение кода и его текстовое описание на естественном языке.

4.1 Архитектура

Для решения данной задачи используется архитектура, предложенная в Главе 2. Для объединения информации об изменениях кода и текстовом описании коммитов использовался ансамбль из двух моделей. Для анализа изменений кода использовалась модель GGNN с графами изменений, полученными с помощью CodeChangeMiner, а для анализа текстового описания – модель BERT. Общая структура решения показана на Рисунке 4.1.

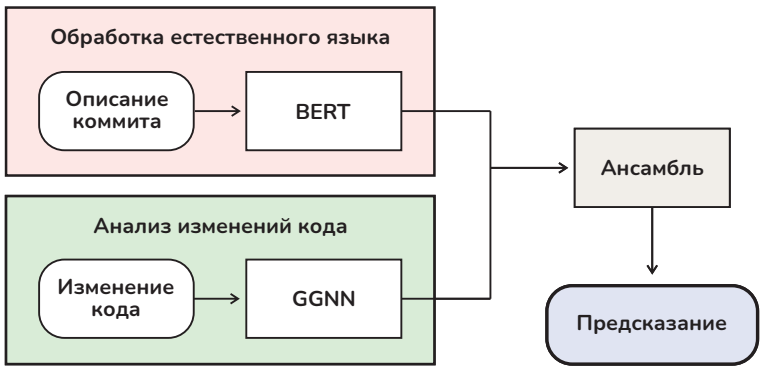

Рис. 4.1: Архитектура предагаемого алгоритма для предсказывания bug-fix коммитов.

4.2 Набор данных

4.2.1 Датасет Linux Kernel

Этот датасет был представлен в работе [78]. Он собран из репозитория Linux[14] и состоит из коммитов, исправляющих ошибки в стабильных версиях проекта. Всего за период с июля 2011 года по июль 2017 было совершено 424 380

[14]Linux: `https://github.com/torvalds/linux`

коммитов. Если исключить коммиты, соответствующие слияниям веток, а также только добавляющие либо только удаляющие код, и затем оставить только те из них, которые изменяют файлы с расширением *.c* или *.h*, то останется 346 570 коммитов. Далее из датасета удалялись слишком объёмные коммиты (те, которые изменяли более 100 строк за раз). Из итогового набора данных в 267 251 коммитов было выделено 42 408 bug-fix коммитов и 39 995 случайных. Таким образом, полученная выборка является сбалансированной по классам. Текстовые описания коммитов разделили на токены, удалили стоп-слова и применили стемминг. Датасет выложен авторами в открытый доступ[15]. Недостатком данного датасета является его нерепрезентативность т.к. все коммиты получены с одного проекта.

Таблица 4.1: Сравнение моделей на датасете с Linux Kernel

Подход	**ROC-AUC**
PatchNet [78]	0.860
CC2Vec [17]	0.916
RandomForest (эта работа)	0.959

4.2.2 Сбор данных с GitHub

Можно выделить два основных пути сбора датасетов bug-fix коммитов. Первый — фильтровать историю изменений проектов по наличию ключевых слов *fix*, *bug*, *error* и т.д. в описании коммитов. Этот подход позволяет достаточно просто и быстро собрать большие датасеты исправлений. Недостатком таких подходов является очень простая структура описаний коммитов. Более-менее любая модель машинного обучения может показывать близкие к единице значения метрик качества (Precision, Recall, Accuracy, ROC-AUC), опираясь на текст описаний. Однако практического значения это иметь не будет, так как при применении на новых данных модель будет считать исправлениями только коммиты с ключевыми словами в описании, а это можно проверить и без машинного обучения.

[15]Датасет коммитов из Linux Kernel: `https://doi.org/10.5281/zenodo.3965234`

Альтернативным подходом является сбор данных с помощью систем багтрекинга. Однако собранные таким образом датасеты оказываются нерепрезентативными [114].

Чтобы избежать подобных проблем, было решено собирать набор данных с помощью анализа пул-реквестов проектов с открытым исходным кодом на GitHub. Выбирались пул-реквесты, посвящённые исправлению ошибок в проекте, из них извлекались все коммиты, добавленные до первого комментария к пул-реквесту. Наши наблюдения показали, что с большой точностью коммиты до первых комментариев посвящены исправлению ошибок, а после коммитов — исправлению замечаний к стилю, рефакторингу и так далее. Для определения релевантных пул-реквестов использовалась встроенная в GitHub система меток. Метка — это функциональность платформы GitHub, позволяющая присваивать пул-реквестам категории. В рамках данного исследования считалось, что пул-реквест исправляет ошибку либо если он сам имеет соответствующую метку (*bug*, *bugfix*, *fix* и т.д.), либо если ссылается на сообщение об ошибке (issue) с соответсвующей меткой.

Описанный подход был применён ко всем проектам с открытым исходным кодом на GitHub, написанным на языке Python и имеющим подходящие лицензии. Для этого сначала был взят список проектов с GitHub Archive[16], проекты были отфильтрованы по количеству звёзд. Получился список из около 91 735 проектов. Далее с помощью публичного API GitHub были получены данные об использованных языках в каждом из проектов, и из этих проектов были оставлены только те, главным языком в которых был Python. Попутно были удалены проекты, являющиеся клонами (fork) других проектов в выборке. Получилось 25 371 проектов. Далее были отфильтрованы только проекты, имеющие лицензии MIT, Apache-2.0 или BSD-3, так как эти лицензии разрешающие и позволяют переиспользовать код для своих целей с условием сохранения текста лицензии и соотвествующих копирайтов. Для каждого проекта были собраны и проанализированы данные о последних 10 000 пул-реквестах, и с помощью описанного выше подхода были отобраны пул-реквесты, исправляющие ошибки. Для каждого пул-реквеста была собрана информация о коммитах и комментариях для последующей фильтра-

[16]GitHub Archive: `https://www.gharchive.org/`

ции побочных коммитов, исправляющих замечания к пул-реквесту. Из пул-реквестов, сочтённых нерелевантными, были случайно выбраны негативные примеры коммитов. Итоговый размер выборки показан в Таблице 4.2.

Таблица 4.2: Информация об итоговом размере выборок

Выборка	**Проекты**	**Коммиты**
Обучающая	4 438	70 365
Валидационная	583	9 247
Тестовая	552	9 675
Всего	5 573	89 287

4.3 Апробация

4.3.1 Целевая метрика

В качестве целевой метрики использовалась та же метрика, которая была в референсных исследованиях — ROC-AUC. Эта метрика считается как площадь под графиком кривой *Receiver Operating Characteristic* [35], показывающей отношение доли правильно классифицированных положительных примеров к доле правильно классифицированных отрицательных примеров при варьировании порогового значения модели.

4.3.2 RandomForest на датасете Linux Kernel

К сожалению, использованный нами подход к выделению графов изменений [46] реализован только для Java и Python, поэтому применить GGNN на данной выборке пока не представляется возможным. Также не получится и применить BERT, так как описания коммитов в датасете уже предобработаны и пропущены через стемминг, что не совместимо с процедурой подготовки данных для предобученной модели BERT. Для сравнения сложностей выборок была обучена модель RandomForest. Векторизация коммитов осуществлялась при помощи мешка токенов из описаний к данным коммитам. Использовалось оригинальное разбиение данных на обучающую и тестовую выборки. Результаты сравнения модели RandomForest с PatchNet [78]

и CC2Vec [17] показаны в Таблице 4.3. Значения метрик для референсных моделей взято из оригинальных статей.

Таблица 4.3: Сравнение моделей на выборке с Linux Kernel

Подход	**ROC-AUC**
PatchNet [78]	0.860
CC2Vec [17]	0.916
RandomForest (эта работа)	0.959

Вопреки ожиданиям, RandomForest, используя только информацию о текстовых описаниях коммитов, превзошёл обе референсные модели. Исследуя эту аномалию, было выяснено, что длина описаний коммитов в данной выборке сильно коррелирует с вероятностью коммита быть bug-fix коммитом. Видимо, это связано с тем, что исправления критических ошибок документируют в комментариях намного усерднее, чем остальные коммиты. Среди коммитов, длина описания которых превышает 20 токенов, процент bug-fix коммитов превышает 80%, а если брать ограничение в 30 символов, то и все 90%. С другой стороны, коммиты, описания которых были короче 10 токенов, в 80% процентах случаев bug-fix коммитами не являлись. Судя по всему, данная аномалия в датасете даёт значительное преимущество моделям, опирающимся на подсчёт количества слов.

4.3.3 Итоговое сравнение

Основное сравнение моделей проводилось на выборке коммитов, собранных с платформы GitHub. В качестве референсных моделей использовались подходы PatchNet [78] и CC2Vec [17]. Так как оригинальные реализации были рассчитаны на работу с кодом на языке C/C++, пришлось самостоятельно адаптировать предобработку фрагментов кода для поддержки языка Python. Итоговые результаты сравнения показаны в Таблице 4.4.

Как и ожидалось, PatchNet показал результаты несколько хуже, чем CC2Vec, хотя на этой выборке разница оказалось не такой существенной. Также можно видеть, что в целом описания коммитов информативнее изменений кода — даже RandomForest показал лучшие результаты, нежели

Таблица 4.4: Сравнение моделей на выборке с GitHub

Подход	**ROC-AUC**
PatchNet [78]	0.692
CC2Vec [17]	0.703
RandomForest (эта работа)	0.713
BERT (эта работа)	0.742
GGNN (эта работа)	0.663
GGNN + BERT (эта работа)	0.761

GGNN. Предложенный в данной работе подход, использующий ансамбль из двух моделей, превзошёл результаты референсных моделей, а также моделей GGNN и BERT по отдельности.

4.3.4 Выводы

В данной главе проверялась применимость предложенной архитектуры для определения bug-fix коммитов. Для сравнения были использованы два комбинированных подхода — PatchNet [78] и CC2Vec [17]. Ансамбль из моделей GGNN и BERT показал лучшие результаты: 0.761 по метрике ROC-AUC против 0.703 у лучшей из референсных моделей.

Заключение

В ходе данной работы были получены следующие результаты.

1. Выделены основные виды данных в задачах программной инженерии.
2. Предложена эффективная архитектура на основе ансамблей для объединения информации из разных источников в задачах программной инженерии.
3. Обучена модель для решения проблемы предсказания тегов задач спортивного программирования.
4. Проведено сравнение подходов к предсказанию тегов задач спортивного программированная и показано, что предложенная в данной работе архитектура превосходит лучшее из существоваших ранее решений на $0,154$ по метрике PR-AUC ($0,542$ у ансамбля из моделей GGNN и BERT против $0,388$ у RandomForest с метриками кода).
5. Обучена модель для решения проблемы определения bug-fix коммитов.
6. Проведено сравнение подходов к определения bug-fix коммитов и показано, что предложенная в данной работе архитектура превосходит лучшее из существоваших ранее решений на $0,058$ по метрике ROC-AUC ($0,761$ у ансамбля из моделей GGNN и BERT против $0,703$ у CC2Vec [17])

Из полученных результатов можно сделать вывод, что предложенная архитектура может успешно применяться для объединения информации из разных источников в задачах программной инженерии.

Список литературы

[1] Allamanis Miltiadis, Brockschmidt Marc, Khademi Mahmoud. Learning to Represent Programs with Graphs // ArXiv. — 2018. — Vol. abs/1711.00740.

[2] Arumugam Lakshmanan. Semantic code search using Code2Vec: A bag-of-paths model : Master's thesis / Lakshmanan Arumugam ; University of Waterloo. — 2020.

[3] Attention-based LSTM for aspect-level sentiment classification / Yequan Wang, Minlie Huang, Xiaoyan Zhu, Li Zhao // Proceedings of the 2016 conference on empirical methods in natural language processing. — 2016. — P. 606–615.

[4] Attention is all you need / Ashish Vaswani, Noam Shazeer, Niki Parmar et al. // Advances in neural information processing systems. — 2017. — P. 5998–6008.

[5] Automatic algorithm recognition of source-code using machine learning / Maged Shalaby, Tarek Mehrez, Amr El Mougy et al. // 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA). — 2017. — P. 170–177.

[6] Automatic metric thresholds derivation for code smell detection / Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, Aiko Yamashita // 2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics / IEEE. — 2015. — P. 44–53.

[7] Automatic source code summarization with extended tree-LSTM / Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto et al. // 2019 International Joint Conference on Neural Networks (IJCNN) / IEEE. — 2019. — P. 1–8.

[8] Baron Kilby. Evaluating the Effectiveness of Code2Vec for Bug Prediction When Considering That Not All Bugs Are the Same : Master's thesis / Kilby Baron ; University of Waterloo. — 2020.

[9] Bert: Pre-training of deep bidirectional transformers for language understanding / Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova // arXiv preprint arXiv:1810.04805. — 2018.

[10] Bhatia Sahil, Singh Rishabh. Automated correction for syntax errors in programming assignments using recurrent neural networks // arXiv preprint arXiv:1603.06129. — 2016.

[11] Big code!= big vocabulary: Open-vocabulary models for source code / Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes et al. // 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE) / IEEE. — 2020. — P. 1073–1085.

[12] Bille Philip. A survey on tree edit distance and related problems // Theoretical computer science. — 2005. — Vol. 337, no. 1-3. — P. 217–239.

[13] Black-Box Test-Coverage Analysis and Test-Cost Reduction Based on a Bayesian Network Model / Renjian Pan, Zhaobo Zhang, Xin Li et al. // 2019 IEEE 37th VLSI Test Symposium (VTS) / IEEE. — 2019. — P. 1–6.

[14] Blei David M., Ng Andrew Y., Jordan Michael I. Latent Dirichlet Allocation // J. Mach. Learn. Res. — 2003. — Vol. 3. — P. 993–1022.

[15] Bloomfield Aaron, Sotomayor Borja. A programming contest strategy guide // Proceedings of the 47th ACM technical symposium on computing science education. — 2016. — P. 609–614.

[16] Bora Ashish, Sinha Abhishek. Predicting algorithmic approach for programming problems from natural language problem description. — 2016.

[17] CC2Vec: Distributed representations of code changes / Thong Hoang, Hong Jin Kang, David Lo, Julia Lawall // Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. — 2020. — P. 518–529.

[18] Change detection in hierarchically structured information / Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina,

Jennifer Widom // Acm Sigmod Record / ACM. — Vol. 25. — 1996. — P. 493–504.

[19] Change distilling: Tree differencing for fine-grained source code change extraction / Beat Fluri, Michael Wuersch, Martin PInzger, Harald Gall // IEEE Transactions on software engineering. — 2007. — Vol. 33, no. 11. — P. 725–743.

[20] Choosing software metrics for defect prediction: an investigation on feature selection techniques / Kehan Gao, Taghi M Khoshgoftaar, Huanjing Wang, Naeem Seliya // Software: Practice and Experience. — 2011. — Vol. 41, no. 5. — P. 579–606.

[21] Classification and Recommendation of Competitive Programming Problems Using CNN / S Sudha, A Arun Kumar, M Muthu Nagappan, R Suresh // International Conference on Intelligent Information Technologies. — 2017. — P. 262–272.

[22] Classification of programming problems based on topic modeling / Chowdhury Md Intisar, Yutaka Watanobe, Manoj Poudel, Subhash Bhalla // Proceedings of the 2019 7th International Conference on Information and Education Technology. — 2019. — P. 275–283.

[23] Codebert: A pre-trained model for programming and natural languages / Zhangyin Feng, Daya Guo, Duyu Tang et al. // arXiv preprint arXiv:2002.08155. — 2020.

[24] Comparing heuristic and machine learning approaches for metric-based code smell detection / Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, Andrea De Lucia // 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC) / IEEE. — 2019. — P. 93–104.

[25] Convolutional sequence to sequence learning / Jonas Gehring, Michael Auli, David Grangier et al. // International Conference on Machine Learning / PMLR. — 2017. — P. 1243–1252.

[26] Davis Jesse, Goadrich Mark. The relationship between Precision-Recall and ROC curves // Proceedings of the 23rd international conference on Machine learning. — 2006. — P. 233–240.

[27] DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction / Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei et al. // 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR) / IEEE. — 2019. — P. 34–45.

[28] Deepcpdp: Deep learning based cross-project defect prediction / Deyu Chen, Xiang Chen, Hao Li et al. // IEEE Access. — 2019. — Vol. 7. — P. 184832–184848.

[29] Distributed representations of words and phrases and their compositionality / Tomas Mikolov, Ilya Sutskever, Kai Chen et al. // Advances in neural information processing systems. — 2013. — P. 3111–3119.

[30] Dominici Massimiliano. An overview of Pandoc // TUGboat. — 2014. — Vol. 35, no. 1. — P. 44–50.

[31] Dropout: A Simple Way to Prevent Neural Networks from Overfitting / Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky et al. // Journal of Machine Learning Research. — 2014. — 06. — Vol. 15. — P. 1929–1958.

[32] Dual graph convolutional neural network for predicting chemical networks / Shonosuke Harada, Hirotaka Akita, Masashi Tsubaki et al. // BMC bioinformatics. — 2020. — Vol. 21. — P. 1–13.

[33] Embedding Java classes with code2vec: improvements from variable obfuscation [Accepted] / Rhys Compton, Eibe Frank, Panagiotis Patros, Abigail Koay // MSR 2020 / ACM. — 2020.

[34] Exploring the limits of transfer learning with a unified text-to-text transformer / Colin Raffel, Noam Shazeer, Adam Roberts et al. // arXiv preprint arXiv:1910.10683. — 2019.

[35] Fawcett Tom. An introduction to ROC analysis // Pattern recognition letters. — 2006. — Vol. 27, no. 8. — P. 861–874.

[36] Fenton Norman E, Neil Martin. Software metrics: successes, failures and new directions // Journal of Systems and Software. — 1999. — Vol. 47, no. 2-3. — P. 149–157.

[37] Fine-grained and accurate source code differencing / Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc et al. // ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. — 2014. — P. 313–324. — Access mode: http://doi.acm.org/10.1145/2642937.2642982.

[38] Friedman Jerome H. Greedy function approximation: a gradient boosting machine // Annals of statistics. — 2001. — P. 1189–1232.

[39] Gage Philip. A new algorithm for data compression // C Users Journal. — 1994. — Vol. 12, no. 2. — P. 23–38.

[40] Gated Graph Attention Network for Cancer Prediction / Linling Qiu, Han Li, Meihong Wang, Xiaoli Wang // Sensors. — 2021. — Vol. 21, no. 6. — P. 1938.

[41] Gated graph sequence neural networks / Yujia Li, Daniel Tarlow, Marc Brockschmidt, Richard Zemel // arXiv preprint arXiv:1511.05493. — 2015.

[42] Ghosh Souvick, Shah Chirag. Towards automatic fake news classification // Proceedings of the Association for Information Science and Technology. — 2018. — Vol. 55, no. 1. — P. 805–807.

[43] Gilbert Henri, Handschuh Helena. Security analysis of SHA-256 and sisters // International workshop on selected areas in cryptography. — 2003. — P. 175–193.

[44] Google's neural machine translation system: Bridging the gap between human and machine translation / Yonghui Wu, Mike Schuster, Zhifeng Chen et al. // arXiv preprint arXiv:1609.08144. — 2016.

[45] Gower John C, Warrens Matthijs J. Similarity, dissimilarity, and distance, measures of // Wiley StatsRef: Statistics Reference Online. — 2014. — P. 1–11.

[46] Graph-based mining of in-the-wild, fine-grained, semantic code change patterns / Hoan Anh Nguyen, Tien N Nguyen, Danny Dig et al. // 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) / IEEE. — 2019. — P. 819–830.

[47] Graph neural networks: A review of methods and applications / Jie Zhou, Ganqu Cui, Zhengyan Zhang et al. // arXiv preprint arXiv:1812.08434. — 2018.

[48] Hajiaghayi Mahdi, Vahedi Ehsan. Code failure prediction and pattern extraction using LSTM networks // 2019 IEEE Fifth International Conference on Big Data Computing Service and Applications (BigDataService) / IEEE. — 2019. — P. 55–62.

[49] Hamano Junio C. GIT–A stupid content tracker // Proc. Ottawa Linux Sympo. — 2006. — Vol. 1. — P. 385–394.

[50] Hamming R. W. Error detecting and error correcting codes // The Bell System Technical Journal. — 1950. — Vol. 29, no. 2. — P. 147–160.

[51] Handling Missing Sensors in Topology-Aware IoT Applications with Gated Graph Neural Network / Shengzhong Liu, Shuochao Yao, Yifei Huang et al. // Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies. — 2020. — Vol. 4, no. 3. — P. 1–31.

[52] Hastie Trevor, Tibshirani Robert, Friedman Jerome. The elements of statistical learning: data mining, inference, and prediction. — 2009.

[53] Ho Tin Kam. Random decision forests // Proceedings of 3rd international conference on document analysis and recognition. — Vol. 1. — 1995. — P. 278–282.

[54] Hosseini Seyedrebvar, Turhan Burak, Gunarathna Dimuthu. A systematic literature review and meta-analysis on cross project defect prediction // IEEE Transactions on Software Engineering. — 2017. — Vol. 45, no. 2. — P. 111–147.

[55] Huang Jin, Ling Charles X. Using AUC and accuracy in evaluating learning algorithms // IEEE Transactions on knowledge and Data Engineering. — 2005. — Vol. 17, no. 3. — P. 299–310.

[56] Improving language understanding by generative pre-training / Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever. — 2018.

[57] Islam Md Rakibul, Zibran Minhaz F. How bugs are fixed: exposing bug-fix patterns with edits and nesting levels // Proceedings of the 35th Annual ACM Symposium on Applied Computing. — 2020. — P. 1523–1531.

[58] Kaur Sukhdeep, Maini Raman. Analysis of various software metrics used to detect bad smells // Int J Eng Sci (IJES). — 2016. — Vol. 5, no. 6. — P. 14–20.

[59] Kim Sunghun, Whitehead E James, Zhang Yi. Classifying software changes: Clean or buggy? // IEEE Transactions on Software Engineering. — 2008. — Vol. 34, no. 2. — P. 181–196.

[60] Kim Yoon. Convolutional Neural Networks for Sentence Classification // Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). — 2014. — P. 1746–1751.

[61] Larose Daniel T, Larose Chantal D. Discovering knowledge in data: an introduction to data mining. — 2014.

[62] Le Quoc V., Mikolov Tomas. Distributed Representations of Sentences and Documents // International Conference on Machine Learning. — 2014.

[63] Learning phrase representations using RNN encoder-decoder for statistical machine translation / Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre et al. // arXiv preprint arXiv:1406.1078. — 2014.

[64] Learning traffic as a graph: A gated graph wavelet recurrent neural network for network-scale traffic prediction / Zhiyong Cui, Ruimin Ke, Ziyuan Pu et al. // Transportation Research Part C: Emerging Technologies. — 2020. — Vol. 115. — P. 102620.

[65] Lee Daniel D, Seung H Sebastian. Algorithms for non-negative matrix factorization // Advances in neural information processing systems. — 2001. — P. 556–562.

[66] Levin Stanislav, Yehudai Amiram. Boosting automatic commit classification into maintenance activities by utilizing source code changes // Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering. — 2017. — P. 97–106.

[67] Liu Yang, Lapata Mirella. Text summarization with pretrained encoders // arXiv preprint arXiv:1908.08345. — 2019.

[68] Maimon Oded, Rokach Lior. Data mining and knowledge discovery handbook. — 2005.

[69] Mass-produced software components / M Douglas McIlroy, J Buxton, Peter Naur, Brian Randell // Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany. — 1968. — P. 88–98.

[70] Mockus Audris, Votta Lawrence G. Identifying Reasons for Software Changes using Historic Databases. // icsm. — 2000. — P. 120–130.

[71] Multi-label Classification for Automatic Tag Prediction in the Context of Programming Challenges / Bianca Iancu, Gabriele Mazzola, Kyriakos Psarakis, Panagiotis Soilis // arXiv preprint arXiv:1911.12224. — 2019.

[72] Naur Peter, Randell Brian. Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7th-11th october 1968. — 1969.

[73] Nayrolles Mathieu, Hamou-Lhadj Abdelwahab. CLEVER: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects // Proceedings of the 15th International Conference on Mining Software Repositories. — 2018. — P. 153–164.

[74] Neural-machine-translation-based commit message generation: how far are we? / Zhongxin Liu, Xin Xia, Ahmed E Hassan et al. // Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. — 2018. — P. 373–384.

[75] On the" naturalness" of buggy code / Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane et al. // 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE) / IEEE. — 2016. — P. 428–439.

[76] Osman Haidar, Lungu Mircea, Nierstrasz Oscar. Mining frequent bug-fix code changes // 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE) / IEEE. — 2014. — P. 343–347.

[77] Pan Kai, Kim Sunghun, Whitehead E James. Toward an understanding of bug fix patterns // Empirical Software Engineering. — 2009. — Vol. 14, no. 3. — P. 286–315.

[78] PatchNet: Hierarchical Deep Learning-Based Stable Patch Identification for the Linux Kernel / Thong Hoang, Julia Lawall, Yuan Tian et al. // IEEE Transactions on Software Engineering. — 2019.

[79] Pearson correlation coefficient / Jacob Benesty, Jingdong Chen, Yiteng Huang, Israel Cohen // Noise reduction in speech processing. — 2009. — P. 1–4.

[80] Pennington Jeffrey, Socher Richard, Manning Christopher D. Glove: Global vectors for word representation // Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). — 2014. — P. 1532–1543.

[81] Peters Matthew E, Ruder Sebastian, Smith Noah A. To tune or not to tune? adapting pretrained representations to diverse tasks // arXiv preprint arXiv:1903.05987. — 2019.

[82] Pilato C Michael, Collins-Sussman Ben, Fitzpatrick Brian W. Version control with subversion: next generation open source version control. — " O'Reilly Media, Inc.", 2008.

[83] Powers David Martin. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. — 2011.

[84] Pre-trained contextual embedding of source code / Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, Kensen Shi // arXiv preprint arXiv:2001.00059. — 2019.

[85] Prest: An Intelligent Software Metrics Extraction, Analysis and Defect Prediction Tool. / Ekrem Kocaguneli, Ayse Tosun, Ayse Basar Bener et al. // SEKE. — 2009. — P. 637–642.

[86] Ramos Juan et al. Using tf-idf to determine word relevance in document queries // Proceedings of the first instructional conference on machine learning. — Vol. 242. — 2003. — P. 133–142.

[87] Randell Brian. The 1968/69 nato software engineering reports // History of software engineering. — 1996. — Vol. 37.

[88] Recommending refactoring solutions based on traceability and code metrics / Ally S Nyamawe, Hui Liu, Zhendong Niu et al. // IEEE Access. — 2018. — Vol. 6. — P. 49460–49475.

[89] Recommending tags for pull requests in GitHub / Jing Jiang, Qiudi Wu, Jin Cao et al. // Information and Software Technology. — 2021. — Vol. 129. — P. 106394.

[90] Rochkind Marc J. The source code control system // IEEE transactions on Software Engineering. — 1975. — no. 4. — P. 364–370.

[91] Rodríguez-Pérez Gema, Robles Gregorio, González-Barahona Jesús M. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm // Information and Software Technology. — 2018. — Vol. 99. — P. 164–176.

[92] Rosen Christoffer, Grawi Ben, Shihab Emad. Commit guru: analytics and risk prediction of software commits // Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. — 2015. — P. 966–969.

[93] Ruparelia Nayan B. The history of version control // ACM SIGSOFT Software Engineering Notes. — 2010. — Vol. 35, no. 1. — P. 5–9.

[94] Schuster Mike, Nakajima Kaisuke. Japanese and korean voice search // 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) / IEEE. — 2012. — P. 5149–5152.

[95] A Self-Attentional Neural Architecture for Code Completion with Multi-Task Learning / Fang Liu, Ge Li, Bolin Wei et al. // Proceedings of the 28th International Conference on Program Comprehension. — 2020. — P. 37–47.

[96] Sennrich Rico, Haddow Barry, Birch Alexandra. Neural Machine Translation of Rare Words with Subword Units // Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). — 2016. — P. 1715–1725.

[97] Shrivastava Manish et al. Predicting Algorithm Classes for Programming Word Problems // Proceedings of the 5th Workshop on Noisy User-generated Text (W-NUT 2019). — 2019. — P. 84–93.

[98] Software bug prediction using machine learning approach / Awni Hammouri, Mustafa Hammad, Mohammad Alnabhan, Fatima Alsarayrah // International Journal of Advanced Computer Science and Applications. — 2018. — Vol. 9, no. 2. — P. 78–83.

[99] Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks / Antonio Mastropaolo, Simone Scalabrino,

Nathan Cooper et al. // 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) / IEEE. — 2021. — P. 336–347.

[100] Sun Chi, Huang Luyao, Qiu Xipeng. Utilizing BERT for aspect-based sentiment analysis via constructing auxiliary sentence // arXiv preprint arXiv:1903.09588. — 2019.

[101] Sundermeyer Martin, Schlüter Ralf, Ney Hermann. LSTM neural networks for language modeling // Thirteenth annual conference of the international speech communication association. — 2012.

[102] Tackling the poor assumptions of naive bayes text classifiers / Jason D Rennie, Lawrence Shih, Jaime Teevan, David R Karger // Proceedings of the 20th international conference on machine learning (ICML-03). — 2003. — P. 616–623.

[103] Terada Kenta, Watanobe Yutaka. Code Completion for Programming Education based on Recurrent Neural Network // 2019 IEEE 11th International Workshop on Computational Intelligence and Applications (IWCIA) / IEEE. — 2019. — P. 109–114.

[104] Tichy Walter F. Design, implementation, and evaluation of a Revision Control System // Proceedings of the 6th international conference on Software engineering. — 1982. — P. 58–67.

[105] Turhan Burak, Bener Ayse Basar. Software Defect Prediction: Heuristics for Weighted Naïve Bayes. // ICSOFT (SE). — 2007. — P. 244–249.

[106] Twitter sentiment analysis via bi-sense emoji embedding and attention-based LSTM / Yuxiao Chen, Jianbo Yuan, Quanzeng You, Jiebo Luo // Proceedings of the 26th ACM international conference on Multimedia. — 2018. — P. 117–125.

[107] Typilus: neural type hints / Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, Zheng Gao // Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. — 2020. — P. 91–105.

[108] Yuill Simon. Concurrent versions system // Software Studies: A Lexicon. — 2008. — Vol. 64.

[109] Zhang Hongyu, Zhang Xiuzhen. Comments on Data Mining Static Code Attributes to Learn Defect Predictors // Software Engineering, IEEE Transactions on. — 2007. — 10. — Vol. 33. — P. 635 – 637.

[110] code2seq: Generating sequences from structured representations of code / Uri Alon, Shaked Brody, Omer Levy, Eran Yahav // arXiv preprint arXiv:1808.01400. — 2018.

[111] code2vec: Learning distributed representations of code / Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav // Proceedings of the ACM on Programming Languages. — 2019. — Vol. 3, no. POPL. — P. 1–29.

[112] An empirical investigation into learning bug-fixing patches in the wild via neural machine translation / Michele Tufano, Cody Watson, Gabriele Bavota et al. // Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. — 2018. — P. 832–837.

[113] A graph-convolutional neural network model for the prediction of chemical reactivity / Connor W Coley, Wengong Jin, Luke Rogers et al. // Chemical science. — 2019. — Vol. 10, no. 2. — P. 370–377.

[114] The missing links: bugs and bug-fix commits / Adrian Bachmann, Christian Bird, Foyzur Rahman et al. // Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. — 2010. — P. 97–106.