Санкт–Петербургский государственный университет

Павлов Иван Александрович

Выпускная квалификационная работа

Обнаружение устаревших комментариев в коде

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5005.2018 «Прикладная математика, фундаментальная информатика и программирование»
Профиль «Современное программирование»

Научный руководитель:
профессор СПбГУ, д.ф.-м.н.
Куликов Александр Сергеевич

Научный консультант:
Курбатова Зарина Идиевна

Рецензент:
старший преподаватель
НИУ ВШЭ в Санкт-Петербурге, к.т.н.
Шпильман Алексей Александрович

Санкт-Петербург
2022 г.

Содержание

Введение

В процессе написания и чтения кода разработчики оставляют к нему комментарии. Читаемость кода во многом зависит от качества комментариев. В исследовании Вудфилда и др. [1] проведен один из первых экспериментов, демонстрирующий положительное влияние комментариев на читаемость программного кода. В дальнейшем это было подтверждено другими исследователями [2, 3].

В то же время, исследования показывают, что сопровождение программного обеспечения – это одна из самых важных частей процесса разработки программного обеспечения (ПО) [4]. Значительную часть сопровождения и написания ПО занимает чтение и понимание уже написанного кода. Например, исследования Пигоски и др. [4], а также Пфлегера и др. [5], сообщают, что от 40 до 60 процентов времени сопровождения ПО тратится на его изучение. Улучшая читаемость кода, качественные комментарии упрощают поддержку ПО. Однако, чтобы поддерживать должное качество комментариев, разработчик должен прикладывать усилия и при необходимости вносить соответствующие правки при каждом обновлении кода, что происходит не всегда [6, 7].

Изменение кода без соответствующих изменений в комментарии может привести к противоречивости кода и комментария. Такие противоречивые комментарии негативно влияют на читаемость кода, принося неоднозначность [6, 8], увеличивая возможность программных ошибок [9]. Чтобы помочь программистам следить за качеством комментариев, исследователи предлагают системы автоматического обнаружения устаревших комментариев.

Устаревшие комментарии (также называемые *inconsistent comments*) – комментарии, описывающие фрагмент программного кода, но сообщающие об этом фрагменте неверную информацию. Часто такие комментарии возникают, когда разработчик при изменении функциональности кода забывает внести соответствующие правки в комментарий.

На Рисунке 1 приведен пример кода на Java из реального проекта seppuku [10]. Комментарий над функцией не соответствует логике ее ко-

```java
/**
 * Removes and existing file or creates a new file
 */
public static File recreateFile(File file) {
    if (file.exists()) {
        file.delete();
    }

    try {
        file.createNewFile();
    } catch (IOException e) {
        System.err.println("IOException thrown, could not recreate file.");
    }

    return file;
}
```

Рис. 1: Пример устаревшего комментария

да. В комментарии сказано, что метод удаляет существующий файл **или** создает новый. Если посмотреть на код, становится ясно, что новый файл создается в любом случае. Таким образом, комментарий противоречит коду.

В этой дипломной работе исследуются способы обнаружения таких *устаревших комментариев*, а также проблема разметки и сбора примеров устаревших комментариев из существующих проектов. Данная работа полностью посвящена комментариям формата *JavaDoc* (см. раздел 1.2) к коду на языке программирования *Java*. Выбор языка программирования обусловлен тем, что *Java* является одним из самых популярных языков программирования [11] и большинство статей в этой области исследуют именно комментарии к коду на *Java*. Кроме этого, формат *JavaDoc*, который используют разработчики на *Java*, облегчает анализ комментариев и алгоритмов.

Цель и задачи

Данная работа ставит целью создание системы обнаружения устаревших комментариев в виде плагина для IntelliJ IDEA, интегрированной среды разработки (IDE, Integrated Development Environment). Для достижения этой цели были сформулированы следующие задачи:

- Изучить работы, посвященные обнаружению устаревших комментариев, и реализовать предложенные подходы;
- Разработать инструмент для разметки и извлечения примеров устаревших комментариев из репозиториев с кодом;
- Используя разработанный инструмент, собрать новый набор данных, дополненный информацией об измененных методах (совершенные рефакторинги и др.);
- Исследовать качество собранных данных, уточнить разметку части данных, используя краудсорсинг;
- Проанализировать результаты работы алгоритма обнаружения устаревших комментариев на новых, очищенных данных;
- Разработать плагин для IntelliJ IDEA, использующий один из существующих алгоритмов обнаружения устаревших комментариев.

Достигнутые результаты

В рамках данной работы разработан плагин для IDE IntelliJ IDEA, **CommentUpdater**. Это инструмент для обнаружения устаревших комментариев в коде, который подсвечивает устаревшие комментарии в реальном времени в процессе написания кода. Плагин использует модель Deep JIT [12], глубокую нейронную сеть, для обнаружения устаревших комментариев. Архитектура плагина подразумевает возможность для улучшения или модификации алгоритма обнаружения устаревших комментариев. Например, **CommentUpdater** анализирует информацию о произведенных

рефакторингах в коде. Использование этой информации в модели потенциально упрощает ее обучение, потому что часто рефакторинги становятся ключевыми причинами изменений кода. Эксперименты в этом направлении – возможный путь развития инструмента.

Кроме плагина **CommentUpdater**, в процессе дипломной работы разработан инструмент **HeadlessCommentUpdater** для сбора и автоматической разметки примеров устаревших и не устаревших комментариев. **HeadlessCommentUpdater** использует возможности IDE IntelliJ IDEA, чтобы собирать дополнительную информацию о данных. Например, инструмент собирает информацию о рефакторингах, совершенных в процессе изменения кода. Также инструмент предлагает новый способ для автоматической разметки примеров, использующий систему контроля версий. Инструмент учитывает изменения из нескольких коммитов для каждого метода, в то время как во всех работах ранее использовался только один коммит.

При помощи **HeadlessCommentUpdater** был собран и проанализирован набор данных из более чем двух миллионов примеров для задачи обнаружения устаревших комментариев из четырех тысяч проектов с открытым исходным кодом, реализованных на языке Java. Тысяча примеров из собранных была затем размечена вручную и разбита на тридцать семь кластеров. Для повышения качества автоматической разметки устаревших комментариев был разработан метод уточнения с помощью ручной проверки, использующий краудсорсинговую платформу Яндекс.Толока [13].

Глава 1. Обзор предметной области

Задача обнаружения устаревших комментариев в программном коде интересует исследователей не первое десятилетие. В этой главе описаны основные способы решения этой задачи, предложенные за это время.

1.1 Постановка задачи

В статьях на тему обнаружения устаревших комментариев задачу ставят двумя способами:

- Авторы более ранних статей [12, 14, 15, 16, 17, 18, 19, 20, 21, 22] используют постановку *Post Hoc* — в таком подходе на основе фрагмента кода и комментария требуется определить, соответствуют ли они друг другу по смыслу.

- В недавних работах [12, 14, 23, 24, 25] используют вариант *Just In Time (JIT)* — при таком подходе на основе комментария к коду и версии кода до и после изменений нужно установить, остался ли комментарий консистентным с кодом после изменений.

Фрагмент кода и комментарий, представленные на Рисунке 1, можно считать примером входных данных для *Post Hoc* подхода. В этом примере предсказание делается на основе конкретной версии кода. На Рисунке 2 представлены комментарий, версия кода до изменения (выделена красным цветом) и после (выделена зеленым цветом), что может послужить входными данными для задачи в *JIT* постановке.

Последние исследования в этой области концентрируются на *JIT* подходе. В такой постановке входные данные содержат больше информации для решения задачи, так как изменения кода могут содержать подсказки о причинах возникновения возможного несоответствия кода и комментария. Одновременно с этим, для реализации на практике, такой подход требует отслеживать изменения методов. Доступное решение – использовать систему контроля версий. Система контроля версий позволяет разработчикам сохранять информацию об изменениях кода, в том числе об изменениях

```java
/**
 * Removes and existing file or creates a new file
 */
public static File recreateFile(File file) {
    if (file.exists()) {
        file.delete();
    } else {
        try {
            file.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    try {
        file.createNewFile();
    } catch (IOException e) {
        System.err.println("IOException thrown, could not recreate file.");
    }

    return file;
}
```

Рис. 2: Пример устаревшего комментария в постановке *JIT*

методов и комментариев. Пример реализации этой идеи описан в Главе 5 об инструменте **CommentUpdater**.

1.2 Типы комментариев

Данная работа посвящена анализу устаревших комментариев формата *JavaDoc*. Основные типы комментариев описаны в таблице 1. Помимо *JavaDoc*, в Java также используются блочные и однострочные комментарии. Такие комментарии могут располагаться в любом месте программного кода, поэтому для них возникает отдельная задача соотнесения комментария и фрагмента кода, к которому он относится. Кроме этого, блочные и однострочные комментарии не имеют структуры, что также осложняет их исследование.

Комментарии *JavaDoc*, напротив, привязаны к классам или методам и имеют структуру. Изначально, *JavaDoc* комментарии были введены для автоматической генерации API (Application Programming Interface, интерфейс программирования приложений) документации [26]. *JavaDoc* комментарий должен быть написан на HTML (HyperText Markup Language) и дол-

жен предшествовать объявлению класса, поля, конструктора или метода. Он состоит из двух частей — описания в свободной форме на естественном языке и набора блочных дескрипторов. На Рисунке 3 описание отмечено как Summary, а блочные дескрипторы `@param`, `@return`, `@throws` как Param, Return и Throws соответственно. Дескриптор `@param` содержит описание параметров метода. Дескриптор `@return` описывает возвращаемое значение. Дескрипторы `@throws` указывают на возможные исключения. Формат *JavaDoc* подразумевает также использование других дескрипторов: `@author`, `@see`, `@deprecated`, и т.д.

Тип комментария	**Формат**	**Описание**
JavaDoc	/** text ... @tag ... @tag ...*/	Многострочный, структурированный комментарий, содержащий специальные дескрипторы [26], например `@deprecated`, `@return`, `@param`, `@see` и т.д. Используется перед определением класса или метода.
Блочный	/* text ...*/	Многострочный комментарий, без какой-либо структуры и дескрипторов. Используется в любом месте в коде.
Однострочный	// text ...	Комментарий из одной строки, без структуры. Используется в любом месте, часто находится перед объявлением переменной или условным выражением.

Таблица 1: Типы комментариев

1.3 Обзор литературы

1.3.1 Узконаправленные методы

Часть работ в области обнаружения устаревших комментариев концентрируется на отдельных сущностях кода, изменение которых может спровоцировать изменение комментария. Авторы отслеживают *идентификаторы* в коде [18], обращают внимание на параметры, исключения и *null*

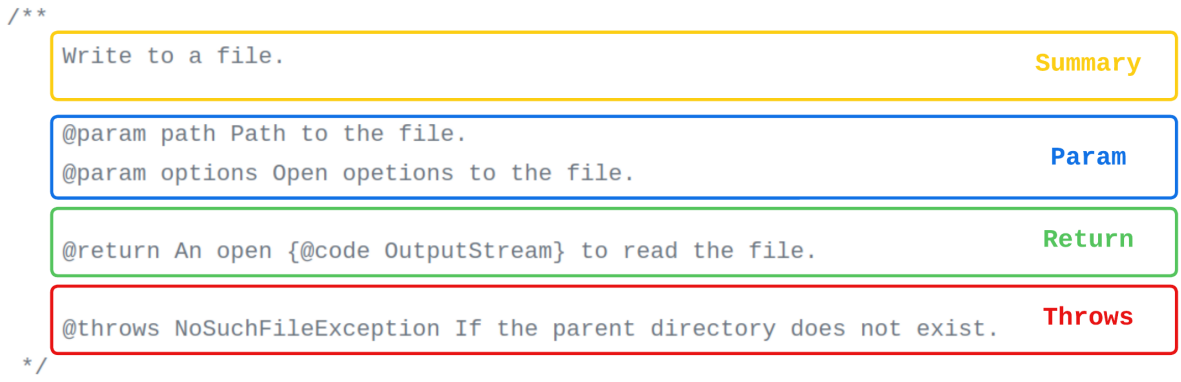

Рис. 3: Пример *JavaDoc* комментария

значения [20, 19], исследуют работу с описанием блокировок (*locks*) и прерываний в комментариях [22], [21]. Все эти работы используют техники информационного поиска, алгоритмы обработки естественного языка и подходы, основанные на правилах. Эффективные в частных случаях, такие правила тяжело обобщаются.

В этой работе исследуется обнаружение устаревших комментариев при произвольных изменениях кода. Такая задача сложнее, поскольку для её решения нужно учитывать различные причины изменения метода, использовать общие признаки, решающие задачу для различных вариантов: одновременно для изменения параметров, исключений, идентификаторов. Используемая формулировка задачи представляет больший практический интерес, поскольку охватывает более широкий диапазон возможных неточностей в комментариях.

1.3.2 Векторная семантика

В части работ авторы используют техники из классической обработки естественного языка. В работах Корацца и др. [17], а также Чимаса и др. [16] исследуется решение задачи в *Post Hoc* постановке. Авторы используют возможные связи между обнаружением устаревших комментариев и выделением лексической информации из программного кода и соответствующих комментариев. В работе Корацца и др. [17] используется векторное представление текстов *tf-idf* и классификатор *Support Vector*

Machine (SVM) [27]. Для обучения и тестирования моделей авторы разметили набор данных из 2,883 примеров. Предложенный метод достигает значения метрики *Accuracy* 0.83 (см. определение метрики 5.3). В работе Чимаса и др. [16] используются более сложные техники векторизации текстов, *word embeddings* (*GloVe* [28], *word2vec* [29]). Для тестирования авторы используют примеры из набора данных, собранного в статье Коравца и др. [17] В основном авторы концентрируются на сравнении различных способов построения векторных представлений слов для решения задачи обнаружения устаревших комментариев. Авторы также используют SVM и достигают максимального значения метрики *Accuracy* 0.80.

1.3.3 Механизмы сопоставления кода и комментария

В работах Стуловой и др. [23] и Саду [24] авторы используют похожую идею — сопоставление фрагментов кода и комментария (matching algorithms). С помощью эвристических правил и метрик схожести авторы сопоставляют части комментария и кода. Каждому предложению комментария сопоставляется фрагмент из кода. Например, описание после дескриптора @param должно быть сопоставлено соответствующему параметру в сигнатуре метода. Таким образом, при изменении части кода алгоритм выделяет возможные предложения в комментарии, которые требуется также изменить. В работе Стуловой и др. [23] авторы используют Word's Mover Distance (WMD) [30] для оценки схожести части комментария и метода и достигают оценки точности предсказания 90%. Однако авторы используют только 67 примеров для оценки результата. В работе Саду [24] используются метрики по оценке схожести на уровне слов комментария и токенов кода. Автор, используя 30 примеров для оценки результата, достигает значения метрики *F1-score* равного 0.81 (см. определение метрики 5.3). Такой подход хорошо интерпретируется, но тяжело обобщается за пределы эвристических правил и подобранных констант.

1.3.4 Классическое машинное обучение

В статье Лиу и др. [25] используется классическое машинное обучение. Исследователи решают *JIT* задачу не только для комментариев к методам, но и для однострочных комментариев внутри методов. С помощью эвристических правил выделяются признаки комментария и кода. Признаки используют информацию как уровня языковых выражений, так и уровня методов и классов. Например, признаки кода содержат информацию об изменениях атрибута класса, добавлении параметров в объявление метода, процент измененных выражений в коде метода и так далее. Признаки комментария содержат следующую информацию: вхождение специфичных токенов, например, *"fixed bug"*; длину комментария; метрику схожести комментария и кода до и после изменения, основанную на расстоянии между векторными представлениями слов. Выделенные признаки используются как вход для случайного леса [31], алгоритма классификации, широко распространенного в машинном обучении. Итоговая модель демонстрирует результат 0.72 по метрике *F1-score*. По сравнению с нейронными сетями, такой подход имеет высокую скорость работы на новых данных, а также его результаты интерпретируемы. Однако метод трудно обобщается за пределы эвристических признаков. Кроме этого, метод лишь косвенно учитывает семантические связи кода и комментария.

1.3.5 Глубокое обучение

Во многом данная дипломная работа опирается на методы, предложенные Пантаплакель и др. [12]. Авторы используют глубокие нейронные сети, чтобы решать задачу как в *Post Hoc*, так и в *JIT* постановке. Архитектура сети, предложенной авторами, представлена на Рисунке 5 (далее данная модель упоминается под названием **Deep JIT**). На вход сети подается комментарий, обозначим его за D. Кроме этого, в PostHoc постановке сети подается код метода M, а в JIT постановке изменения кода метода M_{edit}. Представление в виде последовательности токенов (слов и частей слов) для комментария D и кода M формируется с помощью стандартного для обработки естественных языков процесса: токенизация, фильтрация,

векторизация. Векторизация осуществляется с помощью Embedding слоя сети, сопоставляющего токенам численные векторы с обучаемыми параметрами. Изменения кода M_{edit} содержат в своем представлении кроме токенов кода также токены изменений, пример представлен на Рисунке 4.

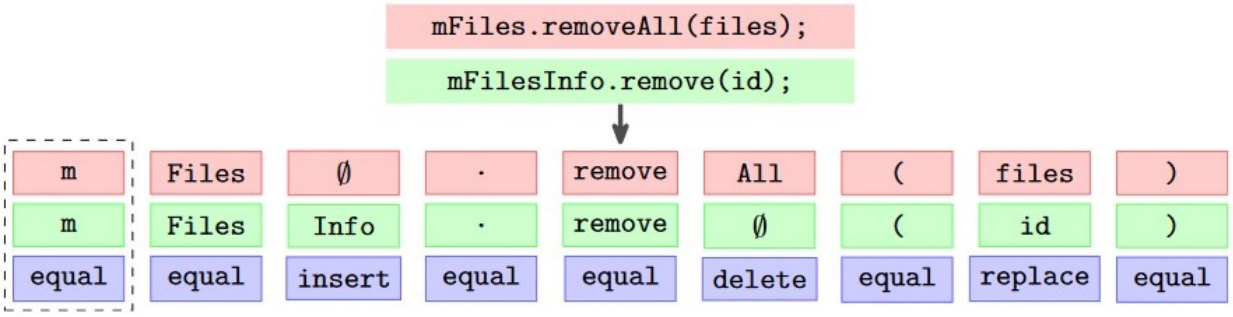

Рис. 4: Представление изменений кода

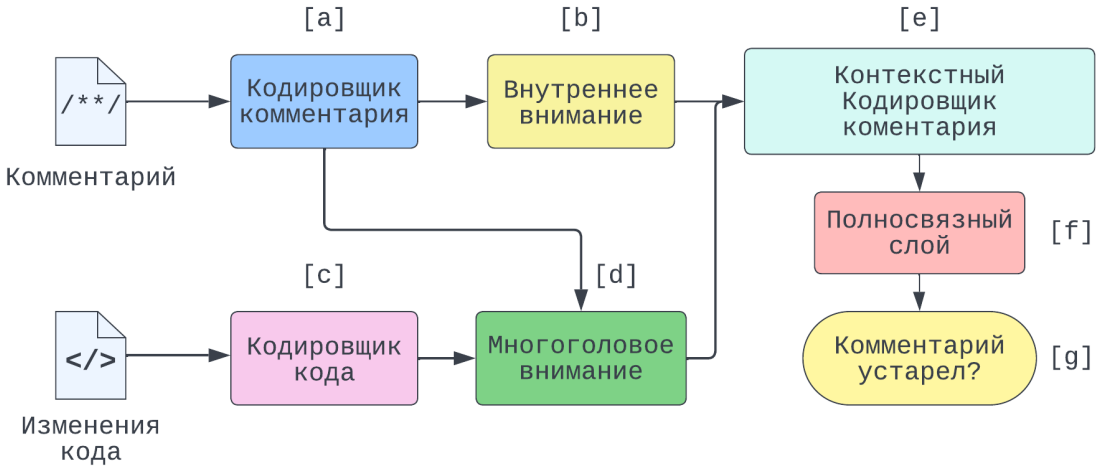

Рис. 5: Архитектура модели `Deep-JIT`

Представление комментария подается на вход кодировщику (Рисунок 5[a]). Кодировщик (encoder) – рекуррентная нейронная сеть GRU [32]. Кодировщик, считывая представления токенов, формирует внутреннее представление о контексте комментария. Однако из-за особенности архитектуры рекуррентных сетей, внутренний контекст кодировщика может терять информацию о прошлых токенах, особенно если количество токенов в последовательности большое. Чтобы бороться с этим, авторы используют

механизм внутреннего внимания (self-attention [33]) (Рисунок 5 [b]), связывающий представления, полученные кодировщиком на разных шагах чтения входной последовательности. Представление кода также подается на вход кодировщику (Рисунок 5 [c]), также представленного GRU. Чтобы связать контексты кода и комментария, построенные кодировщиками, авторы используют многоголовое внимание (Multi-Head attention)(Рисунок 5 [d]). Комбинированные векторы, полученные в модулях внимания (Рисунок 5 [b, d]), передаются на вход еще одному GRU кодировщику (Рисунок 5 [e]). Выход финального кодировщика передается в полносвязный слой (Рисунок 5 [f]), который выдает финальное предсказание (Рисунок 5 [g]).

Используя такой подход, авторы достигают результата 0.80 по метрике *F1-score*. Предложенное решение не только лучше других по качеству, но и работает с любыми типами комментариев (Summary, Return, Param, Throws). Авторы также не концентрируются на специфичных изменениях метода.

Это не единственная статья в этой области, использующая глубокие нейронные сети. В статье Лиу и др. [14] авторы используют очень похожую архитектуру, включающую несколько кодировщиков и механизм внимания. Вместо GRU [32] кодировщика авторы используют LSTM [34]. В этой же статье описывается метод генерации комментариев для обновления устаревших примеров. Подход также использует LSTM [34] и механизм внимания. Однако качество результата не дотягивает до практической применимости. Авторы сообщают о достижении значения метрики *F1-score* 0.27.

Работа Рабби и Сиддик [15] также использует рекуррентные нейронные сети для решения этой задачи. Авторы предлагают обучать сиамские нейронные сети – комбинацию двух идентичных нейронных сетей, тренируемых одновременно. В качестве сетей авторы используют LSTM [34]. Первой модели на вход подается представление кода, второй — комментария. Данный вид сетей позволяет сравнить векторы признаков двух объектов с целью выделения их семантического сходства или различия. Подход в этой области новаторский, но протестирован только на нескольких сотнях примеров. Такой малый объем данных затрудняет анализ качества этой модели.

1.3.6 Данные

Несмотря на множество работ в данной области, трудно назвать проблему обнаружения устаревших комментариев в коде окончательно решенной. Даже модель Deep JIT [12] на практике выдает слишком много ложно положительных результатов.

Одна из проблем — недостаточное количество данных для решения данной задачи способами машинного обучения. В ряде статей обучение и тестирование производится на выборках размером в несколько десятков примеров [23, 24, 18, 22]. В других статьях используется несколько тысяч примеров, но из очень небольшого числа проектов [21, 20, 15]. Коравцца и др. [17] и Чимаса и др. [16] используют один и тот же набор данных, состоящий из нескольких тысяч примеров, собранных из пяти проектов. Примечательно, что это единственные работы из упомянутых в обзоре, которые используют одинаковые данные. В статье Лиу и др. [25] авторы собирают 30 тысяч примеров для обучения и валидации модели, однако все собранные примеры выбраны из пяти проектов.

В работе Пантаплакель и др. [12] авторы собрали и автоматически разметили 40 тысяч примеров из полутора тысяч проектов. Кроме этого, авторы отобрали 300 примеров вручную, чтобы создать набор идеально размеченных данных. Авторы также обозначают проблему шума в автоматически размеченных данных.

В работе Лиу и др. [14] авторы собрали и автоматически разметили больше двух миллионов примеров из полутора тысяч проектов. Стоит отметить, что собранные данные содержат только первое предложение комментария. Авторы в своей работе исследуют только *Summary* часть комментария *JavaDoc*. Авторы, разметив руками 200 примеров, не нашли примеров неправильной разметки в своих данных. Однако специальной фильтрации или техник для улучшения качества разметки авторы не применяли. В свою очередь, Пантаплакель и др. [12] в своей работе признают проблему шума в данных значительной для этой задачи. При этом они собирали свои данные из тех же полутора тысяч проектов похожим образом. Кроме этого, качество итоговой модели, обученной на этих данных, *F1-*

score 0.27. Данный показатель значительно уступает значениям метрики модели Deep JIT [12].

1.4 Выводы

Существующие подходы для обнаружения устаревших комментариев к коду используют множество различных техник для решения этой задачи: классическое машинное обучение, эвристические алгоритмы сопоставления комментария и кода, глубокие нейронные сети. На данный момент не было проведено сравнения существующих подходов между собой из-за отсутствия единого набора данных для тестирования результатов моделей. Наиболее перспективным кажется подход из статьи Пантаплакель и др. [12]: используя значительный по размеру набор данных, авторы достигают высокого качества. Более того, авторы использовали подходы Корацца и др. [17] и Лиу и др. [25] для сравнения и обогнали их по метрике *F1-score*.

Можно отметить, что не все авторы детально описывают проблему сбора данных для задачи определения устаревших комментариев. В разных работах используются различные объемы и форматы данных, что затрудняет сравнение этих работ между собой. Кроме этого, качество автоматической разметки данных, используемой во многих статьях, не анализируется.

Глава 2. Инструмент для сбора данных

В обзорной главе 1.3.6 выделена проблема, общая для большинства работ данной области – отсутствие большого набора размеченных данных для обучения и тестирования моделей в задаче обнаружения устаревших комментариев. Кроме этого, текущие лучшие результаты в области демонстрируют качество на уровне значения метрики *F1-score* 0.8 [12], что недостаточно для широкого практического использования. Важной задачей на пути к улучшению качества моделей машинного обучения является сбор качественных данных. В частности, чтобы улучшить качество модели Deep JIT [12], имеет смысл обучить модель на большем количестве данных с меньшим уровнем шума. В задаче обнаружения устаревших комментариев для сбора датасетов используется автоматическая разметка. Она позволяет собрать большое количество данных за короткое время, в то время как ручная разметка, напротив, требует значительного времени и человеческих ресурсов. Инструмент **HeadlessCommentUpdater**, реализованный как часть данной дипломной работы, предлагает решение для автоматического сбора и разметки примеров для задачи обнаружения устаревших комментариев. Инструмент использует новый подход к автоматической разметке примеров устаревших комментариев описанный в этой главе, разработанный для борьбы с проблемой шума в разметке. Кроме этого, в отличие от ранее разработанных инструментов, **HeadlessCommentUpdater** также собирает дополнительную информацию о каждом примере, например произведенные рефакторинги.

2.1 Подходы к разметке

Данные для задачи обнаружения устаревших комментариев в постановке *JIT* представляют собой тройки из комментария, кода до изменений и кода после изменений. Нужно определить, становится ли комментарий устаревшим после изменений кода.

2.1.1 Классический подход

В большом числе работ, описывающих сбор и автоматическую разметку данных для задачи обнаружения устаревших комментариев, используется один и тот же подход [12, 14]. В этих работах, исследователи собирают данные из истории изменений проектов, используя систему контроля версий (VCS - Version Control System).

Разберем этот подход подробнее. В постановке *JIT*, примеры для задачи обнаружения устаревших комментариев состоят из изменений методов и соответствующих комментариев. Рассматривая коммит с изменением метода с комментарием, введем нотацию:

- D_1 – комментарий метода до коммита
- D_2 – комментарий метода после коммита
- M_1 – код метода до коммита
- M_2 – код метода после коммита

Возможно, что комментарий не изменился в коммите, тогда $D_1 = D_2$. Также, если код метода не изменился в коммите, $M_1 = M_2$. Таким образом, коммит можно рассматривать как преобразование $(D_1, M_1) \rightarrow (D_2, M_2)$.

На Рисунке 6 представлены обозначения изменений методов. Круги обозначают комментарии, квадраты – методы, которым эти комментарии соответствуют. Если в коммите $D_1 = D_2$, круг закрашивается белым. Если $D_1 \neq D_2$, круг закрашивается розовым. Аналогично, если $M_1 = M_2$, квадрат не закрашивается. В случае если $M_1 \neq M_2$, квадрат закрашивается зеленым. Так, в первом примере на Рисунке 6 не изменился ни метод, ни комментарий. Во втором изменился только метод, в третьем напротив, только комментарий.

В работах Пантаплакель и др. и Лиу и др. [12, 14] авторы используют одинаковый подход для разметки данных. На Рисунке 7 слева изображено изменение метода, в котором $D_1 = D_2, M_1 \neq M_2$, в этом случае тройка $< D_1, M_1, M_2 >$ обозначается как пример не устаревшего комментария для

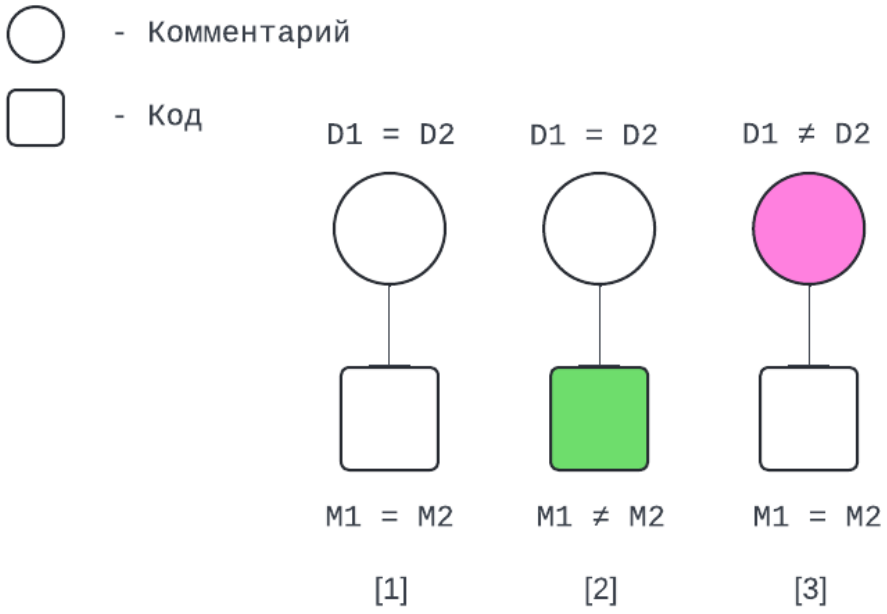

Рис. 6: Обозначение изменений методов

данных изменений кода. Действительно, разработчик не поменял комментарий D_1 после внесения изменений в код M_1, и можно предположить, что комментарий остался валидным.

Справа на Рисунке 7 изображено изменение метода, в котором $D_1 \neq D_2, M_1 \neq M_2$. В этом случае тройка $< D_1, M_1, M_2 >$ обозначается как пример устаревшего комментария для данных изменений кода. Разработчик изменил комментарий D_1 после изменения M_1, значит, комментарий D_1 не переживает изменений $M_1 \rightarrow M_2$.

2.1.2 Проблемы классического подхода

Подход, описанный выше, позволяет автоматически разметить примеры для задачи обнаружения устаревших комментариев, используя историю изменений проекта. Данный подход использует несколько сильных предположений, которые зачастую не выполняются.

- Размечая устаревшие примеры, предполагается, что изменение $D_1 \rightarrow D_2$ соотносится с изменением $M_1 \rightarrow M_2$, а это выполняется не всегда. Комментарий мог измениться не из-за изменения M_1. Например, изменение комментария может относиться к изменению кода несколько

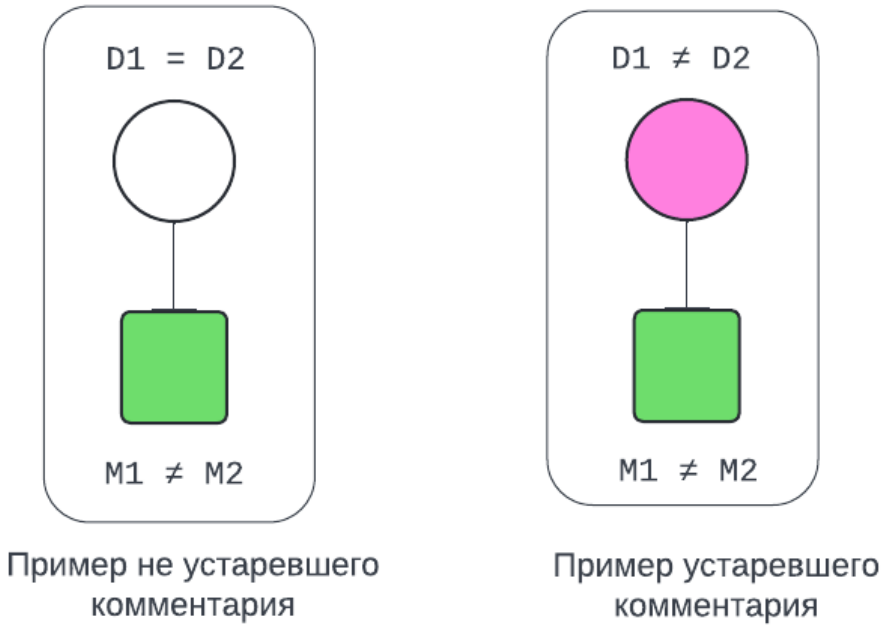

Рис. 7: Классический подход автоматической разметки

коммитов назад, но по каким-то причинам происходит только в текущем коммите. Кроме этого, изменение комментария может быть не связано с кодом метода, например, исправление опечатки. Тогда изменения $M_1 \to M_2$ не содержат информацию, которую алгоритм обнаружения устаревших комментариев может использовать, чтобы посчитать комментарий D_1 устаревшим.

- Размечая не устаревшие примеры, предполагается, что разработчик не забудет обновить комментарий в следствии изменения $M_1 \to M_2$, и сделает это в этом же коммите. Таким образом, собирая разметку для задачи обнаружения устаревших комментариев, предполагается, что разработчики не оставляют устаревших комментариев.

2.1.3 Прыжки

Первая из предложенных в данной работе техник для устранения проблем классического подхода разметки называется *прыжки*. Вместо того чтобы извлекать устаревшие примеры из одного коммита, предлага-

ется задействовать сразу несколько последовательных коммитов. На Рисунке 8 вместо тройки $< D_{1f}, M_{1f}, M_{2f} >$ будем использовать тройку $< D_{1b}, M_{1b}, M_{2f} >$.

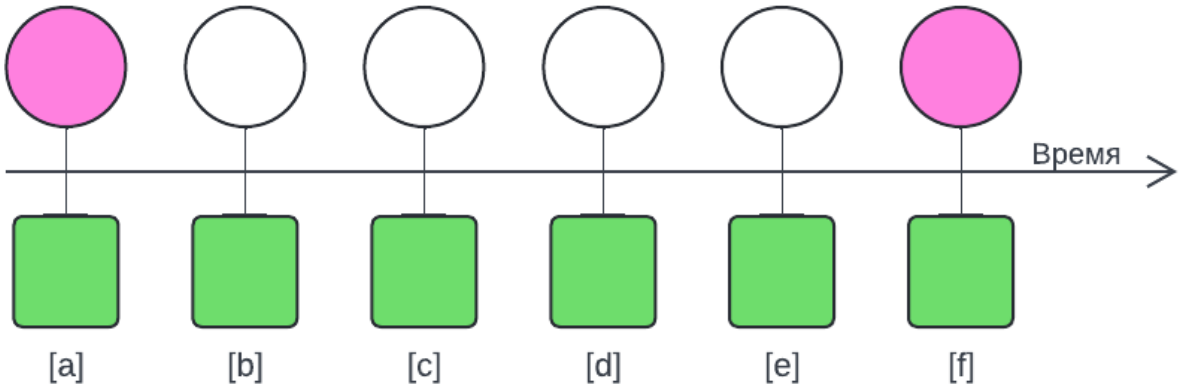

Рис. 8: Ветка изменений методов. Прыжок от коммита [b] к [f]

Таким образом, информация об изменениях метода собирается не только из коммита, когда изменен комментарий, но и из соседних коммитов, где изменялись только методы. Если в коммите f нет информации из-за которой изменился комментарий D_{1f}, возможно, что она присутствует в коммитах $[b - e]$. Длина прыжков может быть любой: единственным ограничением является то, что в коммитах, через которые происходит прыжок, не изменялся комментарий.

2.1.4 Испорченные не устаревшие примеры

На Рисунке 9 изображена ветка изменений методов, в которой в коммите f изменяется только комментарий, $M_{1f} = M_{2f}$. На такие примеры стоит обратить особое внимание, поскольку разработчик совершил коммит с изменением только в комментарии, но не в коде. Возможно, это связано с исправлением устаревшего ранее комментария. В таком случае примеры $[b - e]$ могут содержать устаревший комментарий.

Классический подход разметит примеры $[b - e]$ как примеры не устаревших комментариев. Альтернативным решением будет убрать примеры $[b - e]$ из рассмотрения и не использовать их в наборе данных, потому что

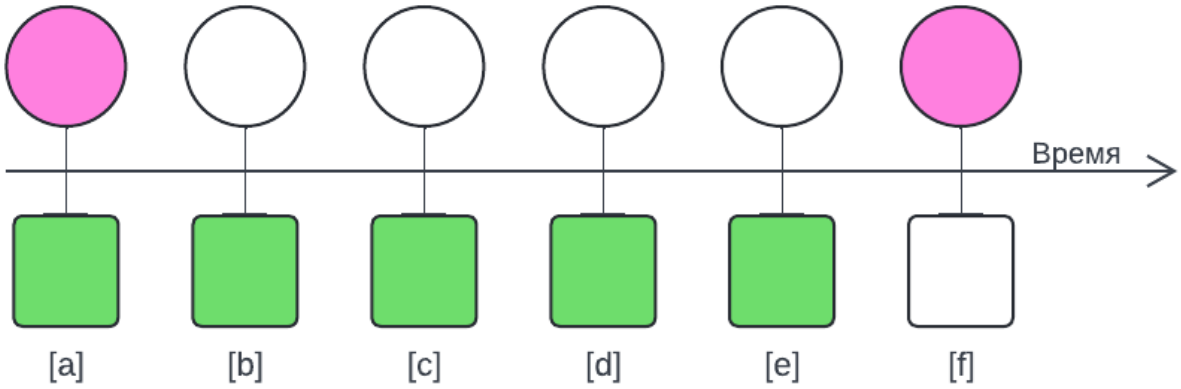

Рис. 9: Ветка изменений методов. Испорченные примеры [b-e]

есть риск наличия устаревшего комментария в одном из этих коммитов. Такие примеры $[b - e]$ назовем испорченными примерами.

2.1.5 Незначительные изменения комментариев

Стоит также отметить ненадежность определения устаревших примеров при незначительных изменениях комментария. Классический подход разметит любые изменения вида $D_1 \neq D_2, M_1 \neq M_2$ как пример устаревшего комментария. Однако если D_2 отличается от D_1 незначительно, например, исправлением опечатки, подбором синонима, добавлением запятой или переформулировкой предложения, такая разметка добавит в набор данных ложноположительный пример. Возможным решением является использование алгоритма и эвристик для детектирования переформулировок и исправления опечаток в подобных примерах и удаление примеров из набора данных.

2.2 Реализация

Альтернативный подход к сбору данных, использующий прыжки и фильтрацию испорченных и ненадежных примеров, реализован в инструменте **HeadlessCommentUpdater**. Этот инструмент реализован как плагин на платформе *IntelliJ Platform* [35] без графического интерфейса (также называемый headless plugin).

IntelliJ Platform предоставляет функционал для написания инстру-

ментов анализа кода без запуска пользовательского интерфейса. Такие плагины без интерфейса могут быть запущены из консоли. При этом плагины без интерфейса могут использовать все технические возможности платформы IntelliJ Platform.

Инструмент **HeadlessCommentUpdater** разделен на две части – Headless Plugin и PostProcessing. Headless Plugin собирает данные из репозиториев с кодом, используя встроенные возможности IntelliJ IDEA для взаимодействия с историей проекта. PostProcessing – набор скриптов для разметки собранных данных, который восстанавливает ветки изменений методов, обнаруживает прыжки и размечает примеры.

2.2.1 Headless Plugin

На Рисунке 10 представлена схема работы Headless Plugin. Плагин обходит репозитории с кодом проектов (см рис. 10 [1]) и для каждого проекта получает доступ к системе контроля версий и извлекает коммиты с помощью `GitRepositoryManager` (см. рис. 10 [2]). Множество коммитов обрабатывается независимо и асинхронно с помощью механизма корутин языка Kotlin [36]. Каждый коммит (см рис. 10 [3]) содержит информацию об измененных файлах (см рис. 10 [4]). Из измененных файлов извлекаются рефакторинги (см рис. 10 [6]) с помощью класса `RefactoringExtractor` (см рис. 10 [5]). `RefactoringExtractor` запускает поиск совершенных рефакторингов в изменениях кода с помощью библиотеки RefactoringMiner [37].

Рефакторинг – процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы [38]. RefactoringMiner [37] позволяет обнаружить совершенные рефакторинги, которые могут быть полезны для задачи обнаружения устаревших комментариев. Например, переименование методов, переменных или классов, перенос методов из одного класса в другой и так далее.

`ProjectMethodExtractor` (см рис. 10 [7]) использует изменения файлов и рефакторинги, чтобы сопоставить методы до изменений коммита и после по сигнатурам. Поскольку название метода и набор аргументов могли

измениться, для точного сопоставления версий методов необходима информация об изменениях сигнатуры функции. Например, полное имя метода из класса `A` с именем `function` и аргументами `(Boolean, Integer)` выглядит так: `A.function[Boolean, Integer]`. После двух рефакторингов изменилось имя метода `function -> method` и добавлен аргумент `add argument [Float]`, полное имя метода изменилось на `A.method[Boolean, Integer, Float]`. Именно эта функция из текущего файла должна быть сопоставлена функции `A.function[Boolean, Integer]` из старой версии файла. Без информации о совершенных рефакторингах, такое соответствие найти невозможно.

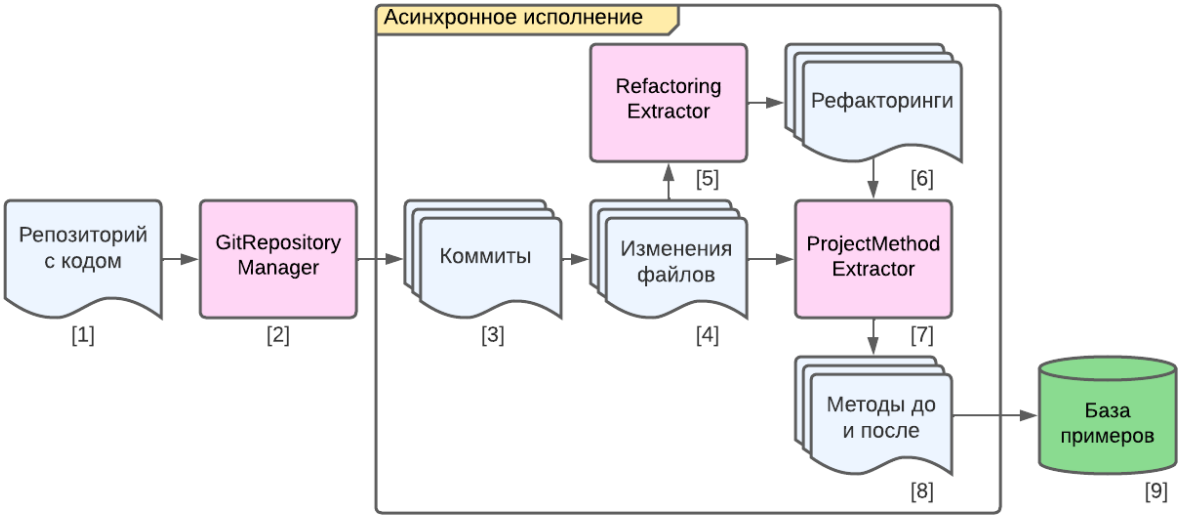

Рис. 10: Схема работы Headless Plugin

В результате плагин извлекает внутреннее представление измененных в коммите методов (см. рис. 10 [8]). Из этого представления извлекается следующая информация: код до и после изменений; комментарий до и после изменений; номер коммита; имя файла до и после изменений; время коммита; сигнатура метода до и после изменений. Эта информация записывается в хранилище (см рис. 10 [9]), в текущей версии плагина – в JSON [39] файл.

2.2.2 PostProcessing

После того как Headless Plugin извлек все измененные методы с комментариями из проектов, PostProcessing скрипт может приступать к автоматической разметке данных для задачи обнаружения устаревших комментариев. Вся информация о коммитах, ветках и файлах сохраняется плагином Headless Plugin вместе с кодом и комментариями. Благодаря этому PostProcessing может восстановить информацию об истории изменений каждого метода.

Для каждого метода PostProcessing скрипт определяет ветку изменений. Скрипт размечает примеры устаревших и не устаревших комментариев, руководствуясь техниками, описанными в прошлом подразделе 2.1.3: примеры устаревших комментариев формируются используя прыжки; примеры не устаревших комментариев добавляются только если ветка изменений метода не испорчена коммитом выше. Обрабатывая изменения методов в обратном хронологическом порядке, скрипт выделяет три случая.

- Код и комментарий метода в коммите изменились. Сформировать пример устаревшего комментария, используя прыжок от коммита непосредственно после текущего до ближайшего коммита с измененным кодом и комментарием. Также, если ветка была помечена испорченной, убрать пометку.
- Изменился только комментарий. Пометить ветку испорченной.
- Изменился только код. Если текущая ветка не испорчена, сформировать пример не устаревшего комментария, используя текущий коммит.

Помимо кода методов до и после изменений и комментариев, скрипт сохраняет дополнительную информацию о примерах. Полный список сохраняемых параметров представлен в таблице 2.

2.3 Результаты сбора

Для сбора данных было отобрано 3,875 проектов с открытым кодом на Java, размещенных в открытом доступе на сайте GitHub [40]. К проектам предъявлялись следующие требования: не менее 500 коммитов, более 10 контрибьюторов (работавших над проектом программистов), более 20 звезд, последний коммит не ранее первого августа 2019 года. Все проекты были обработаны с помощью инструмента **HeadlessCommentUpdater**. Процесс обработки занял 360 часов на вычислительной машине AWS [41] (сервис облачных вычислений), оснащенной 4 vCPU 3.1 GHz Intel Xeon Platinum 8175M.

Из 3,875 проектов только в 1,912 присутствуют JavaDoc комментарии к методам. Из этих 1,912 проектов было извлечено 2,736,073 примеров не устаревших комментариев и 91,392 примеров устаревших комментариев.

Примеры были отфильтрованы для уменьшения возможного шума в данных, некорректной разметки или слишком длинных примеров. Так, были удалены примеры:

- содержащие в комментарии дескрипторы @inheritDoc или @deprecated;
- с пустым телом метода;
- с комментариями, содержащими non-ascii символы;
- с числом токенов в коде или комментарии меньше 5 или больше 500;
- в которых комментарии до и после изменения отличаются только исправлением опечаток или переформулировкой (для обнаружения таких примеров использовалась модель *bert-base-cased-finetuned-mrpc*, обученная для похожей задачи, из библиотеки [42]).

После фильтрации осталось 45 тысяч примеров устаревших комментариев и один миллион примеров не устаревших комментариев. Для формирования набора данных с одинаковым числом устаревших и не устаревших примеров, из не устаревших примеров было случайно выбрано 45 тысяч примеров. Итого: 45 тысяч устаревших и 45 тысяч не устаревших комментариев.

2.4 Выводы

В данной главе описан инструмент **HeadlessCommentUpdater** для сбора и разметки примеров для задачи обнаружения устаревших комментариев и набор данных, собранный с его помощью. Инструмент реализован как плагин без графического интерфейса, разработанный на базе платформы IntelliJ Platform. Используя внутренние инструменты платформы, плагин обходит историю изменений проектов и собирает измененные методы с соответствующими комментариями. После этого плагин восстанавливает историю изменений для каждого метода и размечает примеры для задачи обнаружения устаревших комментариев. Разработанный плагин размещен в открытом доступе [43].

С помощью инструмента **HeadlessCommentUpdater** был собран значительный набор данных для задачи обнаружения устаревших комментариев. На текущий момент, большее число примеров было собрано только в работе Лиу и др. [14]. При этом, набор данных Лиу и др. извлечен из 1,500 проектов, в то время как набор, описанный в этой работе, задействует 3,875 проектов. Кроме этого, это первый набор данных, содержащий подробную информацию о произведенных рефакторингах в процессе изменения кода. Также при сборе данных были использованы новые техники для автоматической разметки примеров, призванные уменьшить процент некорректно размеченных данных.

Наименование	**Описание**
isRenamed	Произошел рефакторинг переименования метода
isParamRemoved	Один из параметров метода удален
isReturnType Changed	Возвращаемый тип метода изменен
isParamType Changed	Изменен тип одного из параметров
isParamRenamed	Один из параметров метода переименован
oldCodeLen	Длина последовательности токенов кода до изменений
newCodeLen	Длина последовательности токенов кода после изменений
commentLen	Длина последовательности токенов комментария до изменений
changedCodeLen	Число измененных токенов в коде
percentageCode Changed	Процент измененных токенов в коде
deleteComment IntersectionLen	Количество токенов комментария, который лежат в множестве удаленных из кода токенов
percentage Comment IntersectionDelete	Процент токенов комментария, которые лежат в множестве удаленных из кода токенов
oldCode CommentSim	Расстояние между векторными представлениями токенов кода до изменений и токенов комментария
newCode CommentSim	Расстояние между векторными представлениями токенов кода после изменений и токенов комментария
newOldCode CommentSim Distance	Абсолютное значение разности двух предыдущих значений
oldChanged CommentSim	Расстояние между векторными представлениями удаленных токенов кода и токенов комментария
newChanged CommentSim	Расстояние между векторными представлениями добавленных токенов кода и токенов комментария
oldNewChanged SimDist	Абсолютное значение разности двух предыдущих значений
added StatementSize	Число токенов, добавленных к коду
deleted StatementSize	Число токенов, удаленных из кода

Таблица 2: Дополнительная информация о примерах

Глава 3. Ручная разметка данных

В главе 2.1.2 описаны проблемы автоматической разметки данных. Несмотря на эвристики, использованные в инструменте **HeadlessCommentUpdater** для устранения некорректной разметки, проблема остается нерешенной. Чтобы оценить и исправить ошибки автоматической разметки, в рамках данной работы была также использована ручная разметка.

3.1 “Золотой” набор данных

В рамках данной работы был выделен и вручную кластеризован *золотой набор данных* – случайная тысяча примеров (500 примеров устаревших и 500 примеров не устаревших комментариев). Золотой набор данных можно использовать как для детальной оценки качества моделей, так и для анализа собранных данных.

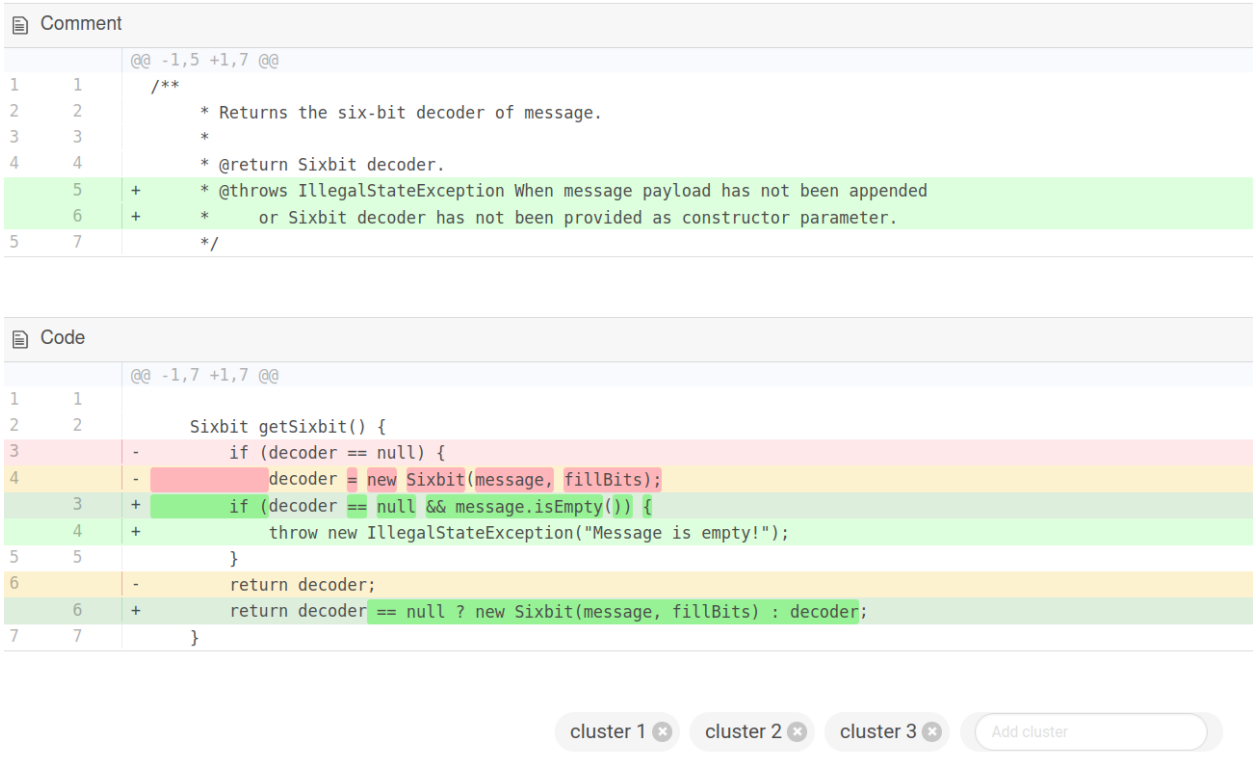

Рис. 11: Интерфейс приложения ручной разметки

Для кластеризации был создан специальный веб-интерфейс 11. В процессе разметки с использованием интерфейса пользователь видит измене-

ния кода и комментария и должен выбрать кластер для примера. Формулировка названий кластеров и ручная разметка золотого набора данных произведена автором данной работы. Полученная кластеризация описана в двух следующих секциях.

3.1.1 Кластеризация устаревших комментариев

На Рисунке 12 представлено разделение 500 устаревших примеров на кластеры.

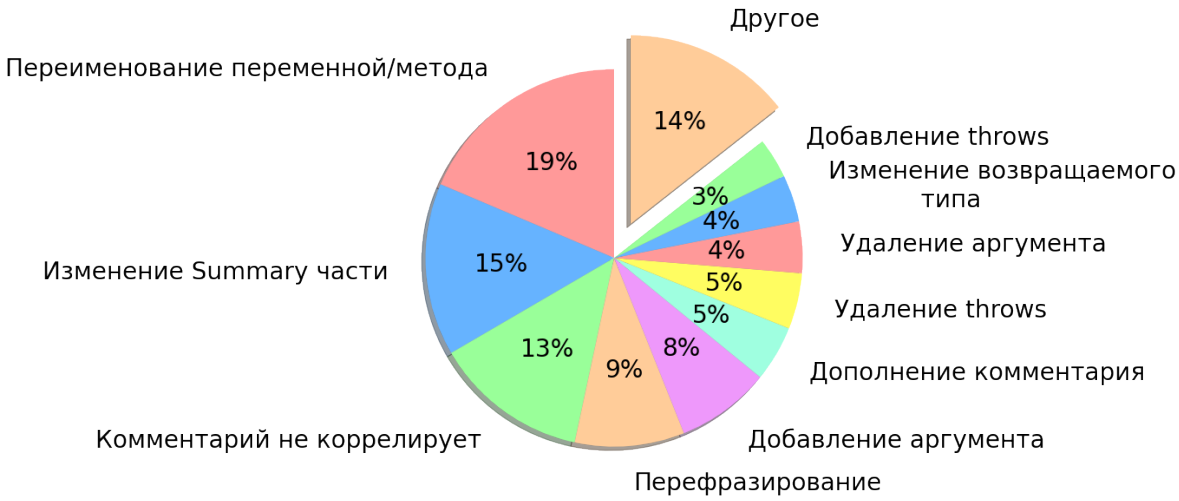

Рис. 12: Примеры устаревших комментариев топ 11 кластеров

Кластеры “Переименование переменной/метода”, “Добавление аргумента”, “Удаление аргумента”, “Добавление throws”, “Удаление throws”, “Изменение возвращаемого типа” соответствуют примерам, в которых комментарий должен измениться в результате соответствующего рефакторинга метода. Например, кластер “Добавление аргумента” содержит примеры, в которых в метод добавился параметр, значит, в JavaDoc комментарий должен добавиться дескриптор @param, описывающий новый аргумент. Кластер “Изменение Summary части” соответствует примерам, в которых меняется Summary часть комментария из-за изменений в логике кода. Перечисленные кластеры – валидные примеры устаревших комментариев, автоматическая разметка для них верна.

Кластер “Комментарий не коррелирует” соответствует примерам, в которых обновление комментария не связано с обновлением метода. Незначительное изменение смысла комментария, исправление опечаток или дополнение сторонней информацией соответствует примерам из кластеров “Перефразирование” и “Дополнение комментария”. Упомянутые три кластера содержат невалидные примеры устаревших комментариев, их не следует использовать для обучения моделей. Примечательно, что в сумме эти невалидные кластеры составляют **27%** всех примеров устаревших комментариев.

Кластер “Другое” содержит несколько небольших подкластеров: 3% примеров, в которых часть кода заменяется вызовами сторонних функций, 3% примеров с комментариями, не содержащими никакой информации о коде, менее 3% примеров содержат обновление комментариев, с целью устранения ранее созданных устаревших комментариев. Другие 5% содержат множество различных кластеров размера менее процента каждый: изменение аннотаций, изменение типа параметров, удаление аннотаций и т.д.

3.1.2 Кластеризация не устаревших комментариев

На Рисунке 13 представлена кластеризация 500 примеров не устаревших комментариев.

Большая часть примеров относится к кластеру “Сохраняющие функциональность” – корректные примеры не устаревших комментариев, в которых происходят изменения логики кода, не требующие изменений комментария. Кластер “Изменение синтаксиса” содержит примеры, в которых изменения кода состоят из незначительных синтаксических изменений, например, замена `for` на `while`. Кластер “Добавление аннотации” состоит из изменений кода, добавляющих аннотации. Примеры, в которых в коде изменяется исключительно вызов внешней функции или использование класса, содержатся в кластере “Изменение внешнего кода”. Примеры из четырех вышеперечисленных кластеров не требуют изменения комментариев, значит, автоматическая разметка для них валидна.

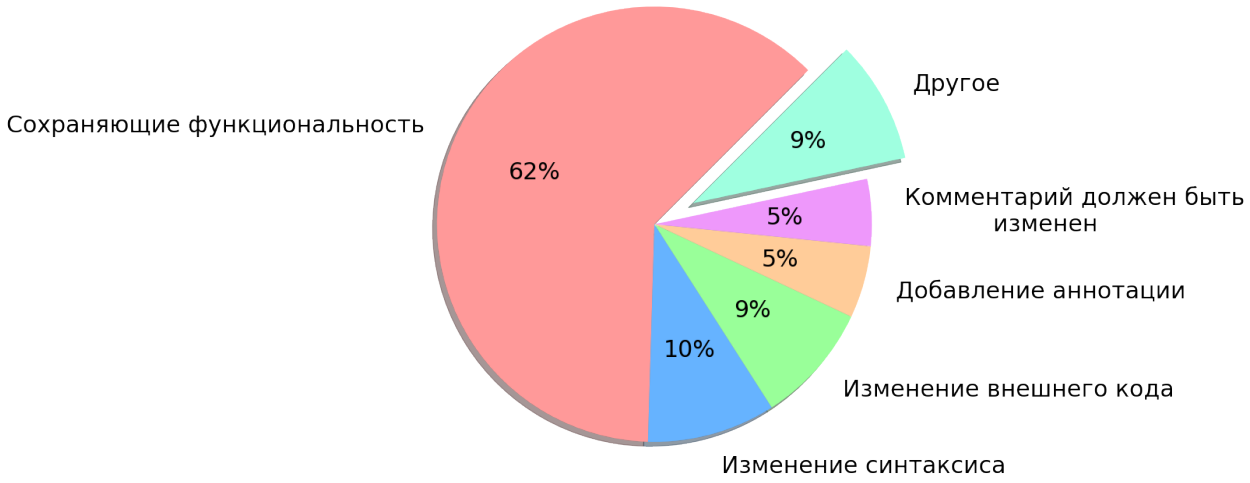

Рис. 13: Примеры не устаревших комментариев топ 6 кластеров

Кластер “Комментарий должен быть изменен” содержит примеры, в которых изменения кода должны спровоцировать изменение комментария, но в действительности комментарий не изменился. Такие примеры размечены неправильно. В кластере “Другое” содержится 4.6% примеров с комментариями низкого качества, не относящимся к коду. Другие 3.2% относятся к примерам, в которых произошло переименование, не требующее изменения комментария. Еще несколько кластеров размером менее 1% отвечают за примеры с пустым телом метода, удалением throws и т.д.

3.2 Модель Deep JIT на новых данных

Размеченный вручную золотой набор данных позволяет детально анализировать качество моделей. Модель Deep JIT [12] была обучена на данных, собранных в рамках этой работы 2.3. Из этого набора 1000 примеров была случайно выбрана для золотого набора данных. При обучении модели задаче обнаружения устаревших комментариев, важно следить за источником примеров: примеры из одного проекта не должны содержаться одновременно в обучающей и в тестовой выборках. Для этого все примеры из проектов золотого набора данных были отброшены. Оставшиеся примеры разделены на тренировочную, валидационную и тестовую подвыборки, без пересечений по проектам. Размеры итоговых подвыборок: 47,116 тре-

нировочных, 16,950 валидационных и 17,342 тестовых примеров. Далее эти подвыборки для обучения будем называть **CU** (**C**omment**U**pdater). Для сравнения, в работе Пантаплакель и др. [12] набор данных состоял из 32,988 тренировочных, 3,756 валидационных и 3,944 тестовых примеров. Эти данные далее будем называть **JITData**. Модель Deep JIT [12] была обучена на данных из этой работы и работы Пантаплакель и др. Для сравнения моделей было измерено качество на золотом наборе данных, результаты представлены в таблице 3. Подробное описание метрик классификации можно найти в приложении 5.3.

Данные для обучения	**Кластеры**	**Precision**	**Recall**	**F1**	**Acc**
CU	Все	0.62	**0.87**	0.72	0.76
JITData	Все	0.64	0.70	0.67	0.68
CU	Валидные	0.71	0.86	**0.78**	**0.83**
JITData	Валидные	**0.75**	0.66	0.70	0.73

Таблица 3: Сравнение качества модели Deep JIT [12] обученной на разных данных

Как было описано ранее, не все кластеры из золотого набора данных содержат валидные примеры. Точность модели Deep JIT, обученной как на CU данных, так и на JITData, выше для валидных кластеров 3. Кроме этого, точность и F1-score выше у модели, обученной на CU данных.

3.3 Выводы

Разметив 1000 случайно выбранных примеров, можно сделать предположения о разбиении всего массива данных на кластеры. Главный вывод – суммарно среди устаревших и не устаревших комментариев 27%+5%=**32%** неверно размеченных примеров. Несмотря на эвристики для уменьшения числа ошибок в автоматической разметке, около трети примеров размечено неверно. Если такой процент неверно размеченных примеров сохраняется для всего набора данных, а не только для золотой подвыборки, трудно ожидать высокого качества решения задачи моделями машинного обучения.

Помимо этого можно сделать выводы о распределении правильно раз-

меченных примеров. Значительная доля примеров устаревших комментариев относится к кластеру определенного рефакторинга **43%**. В примерах не устаревших комментариев существенная часть 19% относится к внешним или незначительным изменениям. Эти изменения не относятся к изменению логики кода. Эти наблюдения можно использовать в дальнейшем построении моделей для задачи обнаружения устаревших комментариев. Например, есть основания полагать, что результативным будет использование информации о произведенных рефакторингах в качестве части входа для модели.

Глава 4. Краудсорсинг

Результаты ручной кластеризации золотого набора данных показывают, что среди автоматически собранных примеров много некорректно размеченных. Чтобы бороться с некорректной автоматической разметкой, часть примеров можно переразметить вручную. Приглашая больше пользователей, потенциально возможно разметить большее число примеров для задачи обнаружения устаревших комментариев. Для привлечения к разметке сторонних экспертов в этой работе используется краудсорсинг. *Краудсорсинг* — способ организации процессов, при котором большая задача разбивается на большое количество маленьких подзадач, которые передаются исполнителям.

4.1 Яндекс.Толока

В данной работе краудсорсинг используется для разметки примеров задачи обнаружения устаревших комментариев с помощью сервиса Яндекс.Толока [13].

Толокерам (пользователям Яндекс.Толоки) предлагается за определенную плату ответить на вопросы о примере для задачи обнаружения устаревших комментариев. Так как не все Толокеры умеют программировать и обладают достаточным знанием Java для разметки данных, для улучшения качества разметки предприняты следующие шаги.

Обучение – прежде чем перейти к разметке, Толокеры должны решить несколько обучающих заданий. Для создания обучения были вручную размечены примеры и написаны подсказки к ним. В процессе обучения Толокеру нельзя перейти к следующему заданию, не ответив верно на текущее. При ошибке Толокеру высвечивается подсказка для решения текущего задания. Чтобы перейти к основным заданиям проекта, Толокер должен с первой попытки верно ответить на определенный процент заданий из обучения (около 75% в этой дипломной работе).

Контрольные задания – вместе с основными заданиями Толокер отвечает на контрольные задания. Контрольные задания размечаются авторами проекта и перемешиваются с основными. Толокер не знает, какое за-

дание он размечает, контрольное или основное. Качество на контрольных заданиях измеряется для каждого Толокера. При качестве разметки на контрольных заданиях ниже определенного порога, Толокер на время удаляется из проекта. Авторам проекта необязательно размечать много контрольных заданий, достаточно разметить на порядок меньше контрольных заданий, чем основных, для осуществления контроля качества.

Перекрытие – каждое задание размечается не одним Толокером, а несколькими одновременно. Таким образом, итоговый ответ для каждого задания получается после применения алгоритма агрегации ответов нескольких Толокеров. Такой метод увеличивает устойчивость разметки к некорректным ответам. В данной дипломной работе использовалось перекрытие от 5 до 10 в разных конфигурациях проекта.

В процессе данного исследования 1000 примеров была размечена на Яндекс.Толоке. Как показали эксперименты, процесс разметки примеров для такой специфичной задачи на Яндекс.Толоке занимает значительное время. Опыт использования механизма краудсорсинга в этой работе будет полезен в последующих, более масштабных экспериментах по разметке данных для задачи обнаружения устаревших комментариев. Примеры для разметки на Яндекс.Толоки были взяты из собранного в данном исследовании набора данных на основе предсказаний DeepJIT. После обучения модели Deep JIT [12] на этом наборе данных (**CU** 3.2) была выбрана тысяча примеров не устаревших комментариев, для которых ответ модели сильнее всего отличается от автоматической разметки.

Для упрощения интерфейса разметки все примеры разделены на три категории: Summary, Param, Return.

4.1.1 Return

Return – примеры из 1000, которые содержат дескриптор @return в комментарии и в коде изменяется тип возвращаемого значения или меняются инструкции с ключевым слово return. Для разметки на Толоке в комментарии оставлена только Return часть (см. Рисунок 3). Цель разметки Return примеров – разделить примеры, в которых Return часть коммента-

рия должна измениться после рефакторинга возвращаемого значения, от примеров, в которых изменения комментария не требуются.

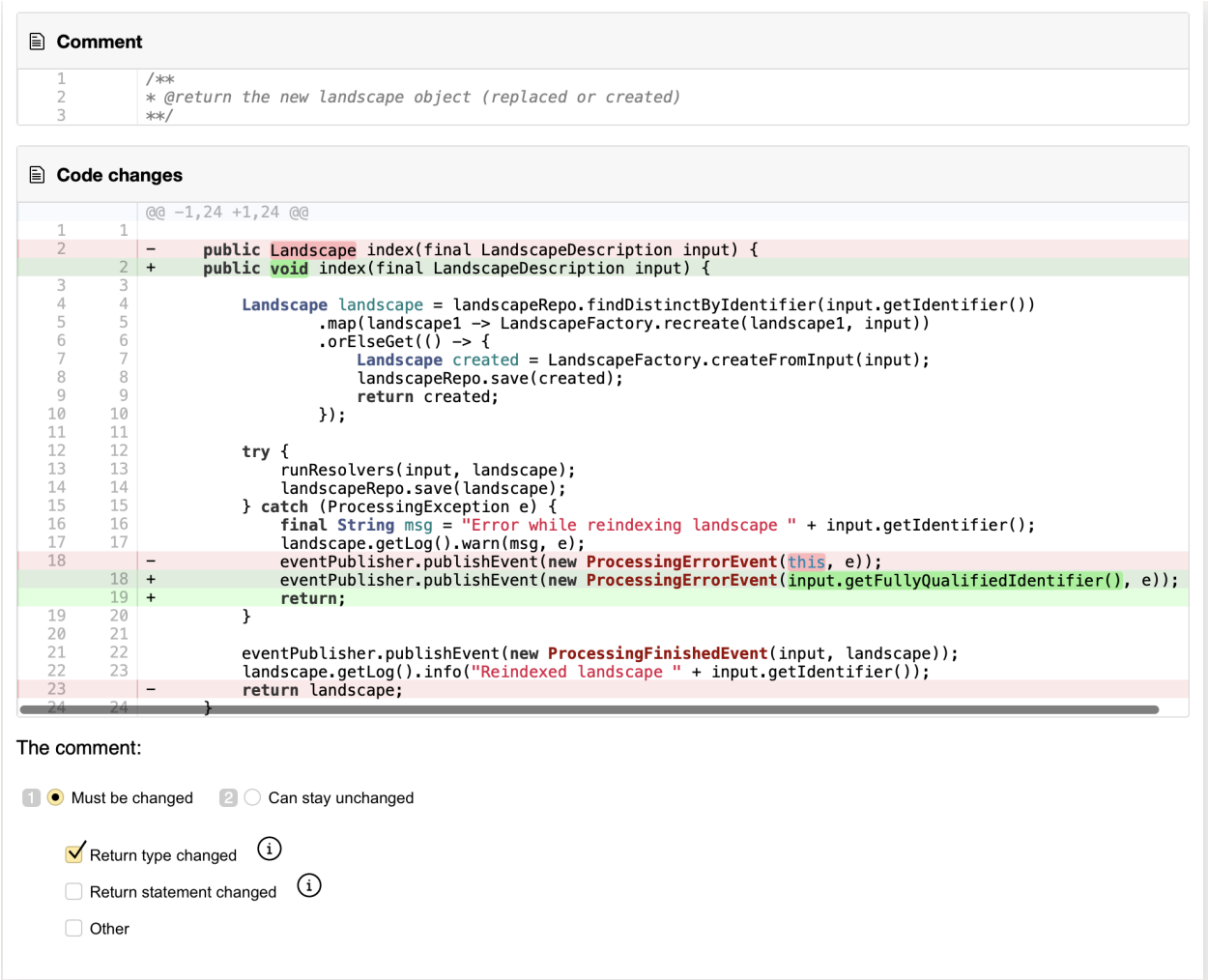

Рис. 14: Пример интерфейса Return задания

На Рисунке 14 представлен пример интерфейса Return заданий. Главный вопрос, который есть во всех трех категориях примеров – *"Должен ли комментарий измениться?"*. Ответив на этот вопрос, Толокер может указать подробную причину. В примере на Рисунке в методе изменяется тип возвращаемого значения с `Landscape` на `void`. При этом комментарий все еще описывает возвращаемый тип как “the new landscape object”. По этим причинам нужно выбрать следующие опции:

- “Must be changed” – комментарий должен быть изменен;
- Причина “Return type changed” – возвращаемый тип изменен.

Опция	Значение
Return type changed	Возвращаемый тип изменен таким образом, что текущее описание @return больше не консистентно.
Return statement changed	Только return инструкции изменены, не возвращаемый тип, но текущее @return описание больше не консистентно.
Changes don't affect return block	Ни возвращаемый тип, ни инструкции return не изменяются в коде
@return remains consistent	В коде изменяется возвращаемый тип или инструкции return, но текущий @return комментарий остается консистентным.

Рис. 15: Опции для разметки Return части комментария

Подробное описание всех опций представлено в таблице 15.

Всего таких примеров было выбрано 301, размечено с перекрытием 5. Средняя точность на контрольных заданиях после агрегации 0.89. Из 301 примера, изначально размеченных как не устаревшие комментарии, по итогам разметки Толокерами, в **99** примерах нужно изменить разметку.

4.1.2 Param

Param – примеры из 1000, которые содержат дескриптор @param в комментарии и в коде изменяется список аргументов метода. Для разметки на Толоке в комментарии также оставлена только Param часть (см 3). Цель разметки Param примеров – также разделить примеры, в которых Param часть комментария должна измениться после рефакторинга списка аргументов, от примеров, в которых изменения комментария не требуются.

На Рисунке 16 представлен пример интерфейса Param заданий.

После изменений в этом примере параметр `atoms` переименован в `txns`. При этом, комментарий все еще описывает параметр как “atoms atom

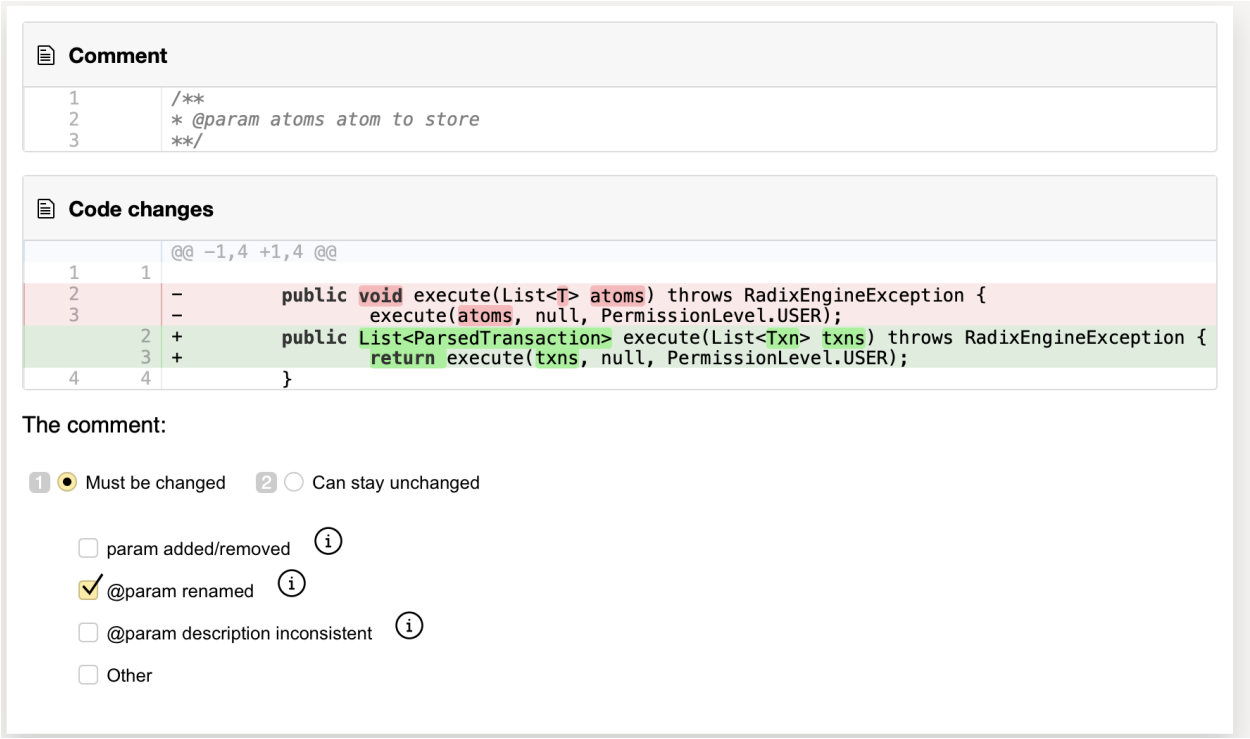

Рис. 16: Пример интерфейса Param задания

to store". По этим причинам нужно выбрать следующие опции:

- "Must be changed" – комментарий должен быть изменен;
- Причина "@param renamed" – параметр переименован.

Подробное описание всех опций представлено в таблице 17.

Всего таких примеров было выбрано 165, размечено с перекрытием 10. Средняя точность на контрольных заданиях после агрегации 0.88. Из 165 примеров, изначально размеченных как не устаревшие комментарии, по итогам разметки Толокерами, в **56** примерах нужно изменить разметку.

Опция	Значение
Param added/removed	Параметр добавлен или удален в изменениях кода.
@param renamed	Параметр переименован и должен быть переименован в комментарии.
@param description inconsistent	Комментарий @param больше не соотвествуют логике использования параметра в коде.
Changes don't affect param list	Изменения кода не влияют на список параметров.
@param remains consistent	Изменения кода влияют на список параметров, но текущее описание остается консистентным.

Рис. 17: Опции для разметки Param части комментария

4.1.3 Summary

Summary – примеры из 1000, которые содержат предложения без дескрипторов в начале комментария. Для разметки на Толоке в комментарии снова оставлена только Summary часть комментария (см 3).

На Рисунке 18 представлен пример интерфейса Summary заданий. До изменений кода, метод удалял существующий файл **или** создавал новый файл (строка 7). После изменений кода метод удаляет существующий файл **и** создает новый файл (строка 8). Однако комментарий не отражает этого изменения и все еще используют **or** вместо **and**. Поэтому следует выбрать следующие опции:

- “Must be changed” – комментарий должен быть изменен;
- Причина “Logic changed” – логика, описанная в Summary, изменилась в коде.

Рис. 18: Пример интерфейса Summary задания

Подробное описание всех опций представлено в таблице 19.

Всего таких примеров было выбрано 541, размечено с перекрытием 5. Средняя точность на контрольных заданиях после агрегации 0.88. Из 541 примера, изначально размеченных как не устаревшие комментарии, по итогам разметки Толокерами, в **42** примерах нужно изменить разметку.

4.2 Модель Deep JIT на данных после краудсорсинга

После изменений разметки части примеров пользователями Яндекс.Толоки были проведены эксперименты по обучению модели Deep JIT [12] на из-

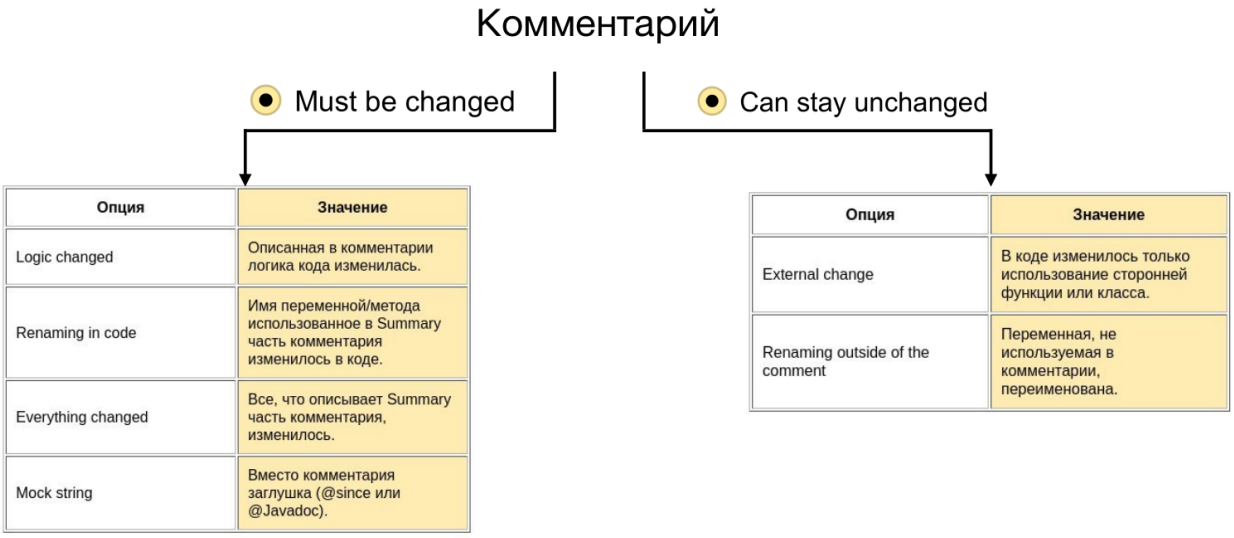

Рис. 19: Опции для разметки Summary части комментария

мененных данных. Пользователи Яндекс.Толоки вручную разметили 1000 примеров, этого не хватит для полноценного обучения глубокой нейронной сети. Однако, можно попробовать добавить размеченные вручную данные к тренировочному набору, размеченному автоматически. Результаты эксперимента описаны в таблице 4. Обучение производится так же, как описано в разделе 3.2. **CU** – набор тренировочных данных, собранный с помощью **HeadlessCommentUpdater**. **CU + Toloka** – те же тренировочные данные, но некоторые примеры переразмечены после Яндекс.Толоки.

Данные для обучения	**Кластеры**	**Precision**	**Recall**	**F1**	**Acc**
CU	Все	0.62	0.87	0.72	0.76
CU + Toloka	Все	0.57	0.89	0.69	0.75
CU	Валидные	0.71	0.86	0.78	0.83
CU + Toloka	Валидные	0.67	0.89	0.76	0.82

Таблица 4: Сравнение качества модели Deep JIT [12] обученной на данных до Толоки и после

Прироста качества не наблюдается. Это можно объяснить недостаточным числом переразмеченных данных. Увеличение объема размеченных вручную данных и более детальное обучение моделей на переразмеченных примерах – варианты для дальнейшего исследования в области обнаружения устаревших комментариев.

4.3 Выводы

В рамках данной работы разработана методология ручной разметки данных для задачи обнаружения устаревших комментариев, с использованием краудсорсинга. В результате нескольких недель разметки было размечено 1000 примеров, с учетом перекрытий выполнено 45,726 заданий, 4,334 пользователя Яндекс.Толоки попробовали сделать хотя бы одно задание на проекте. Интерфейс для разметки, инструкции для пользователей, обучающие примеры и методы агрегирования могут быть переиспользованы в дальнейших исследованиях. Учитывая агрегированные результаты, а также выводы из прошлой главы о золотом наборе данных, можно утверждать, что автоматическая разметка содержит значительный процент шума. Масштабный набор данных, очищенный от шума, – приоритетная цель для исследователей задачи обнаружения устаревших комментариев.

Глава 5. Инструмент CommentUpdater

В ходе данной работы был разработан плагин **CommentUpdater** для среды разработки IntelliJ IDEA как интерфейс для использования алгоритмов обнаружения устаревших комментариев. Каждый раз при изменении разработчиком кода, плагин запускает алгоритм обнаружения устаревших комментариев в измененном коде. Если алгоритм находит комментарий, который считает устаревшим, этот комментарий подсвечивается, сообщая разработчику о потенциальной ошибке в комментарии.

5.1 Использование

Рассмотрим пример использования плагина на простом примере. На первом шаге разработчик реализовал метод `simpleDemoFunction` с двумя аргументами `arg1` и `arg2` и написал комментарий к нему (Рисунок 20).

```
public class Demo {

    /**
     * Sophisticated calculations with arg1 and arg2
     */
    public int simpleDemoFunction(Integer arg1, Integer arg2) {
        return (arg1 * arg1 + arg2 * arg2);
    }
}
```

Рис. 20: Шаг 1 – функция `simpleDemoFunction` до изменений

На втором шаге разработчик решает изменить функцию и удалить аргумент `arg2` (Рисунок 21).

После удаления аргумента `arg2` комментарий становится устаревшим, так как он все еще ссылается на использование этого аргумента в методе. Плагин указывает на это, выделяя желтым цветом комментарий к методу (Рисунок 22). Таким образом в среде разработки IntelliJ IDEA обозначаются предупреждения о проблемах в коде.

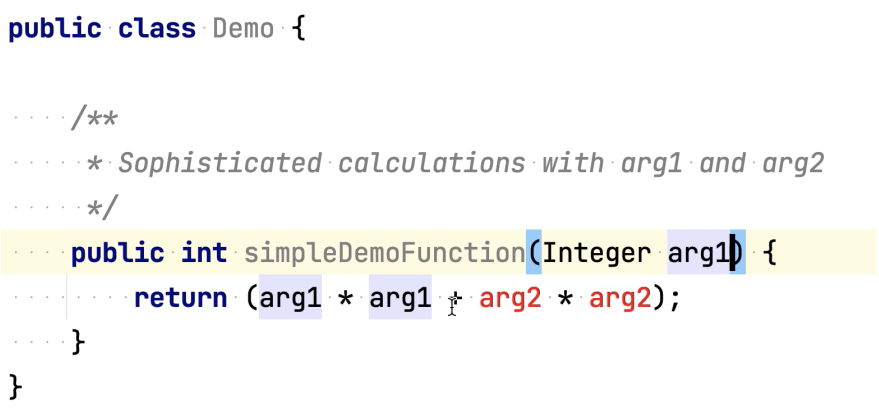

Рис. 21: Шаг 2 – из функции удаляется параметр

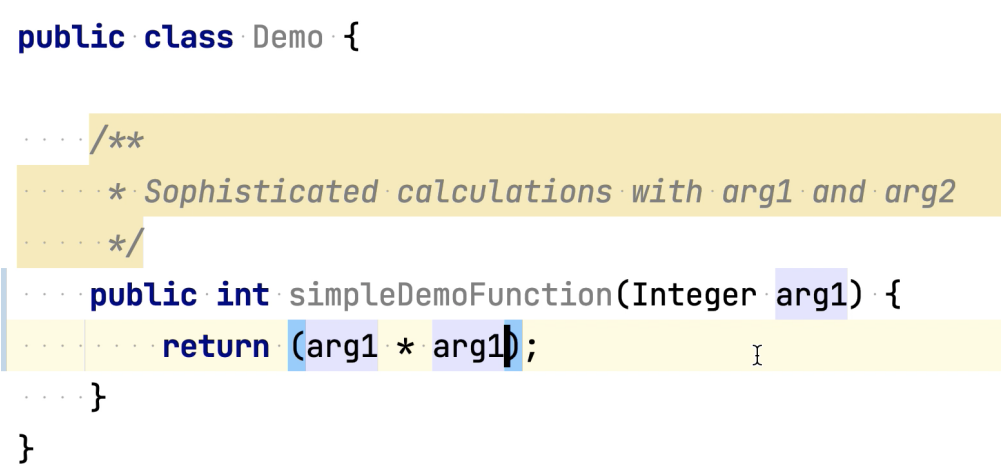

Рис. 22: Шаг 3 – появляется предупреждение, комментарий устарел

Предупреждение содержит информацию о природе обнаруженной проблемы (Рисунок 23). Пользователь может навести курсор на панель справа от кода и увидеть причину выделения: “Inconsistent comment found”.

После того как устаревшая информация удаляется из комментария, алгоритм по обнаружению устаревших комментариев запускается повторно. Поскольку проблема была устранена, плагин не покажет предупреждений 24.

5.2 Реализация

Плагин реализован на основе *IntelliJ Platform* – платформы для разработки интегрированных сред разработки (IDE). На основе этой платформы реализованы такие IDE как IntelliJ IDEA [35], PyCharm, CLion.

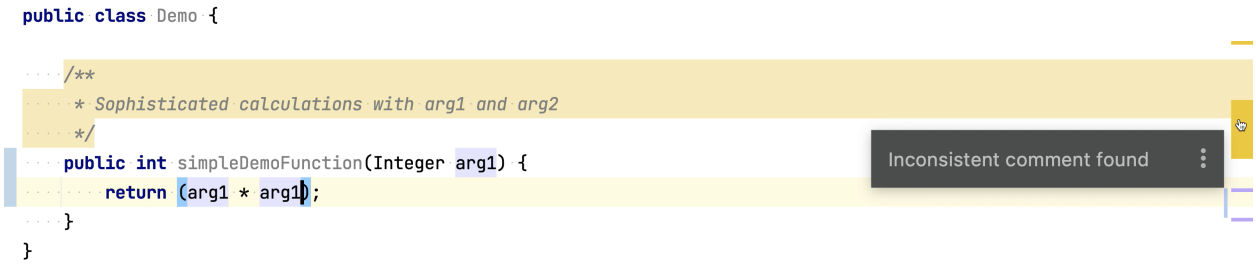

Рис. 23: Шаг 4 – подсказка предупреждения

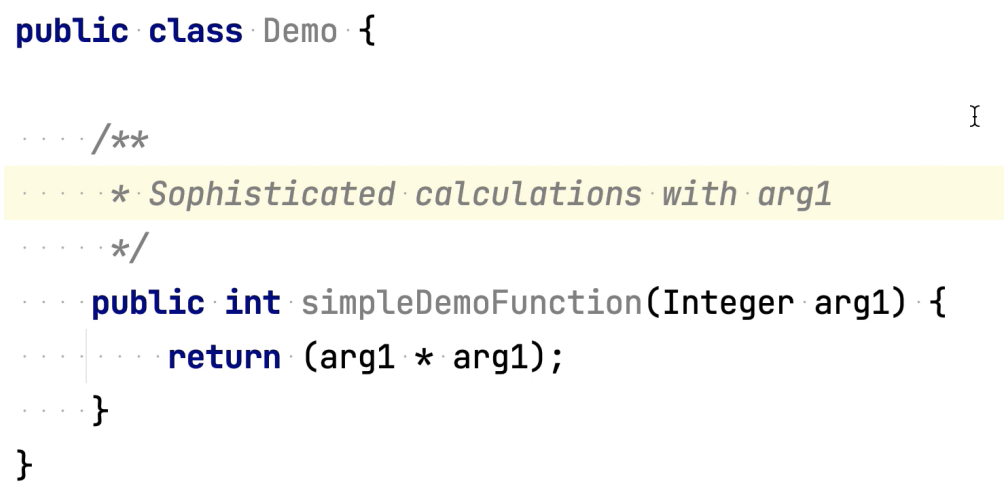

Рис. 24: Шаг 5 - комментарий исправлен, предупреждений нет

IntelliJ Platform предоставляет функционал для анализа кода до этапа компиляции – инспекции [44]. Такой анализ может обнаружить синтаксические ошибки, опечатки, вероятные места программных ошибок. Обнаруженные проблемные места в коде подсвечиваются или подчеркиваются. В плагине **CommentUpdater** основным механизмом взаимодействия с пользователем является инспекция, подсвечивающая устаревшие комментарии (пример подсветки представлен на Рисунке 22). Цель инспекции – обнаружить устаревшие комментарии и сообщить о них пользователю. Для этого инспекция проделывает следующие шаги: сравнивает текущую версию файла с версией, сохраненной в системе контроля версий, находит все пары из измененного метода с комментарием и оригинала метода из VCS и, наконец, запускает для всех таких пар алгоритм обнаружения устаревших комментариев.

Разберем механизм работы инспекции, следуя диаграмме 25. Инспекция срабатывает при каждом изменении файла на Java. Измененный файл

передается для обработки менеджеру изменений `ChangeListManager` и реализации шаблона посетитель [45], классу `JavaElementVisitor`.

`JavaElementVisitor` посещает все элементы внутреннего представления кода и обрабатывает указанным заранее методом. Так, чтобы извлечь из файла с кодом все задокументированные методы, достаточно переопределить функцию `visitDocComment`.

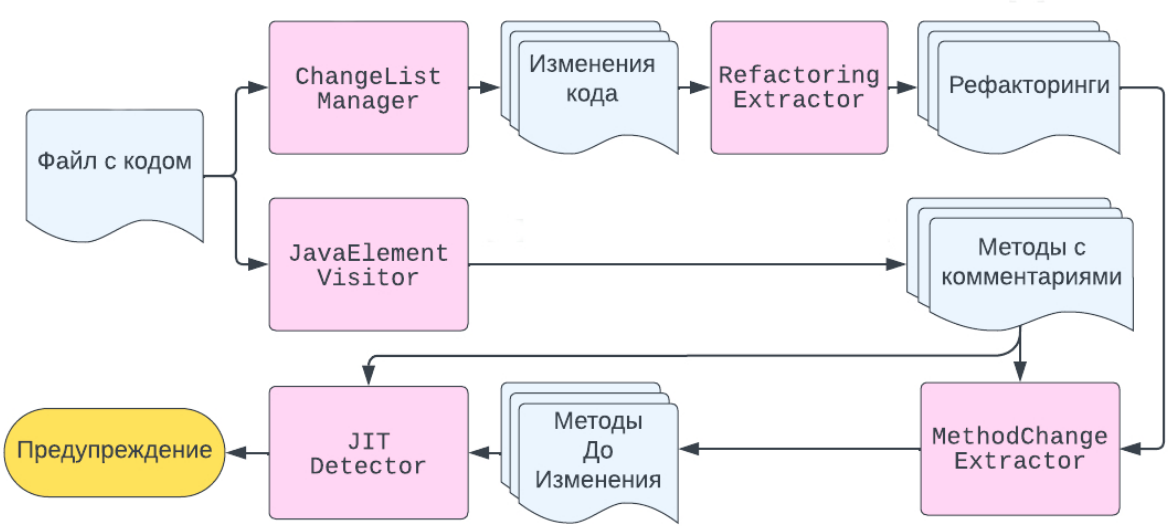

Рис. 25: Диаграмма инспекции плагина CommentUpdater

`ChangeListManager` – внутренний класс IntelliJ Platform, который использует VCS проекта, чтобы извлечь изменения кода в текущем файле. Далее `RefactoringExtractor`, как было описано ранее в разделе 2.2.1, использует библиотеку RefactoringMiner [37] для извлечения рефакторингов.

Далее информацию о совершенных рефакторингах использует `MethodChangeExtractor`, чтобы найти для текущих методов их прошлую версию, сохраненную в VCS. Механизм сопоставления методов из разных версий файла описан ранее в разделе 2.2.1.

Текущая и прошлая версии метода с комментарием передаются в `JITDetector`. Этот класс запускает модель глубокого обучения Deep JIT [12], предварительно преобразовав входные данные в нужный формат. Запуск модели Deep JIT производится с помощью библиотеки ONNXRuntime [46].

Это библиотека для кроссплатформенного запуска нейронных сетей. Чтобы воспользоваться библиотекой, PyTorch [47] модель Deep JIT была конвертирована в формат ONNX [48]. Также, для устранения неточностей при конвертации был использован инструмент netron [49].

Процесс подготовки данных для модели проиллюстрирован на Рисунке 26. Код до и после изменений извлекается и преобразуется в последовательность изменений. Пример последовательности изменений описан ранее на Рисунке 4. Комментарий преобразуется в последовательность токенов. Кроме этого, извлекаются дополнительные признаки кода и комментария, которые подаются модели Deep JIT. Признаки кода это последовательность булевых векторов размера 16, вектор составляется для каждого токена. Координаты вектора содержат информацию, которая потенциально может помочь восстановить контекст и важность текущего токена. Например, содержится ли токен в возвращаемом типе, был ли токен удален или добавлен, состоит ли токен только из букв, является ли токен ключевым словом языка Java, и так далее. Признаки комментария также представлены в виде булева вектора размера 16 для каждого токена комментария. Координаты содержат информацию о том, какую роль играет токен в последовательности изменений кода: был ли такой же токен удален из кода, есть ли такой же токен в возвращаемом значении метода, является ли токен частотным словом для английского языка, и так далее. В исследовании, описывающем модель Deep JIT [12], показано, что подобные признаки дают прирост качества модели.

Архитектура плагина **CommentUpdater** не ограничивает возможности извлечения подобных признаков из кода. Для сбора дополнительных признаков можно использовать всю внутреннюю информацию, предоставляемую IntelliJ Platform о текущем методе. Например, в дальнейших работах можно использовать информацию об обнаруженных в рамках изменения рефакторингах в качестве дополнительных признаков для модели. Также возможно использовать информацию о более ранних версиях того же метода, обратившись к модулю git4idea IntelliJ Platform, который отвечает за работу с системой контроля версий проекта. Использование этих и схожих техник для извлечения дополнительной информации – одно из воз-

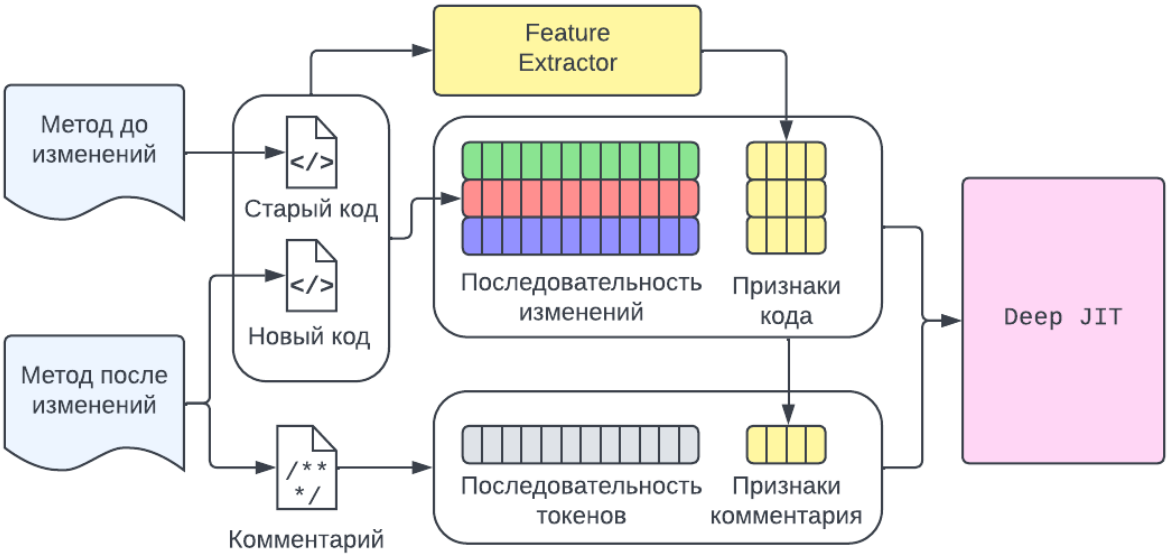

Рис. 26: Подготовка данных для Deep JIT

можных направлений последующего развития данной дипломной работы.

5.3 Выводы

В данной главе описан инструмент **CommentUpdater** для обнаружения устаревших комментариев в коде. Инструмент реализован как плагин для среды разработки IntelliJ IDEA. Используя систему контроля версий и внутренние возможности IntelliJ Platform, плагин извлекает информацию об измененных методах, запускает алгоритм обнаружения устаревших комментариев и оповещает пользователя, если были обнаружены устаревшие комментарии. Плагин **CommentUpdater** доступен в открытом доступе на GitHub [43].

Заключение

В рамках данной работы были достигнуты следующие результаты:

- Разработан инструмент для обнаружения устаревших комментариев в виде плагина для IntelliJ IDEA. Плагин использует механизм инспекций и IntelliJ Platform для извлечения информации из методов с комментариями, которая подается на вход глубокой нейронной сети, обученной для обнаружения устаревших комментариев. Устаревшие комментарии подсвечиваются в коде, обращая на себя внимание разработчика.

- С целью дальнейшего улучшения качества моделей был разработан инструмент для сбора и автоматической разметки примеров для задачи обнаружения устаревших комментариев. Инструмент также использует преимущества IntelliJ Platform и, помимо необходимой информации о методах с комментариями, собирает дополнительную (информацию о рефакторингах, метриках схожести для векторных представлений и т.д.).

- Собран набор данных, содержащий более двух миллионов примеров из 4000 проектов на Java. Автоматическая разметка примеров произведена новым способом, разработанным в данной работе для уменьшения числа некорректных примеров.

- Кластеризован золотой набор данных. Набор из 1000 примеров, содержащий детальную информацию о природе изменений и причинах возникновения устаревших комментариев. Используя золотой набор данных, было проанализировано качество модели, обученной на различных данных (авторских и предложенных ранее).

- Разработан метод ручной разметки данных для задачи обнаружения устаревших комментариев, использующий краудсорсинг. Для этого созданы интерфейс, инструкции, контрольные и обучающие задания. В результате эксперимента размечено 1000 примеров и задействовано несколько тысяч пользователей.

Дальнейшая работа возможна в следующих направлениях:

- Собранный набор данных можно использовать для обучения новых моделей с отличной архитектурой, рассчитанной на большее число данных.
- Дополнительную информацию о рефакторингах, доступную как в собранном наборе данных, так и для новых примеров из плагина, можно использовать как входные признаки для будущих моделей.
- Инструмент для сбора и автоматической разметки данных можно использовать для извлечения примеров из другого набора проектов, формируя тем самым еще больший набор данных для задачи.
- Методику разметки примеров с помощью краудсорсинга можно масштабировать и применить для большего числа примеров.

Благодарность

Автор благодарит исследователей JetBrains Research Зарину Курбатову и Егора Богомолова за менторство, переданный опыт и помощь в проведении данного исследования.

Отдельное спасибо Илье Дорошенко за моральную поддержку в процессе написания данной работы.

Приложение

Метрики классификации

При бинарной классификации объект относится к классу положительных (P) или отрицательных (N). Предсказания делятся на 4 класса: истинно положительные (TP), ложно положительные (FP), истинно отрицательные (TN), ложно отрицательные (FN).

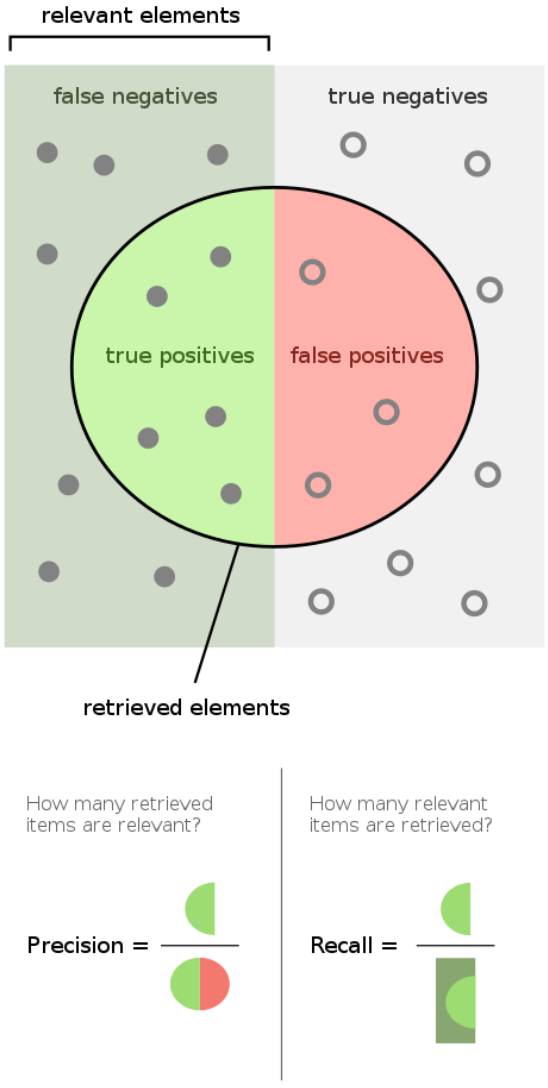

Рис. 27: Precision and Recall, источник [50]

Precision – доля релевантных сущностей среди полученных. В задачах классификации Precision рассчитывается как число истинно положительных предсказаний поделенное на число положительных предсказаний.

$$precision = \frac{TP}{TP + FP}$$

Recall – доля полученных сущностей среди релевантных. Recall в задачах классификации вычисляется как число истинно положительных предсказаний поделенное на число элементов с положительной меткой.

$$recall = \frac{TP}{TP + FN}$$

F1-score – гармоническое среднее Precision и Recall.

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Accuracy (точность, Acc) – доля истинных ответов.

$$Accuracy = \frac{TP + TN}{P + N}$$

Список литературы

[1] Scott N Woodfield, Hubert E Dunsmore, and Vincent Yun Shen. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th international conference on Software engineering*, pages 215–223, 1981.

[2] Ted Tenny. Procedures and comments vs. the banker's algorithm. *Acm Sigcse Bulletin*, 17(3):44–53, 1985.

[3] Ted Tenny. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271, 1988.

[4] Thomas M Pigoski. *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing, 1996.

[5] Shari Lawrence Pfleeger and Joanne M Atlee. *Software engineering: theory and practice*. Pearson Education India, 1998.

[6] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 53–64. IEEE, 2019.

[7] Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C Gall. Analyzing the co-evolution of comments and source code. *Software Quality Journal*, 17(4):367–394, 2009.

[8] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing apis documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 27–37. IEEE, 2017.

[9] Walid M Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E Hassan. On the relationship between comment update practices and software bugs. *Journal of Systems and Software*, 85(10):2293–2304, 2012.

[10] Seppuku Development. sepuku. https://github.com/seppukudevelopment/seppuku/blob/227956e97b61f182763ad3b2155bdc60dfa946ba/src/main/java/me/rigamortis/seppuku/api/util/FileUtil.java, 2020.

[11] Tegawendé F Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillere. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th annual computer software and applications conference*, pages 303–312. IEEE, 2013.

[12] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. Deep just-in-time inconsistency detection between comments and source code. *arXiv preprint arXiv:2010.01625*, 2020.

[13] Wikipedia. Toloka ai — a crowdsourcing platform and microtasking project launched by yandex. https://toloka.ai/. [Online; accessed 13-April-2022].

[14] Zhongxin Liu, Xin Xia, David Lo, Meng Yan, and Shanping Li. Just-in-time obsolete comment detection and update. *IEEE Transactions on Software Engineering*, 2021.

[15] Fazle Rabbi and Md Saeed Siddik. Detecting code comment inconsistency using siamese recurrent network. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 371–375, 2020.

[16] Alfonso Cimasa, Anna Corazza, Carmen Coviello, and Giuseppe Scanniello. Word embeddings for comment coherence. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 244–251. IEEE, 2019.

[17] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. Coherence of comments and method implementations: a dataset and an empirical investigation. *Software Quality Journal*, 26(2):751–777, 2018.

[18] Inderjot Kaur Ratol and Martin P Robillard. Detecting fragile comments. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 112–122. IEEE, 2017.

[19] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing apis documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 27–37. IEEE, 2017.

[20] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 260–269. IEEE, 2012.

[21] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 11–20. IEEE, 2011.

[22] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* icomment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 145–158, 2007.

[23] Nataliia Stulova, Arianna Blasi, Alessandra Gorla, and Oscar Nierstrasz. Towards detecting inconsistent comments in java source code automatically. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 65–69. IEEE, 2020.

[24] Ankita Sadu. *Automatic detection of outdated comments in open source Java projects*. PhD thesis, ETSI_Informatica, 2019.

[25] Zhiyong Liu, Huanchao Chen, Xiangping Chen, Xiaonan Luo, and Fan Zhou. Automatic detection of outdated comments during code changes. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 154–163. IEEE, 2018.

[26] How to write doc comments for the javadoc tool. `https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html`. Accessed: 2022-03-21.

[27] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152, 1992.

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

[29] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[30] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. From word embeddings to document distances. In *International conference on machine learning*, pages 957–966. PMLR, 2015.

[31] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[32] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[34] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[35] Intellij idea. https://www.jetbrains.com/idea/. Accessed: 2022-04-01.

[36] JetBrains. Coroutines. https://kotlinlang.org/docs/coroutines-overview.html.

[37] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, New York, NY, USA, 2018. ACM.

[38] What is refactoring. http://wiki.c2.com/?WhatIsRefactoring. Accessed: 2022-03-21.

[39] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee, 2016.

[40] Github: Where the world builds software. https://github.com/.

[41] Amazon ec2, secure and resizable compute capacity for virtually any workload. https://aws.amazon.com/ec2/.

[42] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[43] Egor Bogomolov Zarina Kurbatova JetBrains Research, Pavlov Ivan. Commentupdater. https://github.com/JetBrains-Research/CommentUpdater, 2021.

[44] JetBrains. Code inspection. [Online; accessed 29-april-2022].

[45] Wikipedia. Visitor pattern — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Visitor%20pattern&oldid=1077794549, 2022. [Online; accessed 13-April-2022].

[46] ONNX Runtime developers. Onnx runtime. https://onnxruntime.ai/, 2021. Version: 0.1.2.

[47] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[48] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[49] Netron: Visualizer for neural network, deep learning and machine learning models. https://www.lutzroeder.com/ai.

[50] Precision and Recall. Precision and recall — Wikipedia, the free encyclopedia. [Online; accessed 27-April-2022].