Санкт-Петербургский Государственный Университет
Математико-механический факультет

Математическое обеспечение и администрирование информационных систем
Кафедра системного программирования

Захаров Глеб Игоревич

Автоматическая рекомендация рефакторинга “Выделение метода” в коде на Java

Бакалаврская работа

Научный руководитель:
к.т.н., доц. Брыксин Т.А.

Рецензент:
руководитель направления “Машинное обучение и анализ данных”,
ООО “Интеллиджей Лабс” Шпильман А. А.

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY

Gleb Zakharov

Automatic "Extract Method" Refactoring Recommender for Java Code

Graduation Thesis

Scientific supervisor:
Associated Prof. Bryksin T. A.

Reviewer:
Head of Machine Learning and Data Analysis Department,
IntelliJ Labs Co. Ltd. Shpilman A. A.

Saint-Petersburg
2019

Оглавление

Введение

Рефакторинг — это процесс изменения внутренней структуры программы, не затрагивающий ее внешнего поведения и имеющий целью облегчить понимание ее работы [3]. Рефакторинг улучшает читаемость кода и уменьшает его сложность. Производить рефакторинг можно во время работы над новой функциональностью: можно подстроить код под свои нужды, исключить дублирование или сделать архитектуру более понятной.

Рефакторинг “Выделение метода” позволяет выделить часть функциональности из метода в новый метод. Такой рефакторинг помогает абстрагировать реализацию этой функциональности и уменьшить сложность метода. После скрытия реализации эту функциональность можно реализовать по-другому или оптимизировать при профилировании. Помимо этого “Выделение метода” помогает скрыть очередность действий, а еще переиспользовать код в будущем. “Выделение метода” является одним из самых часто используемых рефакторингов, которые существенно меняют структуру кода, а не только сигнатуры и названия [9].

Программисту приходится тратить время и силы, чтобы инициировать и выполнить такой рефакторинг. Выбранный блок кода может принимать слишком много аргументов и возвращать несколько значений, и тогда программа станет только сложнее для понимания. К тому же такой рефакторинг может породить ошибки в программе, которые потребуют больших усилий при отладке и написании тестов. Например, программист может выделить блок кода так, что он нарушит семантический смысл метода или будет изменять состояние объекта и такой метод будет трудно переиспользовать. Поэтому нередко программисты боятся переиспользовать собственный и чужой код.

Хотелось бы иметь такой инструмент, который подсказывает программисту блоки кода — кандидаты для выделения. При этом он должен работать быстро, корректно (программист должен быть уверен,

что выделение кандидатов не нарушит поведение программы) и удобно (например, как расширение к используемой среде разработки).

IntelliJ Platform — это платформа для построения IDE. Она используется компанией JetBrains как основа продуктов IntelliJ IDEA, WebStorm, DataGrip, Rider и других. В платформе уже есть средства для осуществления рефакторинга “Выделение метода”, но его хотелось бы дополнить средством автоматической рекомендации таких рефакторингов.

Один из способов расширения функциональности IDE на базе платформы IntelliJ — это написание расширений в виде подключаемых модулей-плагинов. Платформа позволяет осуществлять проход по синтаксическому дереву разбора программы, и таким образом анализировать код.

Постановка задачи

Цель работы: реализация плагина для Intellij IDEA, рекомендущего рефакторинги “Выделение метода” в коде на Java. Для этого необходимо выполнить следующие задачи.

1. Проанализировать существующие решения для других IDE.
2. Продумать архитектуру и реализовать выбранный алгоритм для Intellij IDEA.
3. Оптимизировать алгоритм для работы на больших проектах.
4. Провести апробацию разработанного решения.

1. Обзор существующих решений

На протяжении последних десяти лет было создано несколько инструментов для автоматического поиска и рекомендации рефакторингов “Выделение метода”. Далее будет рассмотрена функциональность некоторых наиболее известных решений в данной области, а также алгоритмы, которые лежат в их основе.

1.1. JDeodorant

Стоит начать с инструмента JDeodorant [2], представленного в 2007 году. Это подключаемый модуль для Eclipse, который распознает проблемы плохого дизайна кода (также известные как “запахи” кода), и к каждой проблеме предлагает подходящий рефакторинг.

В 2011 году инструмент стал поддерживать и рефакторинг “Выделение метода”. Пользователь выбирает проект, и в отдельном окне появляется список кандидатов для выделения. Нажав на понравившейся рефакторинг, пользователь может увидеть подсвеченный код кандидата и возможность выделить метод средствами Eclipse.

Инструмент работает по принципу анализа зависимостей по данным и управлению между операторами [1]. JDeodorant ищет все операторы в методе, отвечающие за вычисление одной переменной или изменение состояния одного объекта, затем предлагает таких кандидатов пользователю.

Однако, несмотря на сложный статический анализ, используемый в инструменте, некоторые зависимости между операторами не определяются из-за их семантической несвязности [11]. Например, в листинге 1 представлен алгоритм сохранения заказов магазина в базу данных. JDeodorant для вычисления переменной t, обозначающей транзакцию, меняет строки 8 и 9 местами, и нарушает функциональный смысл метода.

Если же говорить о подходе, то выделение подсчета переменных и состояния объектов не исчерпывает все причины выделения функциональности в отдельный метод. Такой рефакторинг может осуществляться по разным общим причинам и не ко всем получается применить готовый рецепт.

Листинг 1: метод из проекта MyWebMarket с внешней связностью кода

```
1 public String save() throws Exception {
2     ....
3     Session s = HibernateUtil.getSessionFactory().openSession();
4     Transaction t = sess.beginTransaction();
5     for (PurchaseOrderItem item : purchaseOrder.getPurchaseOrderItems()){
6         item.setPurchaseOrder(this.purchaseOrder);
7     }
8     s.save(this.purchaseOrder);
9     t.commit();
10    s.close();
11    ....
12 }
```

1.2. JExtract

Появившийся в 2015 году инструмент JExtract [12] также позволяет пользователю получить список кандидатов для выделения, подсветить и выделить их средствами Eclipse IDE. В отличие от JDeodorant это решение не основывается на специальных шаблонах кода. JExtract сортирует всех кандидатов, исходя из принципа *максимальной внутренней связности* и *минимальной внешней связности* кода.

Работу инструмента можно разделить на несколько этапов:

1. инструмент генерирует возможных кандидатов;
2. инструмент проверяет кандидатов на корректность;
3. инструмент сортирует кандидатов и выводит пользователю.

На первом шаге для каждого блока кода JExtract перебирает все его подблоки (просто выбирая начальный и конечный оператор) и про-

веряет, можно ли их выделить с помощью встроенных средств Eclipse. Так как теперь кандидаты — это непрерывные блоки кода, решается проблема JDeodorant с изменением порядка операторов.

На втором шаге JExtract проверяет, можно ли выделить кандидатов с помощью функции isValid(). В инструменте isValid() использует код Eclipse IDE для проверки возможности выделения. Функция проверяет, например, что кандидат не может возвращать больше одного значения.

В конце для каждого такого подблока-кандидата вычисляется функция ранжирования. Сначала вычисляются множества локальных переменных, полей кандидата и оставшейся части метода — *дополнения кандидата* до метода. Затем с помощью функции расстояния между множествами вычисляется, насколько далеко эти множества лежат друг от друга (в дополнение такая же функция вычисляется для множеств используемых типов и пакетов в функции). В инструменте выбирали между несколькими функциями и остановились на функции расстояния Кульчинского. Если существует два множества S и S', то:

$$\mathrm{dist}(S, S') = 1 - \frac{1}{2}\left(\frac{a}{a+b} + \frac{a}{a+c}\right),$$

где $a = \left|S \cap S'\right|, b = \left|S \setminus S'\right|, c = \left|S' \setminus S\right|$.

Интуитивно, чем дальше множества находятся друг от друга, тем менее связны между друг другом кандидат и его дополнение. Таким образом, можно надеяться, что у кандидата будет своя обособленная функциональность.

Для ускорения генерации кандидатов JExtract ограничивает количество операторов в кандидате и дополнении кандидата константой, которую может задать пользователь. Также нужно отметить, что пользователю не нужно показывать всех кандидатов, которых создается гораздо больше, чем в JDeodorant, можно показывать только несколько первых — top-k — кандидатов.

Если говорить о недостатках, то нужно сказать, что конкретная

функция расстояния между множествами взялась из эксперимента на 14 методах одного проекта, поэтому она необязательно правильно ранжирует кандидатов. К тому же такой подход учитывает только структурную зависимость кода, пытаясь отделить как можно более независимого кандидата, но это не всегда правда: в коде листинга 2 можно выделить условие if-оператора, однако множество используемых полей в нем пересекается с дополнением кандидата в строках 6 и 7.

Листинг 2: код ранней версии JUnit с невыделенным условием

```
1 public String compact(String message) {
2     if (expected == null || actual == null || areStringsEqual())
3         return Assert.format(message, expected, actual);
4     findCommonPrefix();
5     findCommonSuffix();
6     String expected = compactString(this.expected);
7     String actual = compactString(this.actual);
8     return Assert.format(message, expected, actual);
9 }
```

1.3. SEMI

В 2016 году появился инструмент SEMI [5]. Он основывается на принципе единой ответственности (Single Responsobility Principle, SRP) на уровне метода и нужен для декомпозиции длинных методов. Алгоритм работы SEMI разделен на 2 этапа: сначала выбираются кандидаты для выделения из метода, затем кандидаты группируются и выводятся пользователю.

При генерации кандидатов авторы инструмента считают, что два оператора связны, если они используют одну переменную, вызывают методы одного объекта или вызывают методы одного класса. Все операторы, которые удовлетворяют одному из этих условий и при этом меньше по размеру заданной константы, собираются в кандидата, который затем проверяется с помощью функции isValid() из инструмента JExtract.

На втором шаге инструмент группирует похожих кандидатов. Идея

в том, что два кандидата, у которых сильно пересекается множества операторов и которые имеют примерно одинаковый размер, скорее всего предоставляют одну функциональность. Из “похожих” кандидатов выбирается один представитель всей группы, и он показывается пользователю. Более точно, если существуют два кандидата A и B, авторы вводят две функции схожести:

$$\text{Difference\_In\_Size}(A, B) = \frac{|A.LoC - B.LoC|}{\text{MIN}(A.LoC, B.LoC)},$$

$$\text{Overlap}(A, B) = \frac{\text{overlap}(A, B)}{\text{MAX}(A.LoC, B.LoC)},$$

где $\text{overlap}(A, B)$ — количество одинаковых операторов. Если обе функции входят в промежуток, который установил пользователь, то их можно считать похожими. Авторы выбирают оптимум с помощью метрики LCOM2 или Lack of Cohesion Metric, которая показывает степень внутренней связности класса. LCOM2 вычисляется таким образом: пусть у класса есть n методов, $I_1, ..., I_n$ — множества полей, используемых в каждом методе. Метрика LCOM вводит два множества:

$$P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\},$$
$$Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\},$$

тогда LCOM2 вычисляется по формуле:

$$LCOM2 = |P| - |Q|.$$

Авторы вычисляют LCOM2 до и после рефакторинга и вычитают одно из другого. Таким образом, показываются улучшение связности класса. Наилучший кандидат выбирается представителем группы.

К недостаткам можно отнести то, что такой способ не ищет все рефакторинги. Как показано в разделе 1.5, точность такого метода в экспериментах оказалась невысока. Однако из-за группировки SEMI хорошо справляется с нахождением разных кандидатов в длинных методах.

1.4. GEMS

В 2017 году появился инструмент GEMS [4]. Используя такой же алгоритм генерации и проверки кандидатов, как у JExtract, GEMS заменяет фазу сортировки кандидатов машинным обучением.

Для проверенного кандидата GEMS считает набор метрик: число операторов, число условных операторов, литералов, тернарных операторов, присваиваний, локальных переменных, а также количество используемых типов и пакетов. Кроме того, в алгоритме считаются метрики внутренней и внешней связности (cohesion и coupling). Для каждого кандидата GEMS рассматривает 4 типа элементов p из множества P: вызовы функций, пакеты (и их родители), доступ к переменным и доступ к типам. Для каждого типа p GEMS выбирает два самых часто встречающихся элемента и в качестве меры внешней связности авторы берут отношение двух дробей: насколько часто элемент встречается в кандидате и насколько часто он встречается в дополнении кандидата до целого метода. Для вычисления внутренней связности GEMS берет отношения количества операторов с выбранным элементом к количеству всех операторов.

В качестве тренировочных данных для модели брались “позитивные” настоящие и искусственные рефакторинги, а также “негативные” искусственные рефакторинги. Настоящие “позитивные” рефакторинги были получены с помощью инструмента RefactoringMiner [8] из репозиториев GitHub с высоким рейтингом (больше 8000 звезд). RefactoringMiner позволяет находить рефакторинги в истории коммитов с большой точностью. Затем каждый рефакторинг был просмотрен авторами, и были убраны рефакторинги, добавляющие новую функциональность. Всего получилось 267 методов.

Естественно, такой подход требует больших человеческих усилий, поэтому к “настоящему” набору данных также был добавлен искусственный: находились методы, которые вызываются один раз, и с помощью рефакторинга “Inline method” они вносились в код. В качестве

“негативных” примеров брался любой кандидат, который проходил проверку isValid(): такой рефакторинг скорее всего окажется “плохим”.

В качестве модели использовался Gradient Boosting Classifier. Градиентный бустинг сочетает множество слабых моделей (например, решающих деревьев), которые запускаются последовательно, при этом каждая последующая модель пытается минимизировать функцию потерь предыдущей модели и обучающей выборки. Поэтому если функция потерь на предыдущем шаге имеет неслучайную структуру, то модели на текущем и следующих шагах учатся эту структуру распознавать.

К недостаткам можно отнести слабое выделение методов в длинных методах. Это связано с тем, что в отличие от SEMI, GEMS не группирует кандидатов, поэтому в первых top-k (скажем, в первых пяти) показываются кандидаты из одной группы SEMI.

1.5. Сравнение инструментов

В таблице 1 можно увидеть результаты сравнения четырех инструментов из статьи [4]. Здесь 1% Tolerance означает, сколько дополнительных операторов (в данном случае один) кандидат может иметь, чтобы его при этом считать правильным. Наилучшие показатели выделены жирным шрифтом. Как можно увидеть, GEMS точнее и полнее находит рефакторинги на тестовом наборе данных в коротких методах.

Можно также увидеть, что при малом Tolerance в длинных методах SEMI находит больше рефакторингов, чем GEMS. Авторы статьи GEMS предполагают, что это связано с группировкой похожих кандидатов.

		Методы < 30 операторов			Методы ≥ 30 операторов		
Tools	Tolerance	Precision	Recall	F-measure	Precision	Recall	F-measure
GEMS	1%	**22.5%**	**54.2%**	**31.8%**	13.3%	31.9%	18.8%
	2%	**28.5%**	**59.8%**	**38.6%**	**17.4%**	**41.5%**	**24.5%**
	3%	**34.3%**	**62.6%**	**44.3%**	**25.3%**	**46.2%**	**32.7%**
JExtract	1%	12.6%	52.2%	20.4%	6.6%	16.1%	9.4%
	2%	13.1%	**59.3%**	21.5%	8.0%	19.3%	11.3%
	3%	15.0%	61.9%	24.2%	8.0%	19.3%	11.3%
SEMI	1%	12.9%	38.0%	19.2%	**16.4%**	**38.7%**	**23.0%**
	2%	14.6%	47.0%	22.3%	**17.9%**	**41.9%**	**25.0%**
	3%	18.8%	55.5%	28.1%	19.1%	45.1%	26.9%
JDeodorant	1%	17.4%	14.8%	16.0%	12.0%	9.6%	10.7%
	2%	21.1%	18.4%	19.7%	14.3%	12.9%	13.5%
	3%	28.0%	23.8%	25.7%	16.0%	12.9%	14.2%

Таблица 1: Сравнение инструментов рекомендации рефакторингов “Выделение метода” [4]

1.6. Выводы

В рассмотренных решениях представлено множество интересных идей. Авторы инструментов исходили из мысли, что выделенный метод должен обладать собственной, отличной от оставшегося метода, функциональностью. По способу выделения этого метода инструменты можно разделить на две части. JDeodorant и SEMI предлагают новый подходы к генерации кандидатов для выделения, в то время как JExtract и GEMS генерируют всех возможных кандидатов и затем сортируют их.

У каждого из представленных инструментов есть свои достоинства и недостатки:

- JDeodorant имеет хороший интерфейс и долгую историю разработки, однако его подход оказывается неполным и не всегда кор-

ректным;

- JExtract не опирается на конкретные случаи выделения, но его ранжирующая функция не всегда дает ожидаемые результаты;
- SEMI хорошо показывает себя в длинных методах из-за группировки кандидатов, но выделение кандидатов основано только на конкретных случаях;
- GEMS показывает себя лучше всего в сравнении с другими инструментами и рассматривает как внутреннюю, так и внешнюю связность, но его результат был хуже в длинных методах.

Поэтому решено было перенести инструмент GEMS на IntelliJ Platform, а затем рассмотреть другой набор данных и также реализовать группировку SEMI после сортировки всех кандидатов.

1.7. Обзор IntelliJ Platform

IntelliJ Platform предлагает множество разных интерфейсов для расширения функциональности[1]. С помощью создания плагинов можно не только добавлять простые меню в IDE, но и поддерживать новые языки и отладки. Далее будет рассказано о частях платформы, которые будут необходимы для реализации требуемого решения.

Для создания пользовательского интерфейса в платформе существует концепт “actions” или “действий”. Действия — это классы, унаследованные от базового класса AnAction, они вызываются при нажатии на кнопку в меню или тулбаре. Таким образом можно реализовать базовый класс, который запускает вычисления и выводит пользователю результат. Графический интерфейс пользователя в Intellij Platform основан на библиотеке Swing. Эти классы согласуют стиль платформы и плагина, поэтому рекомендуется использовать их.

[1] `http://www.jetbrains.org/intellij/sdk/docs/intro/intellij_platform.html`

Для анализа какого-то участка кода существует класс AnalysisScope. С помощью него пользователь может выбрать область кода, которую хочет проанализировать на возможные рефакторинги. Таким образом, можно проанализировать отдельный файл, модуль, директорию или целый проект, а также использовать поддержку системы контроля версий и анализировать только измененные файлы.

Если пользователь захочет проанализировать только отдельный метод или кандидата, у платформы есть доступ к текущему редактору кода (класс Editor), а также каретке и выделенному месту в коде. Поэтому пользователь может выделить часть кода или весь метод, и плагин покажет только кандидатов из выделенной части.

Интерфейс PSI (Programming Structure Interface) предоставляет доступ к синтаксической структуре кода. С помощью паттерна “Посетитель” можно перемещаться по дереву кода и считать разнообразные метрики или искать использования методов. У платформы реализовано несколько классов “Посетитель”, каждый с собственным поведением.

PSI-tree представляет собой дерево кода, состоящее из PSI элементов (PsiElement). От PsiElement унаследованы разные сущности Java-кода, например PsiStatement и PsiExpression. При перемещении по дереву можно собирать статистику по элементам и считать метрики GEMS.

2. Архитектура и оптимизация решения

На рис. 1(б) представлена схема предлагаемого решения 'Apeman' (Another Plugin for 'Extract Method' Automatically and Neatly), получившаяся из схемы решения GEMS, показанного на рис. 1(а). Синим выделены измененные части инструмента, а зеленым — что добавлена группировка SEMI.

(а) Схема решения GEMS (б) Схема решения Apeman

Рис. 1: Сравнение инструментов, синим показано, что изменено, зеленым — что добавлено

На рис. 2 показана диаграмма классов инструмента, цветом выделены классы Intellij Platform и связи плагина с ними. Решение реализовано с помощью архитектуры “Каналы и фильтры”, что делает его гибким для расширения и простым для понимания. Сначала пользователь вызывает из метод getCandidates() из модуля User Interface. getCandidates() создает анализатор с помощью главного класса ядра Apeman Core. В ядре инструмент генерирует всех возможных кандидатов для выделения с помощью класса CandidateGenerator. Затем для каждого кандидата вычисляется множество метрик с помощью классов Metric и Calculator, которые кодируют информацию о сопряжении и связности кандидата с методом. Это происходит с помощью системы Feature Extraction. По полученному вектору метрик модель классифицирует всех кандидатов и присваивает им вероятность, насколько они подходят для выделения. Все кандидаты ранжируются по вероятности, похожие кандидаты группируются и пользователю показываются

только top-k кандидатов с помощью класса FilteringTool. В следующих подразделах будут рассмотрены шаги алгоритма по отдельности.

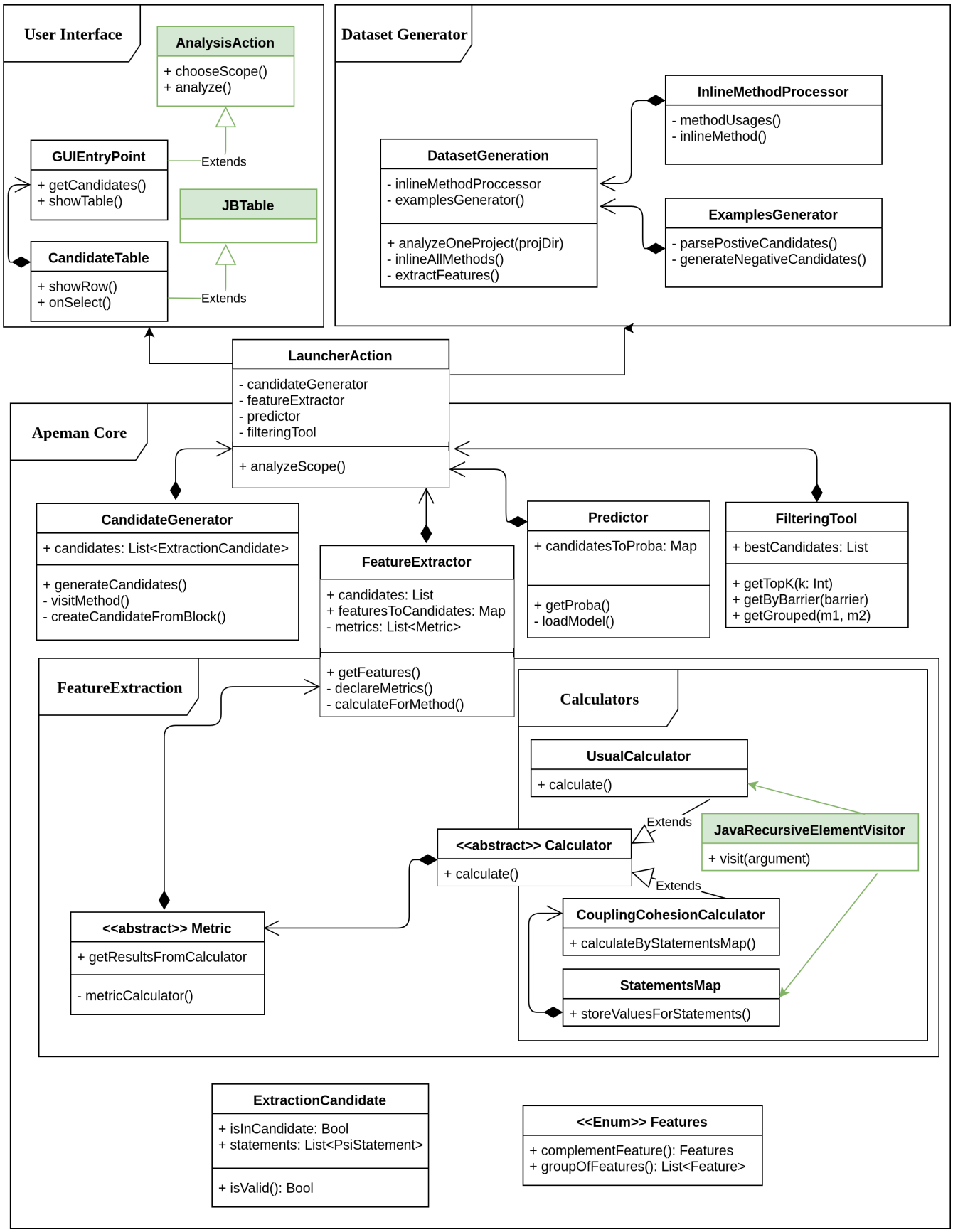

Рис. 2: Диаграмма классов apeman-plugin

2.1. Генерация кандидатов

Как и в инструменте GEMS, для каждого метода генерируются все возможные кандидаты, которые при выделении не нарушат семантический смысл программы. Для этого авторам GEMS (и авторам JExtract, этот подход взят у них) пришлось ограничить пространство всех возможных кандидатов для выделения [4, 12]:

1. весь метод рассматривается как набор блоков кода, каждый блок состоит из множества операторов;
2. кандидат — это последовательное множество операторов в блоке кода;
3. корректный кандидат — кандидат, который прошел валидацию;
4. валидация — функция, которая проверяет, можно ли выделить этого кандидата так, чтобы он не нарушил поведение программы.

Так же, как в инструментах GEMS и JExtract, функция валидации проверяет кандидата встроенными средствами IDE. Например, кандидат не должен иметь несколько возвращаемых значений. Также в ходе использования инструмента все кандидаты, которые неадекватно выделялись средствами IDE, были дополнительно включены в функцию валидации. Например, с помощью IDEA можно выделить несколько case из switch-оператора, но при этом сам оператор switch в этом методе не появится. Такие кандидаты также были исключены из рассмотрения.

2.2. Вычисление метрик

После генерации кандидатов для каждого метода все кандидаты передаются фильтру, который считает метрики. В Apeman все метрики можно разделить на 4 части:

- количество элементов в кандидате;

- количество элементов в дополнении до кандидата;
- сопряжение кандидата с методом;
- внутренняя связность кандидата.

В отличии от GEMS, которое вычисляет метрики для каждого кандидата, в Apeman метрики вычисляются для каждого оператора. Так как один оператор находится во множестве кандидатов, это существенно снижает сложность вычисления. Например, для метода размером 40 операторов (около 1300 кандидатов) GEMS вычисляет все метрики и показывает результаты пользователю примерно за 5 минут, а Apeman делает это на том же компьютере за 3 секунды. Таким образом, за то же время можно находить кандидатов в десятках файлов, а не в одном методе.

2.3. Предсказание вероятности и сбор данных

Как и в GEMS, в инструменте решено использовать алгоритм градиентного бустинга [4]. Так как инструмент GEMS записывает всех кандидатов в файл, а затем запускает Python-скрипт, который предсказывает вероятности с помощью библиотеки scikit-learn[2], решение получается достаточно медленным. Поэтому было решено обучить собственную модель с помощью TensorFlow Java API[3]. Таким образом, предсказание вероятности было ускорено в 3 раза.

Так как в открытом доступе кода по вычислению метрик нет, не всегда было понятно, как именно авторы вычисляют те или иные метрики. GEMS предоставили набор данных только в виде таблицы метрик и пересчитать их данные было невозможно. Поэтому было решено создать собственный набор данных с посчитанными метриками.

Так же, как и в GEMS, для сбора данных были выбраны семь популярных проектов на GitHub на языке Java. Из проектов были выделены

[2] `https://scikit-learn.org/stable/`

[3] `https://www.tensorflow.org/api_docs/java/reference/org/tensorflow/package-summary`

позитивные и негативные кандидаты.

Если метод используется ровно один раз, можно произвести рефакторинг “Inline Method”, то есть внести его в тот код, где он используется, и считать его позитивным кандидатом для выделения. Интуитивно, если программист выделил метод, который используется один раз, то он это сделал для улучшения читаемости кода, и такой метод должен быть выделен снова.

В качестве негативных примеров были выбраны случайные корректные кандидаты. Скорее всего, случайно взятые кандидаты окажутся плохими для выделения. Негативные примеры брались из тех же методов, куда были внесены позитивные кандидаты. Таким образом, модель должна научиться выделять только позитивных кандидатов независимо от самого метода. Кроме того, цель инструмента — выделять методы в тот же класс, поэтому позитивные кандидаты были ограничены условием, что используются они в том же классе. Так исключаются ошибки импорта и неправильного доступа к private и protected полям класса (рефакторинг “Inline Method” из IntelliJ IDEA не импортирует классы автоматически).

В итоге получилось 6100 позитивных примеров и 6100 негативных примеров.

Перед обучением была произведена предобработка полученного набора данных. Так как негативных кандидатов достаточно легко собрать, можно выбрать их в несколько раз больше, чем позитивных кандидатов, и оставить только те, которые точно не стоит выделять. Такая техника называется выбор подвыборки данных или undersampling.

Также набор данных был отфильтрован не только по строчкам-примерам, но и по метрикам с помощью техники permutation importance [10] или отбора признаков с помощью перестановок. В разделе 3.1 приведены сравнения алгоритма с разным соотношением веса позитивных и негативных кандидатов и разными техниками выделения подвыборок.

2.4. Ранжирование и группировка кандидатов

Кандидаты группируются с помощью алгоритма группировки SEMI. Таким образом, можно исключить похожие результаты из top-k кандидатов, и в больших методах пользователь увидит только разных кандидатов.

Также можно установить барьер по вероятности, и показывать пользователю только те кандидаты, у которых вероятность больше.

2.5. Пользовательский интерфейс

Инструмент написан на языке Kotlin. На рис. 3 представлен его пользовательский интерфейс. Можно проанализировать заданную область видимости (например, директорию, файл или модуль), а также есть возможность проанализировать один метод или отдельного кандидата, выбрав соответсвующий пункт в меню “Check methods with apeman”. После этого появится таблица с результатирующими кандидатами, вероятностью и результатами метрик. При перемещении по таблице каждый кандидат выделяется в редакторе кода. На этом рисунке показан выделенный кандидат из библиотеки spoon[4]. После поиска подходящего кандидата можно выбрать ”Рефакторинг” в выпадающем меню IntelliJ IDEA и выделить его. Код проекта представлен на GitHub[5].

[4] https://github.com/INRIA/spoon

[5] https://github.com/ml-in-programming/apeman

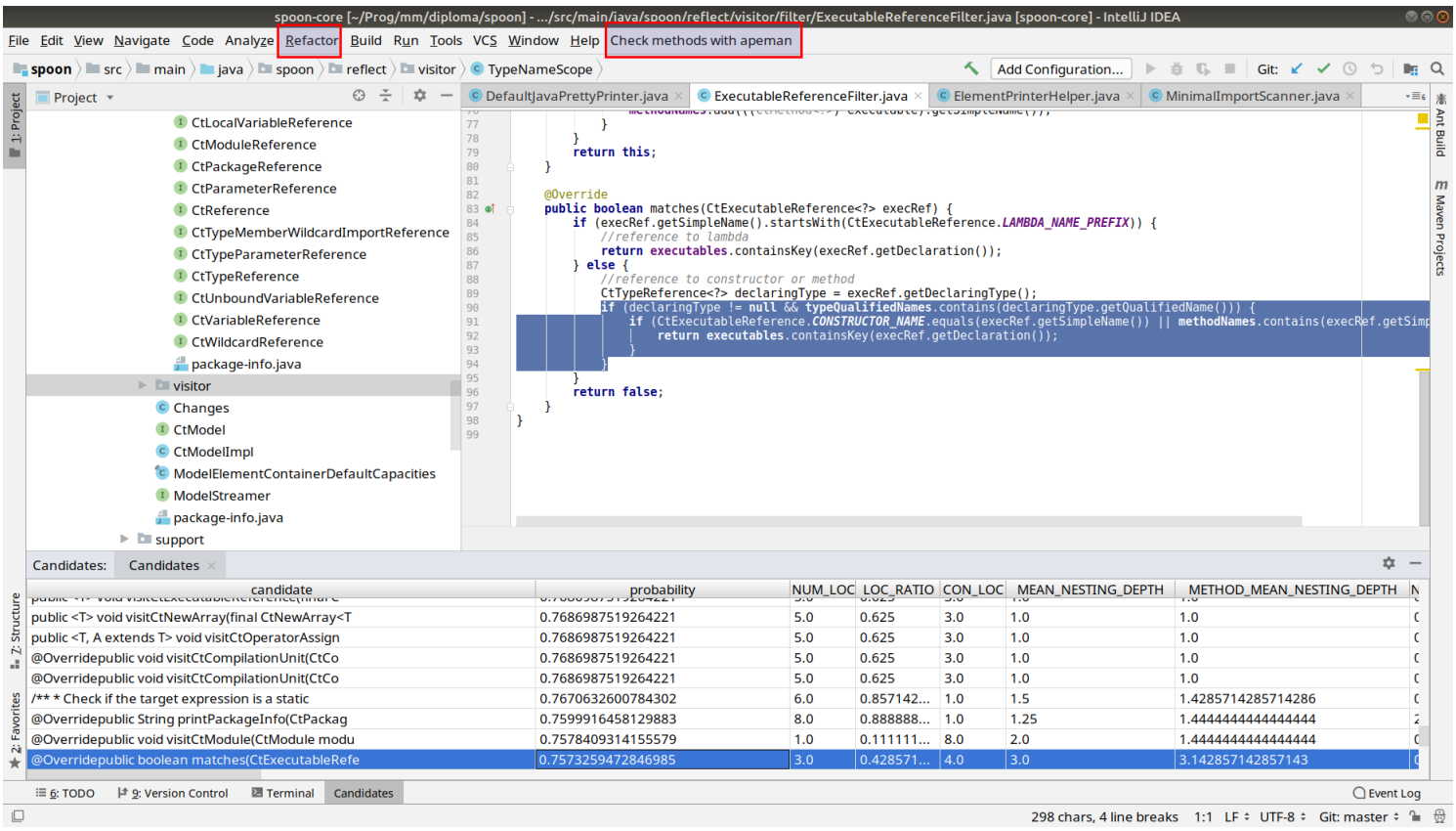

Рис. 3: Пользовательский интерфейс разработанного решения

3. Апробация решения

В данном разделе будут сравнены техники выбора подвыборки из существующего набора данных, а затем произведено сравнение с инструментом GEMS.

3.1. Выбор наилучшей подвыборки

Изначально в наборе данных было около 52000 негативных кандидатов и 6100 позитивных кандидатов. В качестве корректировочного набора были взяты случайные 700 позитивных и 700 негативных кандидатов из набора данных. Этот набор данных нужен для того, чтобы определить оптимальные параметры градиентного бустинга. Для проверки точности решения были выбраны еще по 700 позитивных и негативных кандидатов. Для каждого элемента классификатор должен

правильно определить, можно ли его выделять.

В начале данные были нормализованы. Это было необходимо, так как многие техники выбора подвыборки требуют сравнимых величин в разных колонках. Затем были выбраны несколько техник выбора подвыборки и удаления выбросов в данных.

В качестве техник выбора подвыборки по строкам были рассмотрены InstanceHardnessThreshold [13], NearMiss-1 и NearMiss-3 [14], а также случайный выбор подвыборки и вся выборка с измененным весом позитивных кандидатов. Все техники брались из библиотеки imbalanced-learn[6] для языка Python. В качестве способов удаления выбросов были рассмотрены IsolationForest [7] и LocalOutlierFactor [6] из библиотеки scikit-learn.

Также набор данных был отфильтрован по колонкам с помощью техники permutation importance [10], при этом сравнивались результаты после двух таких фильтров. Сначала добавляется колонка со случайными значениями от 0 до 1, затем по очереди для каждой колонки перемешивались все значения, и сравнивались значения точности до и после. Разница считалась результатом "важности" колонки. Все колонки, у которых результат оказывался меньше, чем у случайной, отбрасывались.

Таким образом, можно было выделить только самые важные признаки для предсказания. В качестве сравнения использовалась метрика accuracy. Тестирование было запущено 3 раза, и из него были выделены максимальные результаты. Это связано с тем, что в алгоритме градиентного бустинга на TensorFlow нельзя выбрать зерно для псевдослучайных вычислений.

Сравнение представлено на таблице 2, цветами отмечены лучшие результаты. Лучше всего себя показало увеличение веса положительных примеров в функции потерь (строка "Weighted classes"), после исключения части признаков с помощью перестановок и выбросов данных с

[6] `https://imbalanced-learn.readthedocs.io/en/stable/introduction.html`

Outliers	No outliers			Isolation Forest			Local Outlier Factor		
Permutation Importance	#0	#1	#2	#0	#1	#2	#0	#1	#2
Weighted classes	0.59	0.64	0.68	0.54	0.53	0.66	0.63	0.68	0.66
Instance Hardness Threshold	0.60	0.61	0.58	0.57	0.59	0.60	0.58	0.63	0.61
NearMiss-1	0.54	0.51	0.58	0.57	0.56	0.61	0.51	0.53	0.54
NearMiss-3	0.56	0.58	0.54	0.58	0.59	0.62	0.60	0.51	0.59
Random Under Sampler	0.67	0.56	0.55	0.64	0.63	0.63	0.63	0.66	0.66

Таблица 2: Сравнение по метрике accuracy

помо. 0.68 для accuracy — это ожидаемый результат. Во-первых, проблема сложная, а во-вторых, сравнение происходит на неотфильтрованном наборе данных, то есть результаты сравниваются на другой выборке, из-за чего модели сложно предсказать результат.

3.2. Сравнение с GEMS

Для сравнения с GEMS были выбраны случайные 30 позитивных и 30 негативных кандидатов, которые не входили в другие наборы. Такой малый размер кандидатов связан с тем, что на нем проводилось сравнение с инструментом GEMS, а его было необходимо запускать вручную из-за закрытой реализации. На тестовом наборе было произведено сравнение с GEMS. На 24 примерах (13 отрицательных и 11 положительных примеров) инструмент GEMS не запустился из-за исключений во время подсчета метрик. На графиках 4 приведены сравнения по точности, полноте, accuracy и F-мере для тестового набора данных на оставшихся 36 примерах. Здесь Apeman-1 — первый результат в таблице 2 с accuracy 0.68, а Apeman-2 — второй результат.

3.3. Анализ результатов

Как видно из графиков, Apeman превосходит GEMS по точности, F-мере и accuracy-score. Apeman не превосходит GEMS по полноте. Это связано с тем, что на большинстве примеров GEMS выдает положительный результат, из-за этого получается почти 100%-ая полнота при

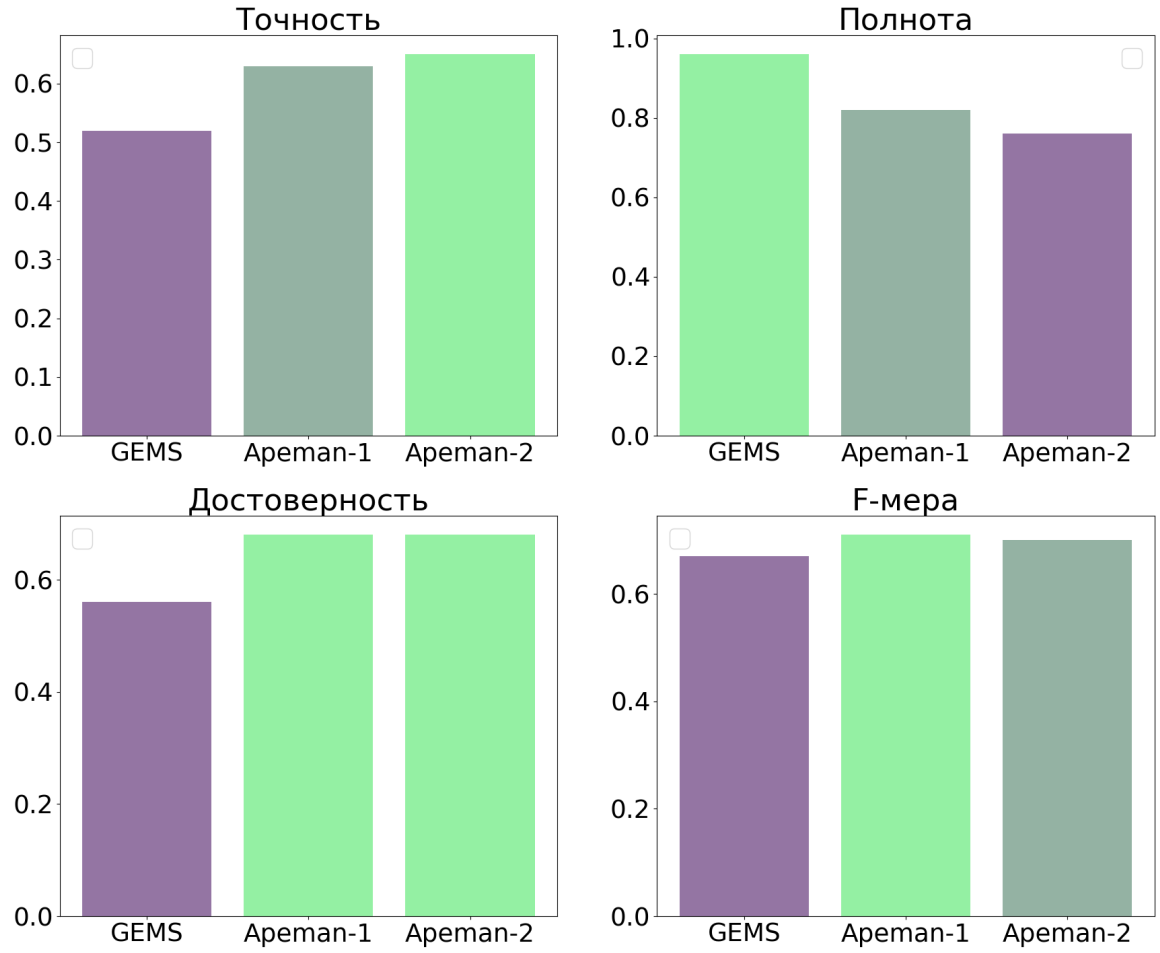

Рис. 4: Сравнение инструментов

маленькой точности. Поэтому было принято решение сравнить инструменты также по accuracy и F-мере.

Разница в accuracy и точности связана со способом отбора данных, а именно идеей выделения признаков с помощью permutation importance и убиранием выбросов в данных. Также в работе GEMS не использовался корректировочный набор данных для оптимизации модели градиентного бустинга.

Из-за группировки SEMI инструмент Apeman в 12 длинных методах из тестового набора данных (длинный — больше 30 операторов) не показывает пользователю похожих кандидатов в top-5, в отличие от

GEMS.

Также стоит отметить, что Apeman работает быстрее и на большем количестве примеров. Инструмент отработал на всех примерах из тестового набора данных, в отличие от GEMS.

3.4. Границы применимости

Подход Apeman не рассматривает семантический смысл выделяемого метода, а рассматривает только структурную похожесть. Для кодирования семантического смысла, возможно, стоит анализировать абстрактное дерево кода полностью, а не только сопряжение и связность.

4. Заключение

В ходе работы были получены следующие результаты:

- проанализированы существующие решения;
- решение GEMS реализовано для IntelliJ IDEA;
- решение оптимизировано по скорости и точности классификации;
- произведено сравнение с инструментом GEMS.

Код проекта доступен на GitHub[7].

Набор данных также доступен онлайн[8].

[7] https://github.com/ml-in-programming/apeman

[8] https://drive.google.com/drive/folders/1-s2qf-ZMQjzJGq2dbqs3zjIxkk2LKCy9?usp=sharing

Список литературы

[1] Ferrante Jeanne, Ottenstein Karl J., Warren Joe D. The Program Dependence Graph and Its Use in Optimization // ACM Trans. Program. Lang. Syst. — 1987. — . — Vol. 9, no. 3. — P. 319–349. — URL: http://doi.acm.org/10.1145/24039.24041.

[2] Fokaefs M., Tsantalis N., Chatzigeorgiou A. JDeodorant: Identification and Removal of Feature Envy Bad Smells // 2007 IEEE International Conference on Software Maintenance. — 2007. — Oct. — P. 519–520.

[3] Fowler Martin. Refactoring: Improving the Design of Existing Code. — Boston, MA, USA : Addison-Wesley, 1999. — ISBN: 0-201-48567-2.

[4] GEMS: An Extract Method Refactoring Recommender / S. Xu, A. Sivaraman, S. Khoo, J. Xu // 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE). — 2017. — Oct. — P. 24–34.

[5] Identifying Extract Method Refactoring Opportunities Based on Functional Relevance / S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou et al. // IEEE Transactions on Software Engineering. — 2017. — Oct. — Vol. 43, no. 10. — P. 954–974.

[6] LOF: Identifying Density-based Local Outliers / Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, Jörg Sander // SIGMOD Rec. — 2000. — . — Vol. 29, no. 2. — P. 93–104. — URL: http://doi.acm.org/10.1145/335191.335388.

[7] Liu Fei Tony, Ting Kai Ming, Zhou Zhi-Hua. Isolation-Based Anomaly Detection // ACM Trans. Knowl. Discov. Data. — 2012. — . — Vol. 6, no. 1. — P. 3:1–3:39. — URL: http://doi.acm.org/10.1145/2133360.2133363.

[8] A Multidimensional Empirical Study on Refactoring Activity / Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, Abram Hindle //

Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research. — CASCON '13. — Riverton, NJ, USA : IBM Corp., 2013. — P. 132–146. — URL: http://dl.acm.org/citation.cfm?id=2555523.2555539.

[9] Parnin C., Black A. P., Murphy-Hill E. How We Refactor, and How We Know It // IEEE Transactions on Software Engineering. — 2011. — 04. — Vol. 38. — P. 5–18. — URL: doi.ieeecomputersociety.org/10.1109/TSE.2011.41.

[10] Permutation importance: a corrected feature importance measure / André Altmann, Laura Toloşi, Oliver Sander, Thomas Lengauer // Bioinformatics. — 2010. — 04. — Vol. 26, no. 10. — P. 1340–1347. — http://oup.prod.sis.lan/bioinformatics/article-pdf/26/10/1340/16892402/btq134.pdf.

[11] Silva Danilo, Terra Ricardo, Valente Marco Tulio. Recommending Automated Extract Method Refactorings // Proceedings of the 22Nd International Conference on Program Comprehension. — ICPC 2014. — New York, NY, USA : ACM, 2014. — P. 146–156. — URL: http://doi.acm.org/10.1145/2597008.2597141.

[12] Silva Danilo, Terra Ricardo, Valente Marco Tulio. JExtract: An Eclipse Plug-in for Recommending Automated Extract Method Refactorings. — 2015. — 1506.06086.

[13] Smith Michael R., Martinez Tony, Giraud-Carrier Christophe. An instance level analysis of data complexity // Machine Learning. — 2014. — May. — Vol. 95, no. 2. — P. 225–256. — URL: https://doi.org/10.1007/s10994-013-5422-z.

[14] Zhang J., Mani I. KNN Approach to Unbalanced Data Distributions: A Case Study Involving Information Extraction // Proceedings of the ICML'2003 Workshop on Learning from Imbalanced Datasets. — 2003.