Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Кафедра системного программирования

Бережных Алексей Владимирович

Ранжирование найденных дубликатов кода в Intellij IDEA

Курсовая работа

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

Санкт-Петербург
2019

Оглавление

Введение

Дублирование кода – одна из наиболее часто встречающихся проблем в разработке программного обеспечения [2]. К основным последствиям дублирования кода можно отнести следующие:

- большое количество кода затрудняет его понимание – программисту становится тяжело уловить разницу в повторяющихся участках кода и понять назначение того или иного его фрагмента;

- дублирование кода нарушает общепринятые принципы разработки программного обеспечения: избыточность кода приводит к повышению затрат на его обслуживание, так как любое изменение во фрагменте кода необходимо применить ко всем его дубликатам, из-за чего потраченное на внесение изменений и последующее тестирование кода время увеличивается пропорционально количеству дубликатов, а пропуск хотя бы одного дубликата может привести к возникновению программных ошибок.

Таким образом, редуцирование количества дубликатов повышает общее качество кода и уменьшает время, необходимое на разработку и сопровождение программного обеспечения.

В IntelliJ IDEA Ultimate есть инструменты для поиска дубликатов кода и последующего применения к ним рефакторинга [3].

Недостатком данного инструментария является то, что пользователю отображается полный список групп дубликатов, включающий в себя группы, заранее не поддающиеся рефакторингу. По этой причине пользователям приходится просматривать список всех групп дубликатов самостоятельно. Проблема состоит в том, что применение тривиального метода фильтрации на основе встроенного инструмента рефакторинга неприемлемо по времени работы, так как, обычно, поиск дубликатов применяется к проектам с объёмной кодовой базой (более 500мб).

Необходим инструмент, фильтрующий список найденных групп дубликатов и показывающий пользователю лишь те, к которым возможно

применение встроенных средств рефакторинга, но работающий эффективнее тривиального метода. Там, где медленно работают классические алгоритмы, показывают своё успешное применение методы машинного обучения. В частности, это применимо и к программной инженерии:

- интеллектуальный поиск ошибок в исходном коде [5];
- автоматическое выделение метода [1];
- деобфускация исходного кода [7];
- подсказка имен классов и методов на основе контекста [6].

Поэтому предлагается рассмотреть данную задачу как задачу бинарной классификации групп найденных клонов на возможность применения к ним встроенного инструмента рефакторинга. Для обучения классификатора требуется большой набор данных, содержащих группы дубликатов, к которым применим и неприменим рефакторинг. В дальнейшем каждый дубликат преобразуется в векторное представление, а полученный набор векторов и используются для обучения модели классификации.

Решение данной задачи позволит разработчикам, использующим IntelliJ IDEA Ultimate, ускорить процесс редуцирования количества клонов кода в их проектах.

1. Постановка задачи

Целью данной курсовой работы является найти эффективный способ фильтровать список выдаваемых дубликатов на возможность применения к ним инструмента выделения метода, используя методы машинного обучения. Для достижения поставленной цели необходимо:

- изучить средства поиска и рефакторинга клонов кода в IntelliJ IDEA Ultimate;
- собрать данные, необходимые для обучения модели классификации групп клонов кода;
- выбрать и реализовать способ векторизации дубликатов кода;
- сравнить эффективность выбранного метода векторизации с аналогами, используемыми в курсовой работе студента 343 группы Максима Шамрая.

2. Обзор

2.1. Инструментарий IntelliJ IDEA Ultimate

2.1.1. Инструмент поиска дубликатов кода

В IntelliJ IDEA Ultimate содержится встроенный инструмент для поиска дубликатов, работающий по принципу хэширования поддеревьев AST и использующий семантическое дерево разбора кода для точной проверки типов переменных и сигнатур методов, который можно применить ко всему проекту на Java или отдельной его части. Результатом работы инструмента является выдаваемый пользователю список групп дубликатов, каждая из которых содержит два или более дубликата кода первого, второго или третьего типа. Результат работы инструмента представлен на рис. 1.

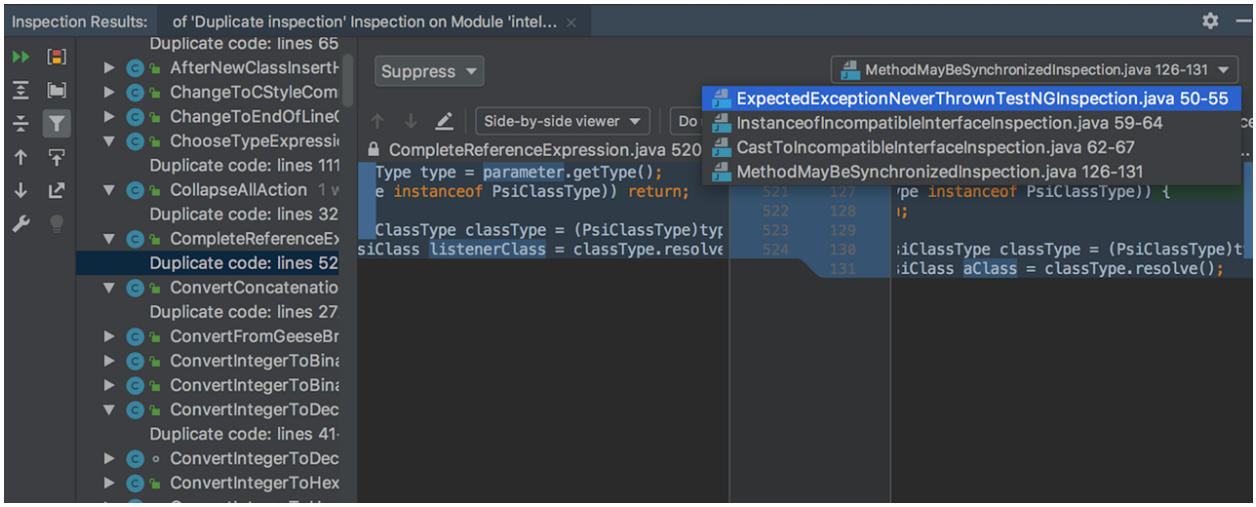

Рис. 1: Список найденных групп клонов кода

2.1.2. Инструмент рефакторинга дубликатов кода

После получения списка всех найденных групп дубликатов, встроенный инструмент рефакторинга дубликатов кода анализирует выбранную пользователем группу из списка на возможность выделить код дубликатов в отдельную функцию для дальнейшего переиспользования, после чего предлагает заменить дубликаты на её вызов. Пользователь

может выбрать название для функции и имена параметров, после чего выделенная функция добавляется как статическая в класс, содержащий первый дубликат из группы.

2.2. Данные для обучения модели

IntelliJ IDEA Ultimate предоставляет возможность сохранить список найденных групп дубликатов в файле формата XML, содержащем следующую информацию:

- путь к файлу исходного кода каждого из дубликатов;
- позицию начала и конца фрагмента кода, содержащего дубликат, относительно начала указанного файла.

Минусом получаемых данных является отсутствие разметки – информации о возможности применения к группе дубликатов инструмента рефакторинга.

2.3. Векторное представление программного кода

Для обучения модели классификации дубликаты кода из набора обучающих данных должны быть представлены в векторном виде. Существуют различные методы векторизации программного кода, по-разному учитывающие структуру кода, его семантический контекст, внешнюю и внутреннюю связность. К основным методам векторизации кода относятся:

- представление на основе токенов и N-грамм;
- свёртка AST, полученного из кода;
- графы потока управления и потока данных;
- промежуточное представление на основе байт-кода.

3. Описание реализации

3.1. Инструмент для автоматического сбора данных

Для обучения модели машинного обучения требуется большое количество данных, которое сложно собрать вручную. Поэтому в рамках данной курсовой работы был разработан инструмент[1] для автоматического сбора данных о дубликатах из проектов, открытых в IntelliJ IDEA.

3.1.1. Принцип работы инструмента

Инструмент написан на C# и использует библиотеку[2] для тестирования графического интерфейса, чтобы управлять окном IntelliJ IDEA. Алгоритм использования программы следующий:

- пользователь запускает поиск групп дубликатов в некотором проекте, после чего сохраняет список найденных групп в файл и запускает инструмент;
- инструмент принимает на вход список найденных дубликатов из файла и подключается к окну IntelliJ IDEA с открытым в нём списком дубликатов;
- программа эмулирует действия пользователя, поочерёдно выбирая группы дубликатов, ожидая от IDEA информации о возможности применения рефакторинга к выбранной группе и отмечая в файле полученную информацию (exp = 1, если можно применить, exp = 0, если нет) о выбранной группе.

3.1.2. Собранные данные

С помощью данного инструмента были собраны данные (рис. 2) из нескольких популярных репозиториев на Java:

[1] `https://github.com/DedSec256/Medooza.NET`

[2] `https://github.com/TestStack/White`

- LWJGL[3], LWJGL3[4] – библиотеки для работы с графикой;
- jUnit5[5] – библиотека для тестирования кода на Java;
- Jenkins[6] – система непрерывной интеграции;
- Spring[7] – фреймворк для разработки веб-приложений;
- RxJava[8] – библиотека для асинхронных и событийно-ориентированных приложений.

Название проекта	Обьём (Строк кода)	Всего групп дубликатов	Нельзя выделить	Можно выделить
LWJGL3	388199	3467	2505	962
LWJGL	72201	851	732	119
Junit5	68686	979	861	118
Jenkins	157361	1877	1464	413
Spring	642204	8718	6507	2211
RxJava	278931	5077	3352	1725
		20969	**15421**	**5548**

Рис. 2: Статистика по собранным данным

3.2. Векторизация дубликатов

В рамках данной курсовой работы был реализован метод векторизации кода на основе свёртки AST с использованием word2vec[9] и рекуррентной нейронной сети Tree-LSTM [4]. Данный метод векторизации позволяет учитывать структуру кода и его внутренний контекст, а также демонстрирует хорошие результаты в задаче поиска дубликатов кода. Выбранный метод векторизации работает следующим образом:

[3] https://github.com/LWJGL/lwjgl
[4] https://github.com/LWJGL/lwjgl3
[5] https://github.com/junit-team/junit5
[6] https://github.com/jenkinsci/jenkins
[7] https://github.com/spring-projects/spring-framework
[8] https://github.com/ReactiveX/RxJava
[9] https://code.google.com/archive/p/word2vec/

- с использованием Eclipse JDT[10] выделяется AST из кода фрагмента;
- каждый тип и содержимое узла AST векторизуются с помощью word2vec:
  - размер векторного представления для типа и содержимого узла – 200;
  - из полученных векторов заранее составляется словарь;
- AST нормализуется;
- в каждом узле заменяется тип и содержимое узла на его векторное представление из словаря;
- полученное дерево сворачивается с помощью рекуррентной нейросети Tree-LSTM, обученной авторами на датасете BigCloneBench[11].

3.3. Нормализация AST

Для эффективного использования полученного AST реализована его нормализация:

- из него удалены идентификаторы и литералы – чтобы убедиться, что нейронная сеть не зависит от них;
- для него строится эквивалентное бинарное дерево: если у родительской вершины более двух дочерних, она рекурсивно разбивается на две, каждая из которых будет родительской для соответсвующей половины дочерних.

Пример нормализации AST для фрагмента кода можно подробнее рассмотреть на рис. 3.

[10] `https://www.eclipse.org/jdt`

[11] `https://github.com/clonebench/BigCloneBench`

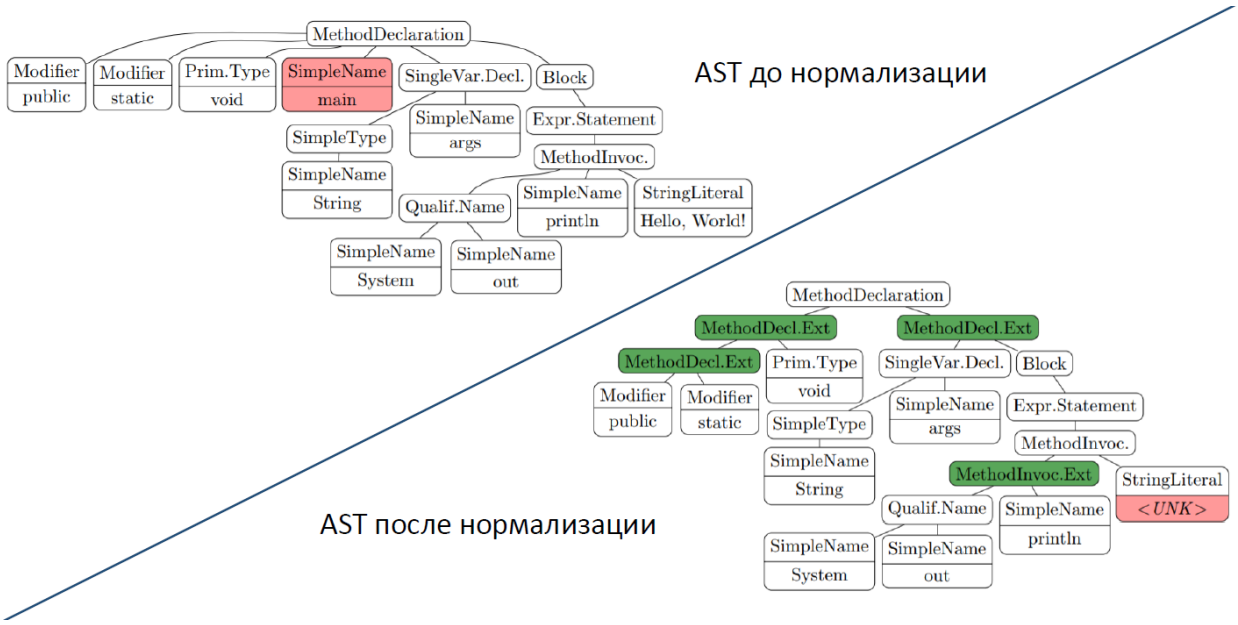

Рис. 3: Пример нормализации AST

3.4. Процесс обучения

На основе выбранного метода векторизации дубликатов и полученных данных необходимо обучить классификаторы групп дубликатов. Для обучения моделей был выбран фреймворк Hyperopt Sklearn[12], поскольку он позволяет:

- автоматически подбирать гиперпараметры и выбить лучший классификатор в процессе обучения;
- содержит 9 различных моделей классификации:
  1. SVC.
  2. LinearSVC.
  3. KNeightborsClassifier.
  4. RandomForestClassifier.
  5. SGDClassifier.
  6. MultinomialNB.
  7. BernoulliRBM.

[12] `https://github.com/hyperopt/hyperopt-sklearn`

8. ColumnKMeans.

Для обучения и оценки полученных результатов использован готовый инструментарий[13], написанный в курсовой работе Максима Шамрая.

[13] https://github.com/MaxVortman/IDEA-code-clones

4. Оценка полученных результатов

4.1. Используемые метрики

Для сравнения с аналогами из курсовой работы Максима Шамрая были выбраны следующие метрики:

- $FNR = \frac{FN}{n}$;
- $FPR = \frac{FP}{n}$;
- $recall = \frac{TP}{TP+FN}$;
- $presicion = \frac{TP}{TP+FP}$;
- $accuracy = \frac{TP+TN}{n}$;
- $ROCAUC = \int_0^1 \gamma dt, \gamma(t) = (FPR(t); TPR(t))$.

Где:

- n – общее число групп дубликатов в датасете;
- FN – группы, неверно помеченные моделью как не выделяемые;
- FP – группы, неверно помеченные моделью как выделяемые;
- TN – группы, верно помеченные моделью как не выделяемые
- TP – группы, верно помеченные моделью как выделяемые
- ROC AUC – площадь под кривой ошибок; метрика, не зависящая от выбранного порога бинаризации, используемого для классификации.

4.2. Сравнение с аналогами

Результаты аналогов взяты из работы Максима Шамрая, для сравнения с ними было подобрано значение порога бинаризации так, чтобы значение FPR равнялось 0.1

На рис. 4 видно, что реализованный в данной работе метод векторизации ast-lstm (выделен зелёным) показал себя лучше тривиальных аналогов на основе токенов tf-idf и bow, однако по всем метрикам хуже fast-text.

	threshold	FNR	FPR	accuracy	precision	recall	ROC AUC
tf-idf	0.5	0.19	0.1	0.71	0.76	0.62	0.8
bow	0.52	0.18	0.1	0.72	0.76	0.65	0.82
code2vec	0.51	0.17	0.1	0.72	0.76	0.66	0.83
ast-lstm	0.57	0.16	0.1	0.73	0.77	0.68	0.83
fastText	0.62	0.15	0.1	0.75	0.78	0.72	0.83

Рис. 4: Таблица сравнения полученных результатов с аналогами

Заключение

Полученные результаты

В ходе выполнения данной работы были достигнуты следующие результаты:

- изучены средства поиска и рефакторинга клонов кода в IntelliJ IDEA Ultimate;
- реализован инструмент для сбора данных из IntelliJ IDEA, с помощью которого был получен датасет, используемый в рамках данной курсовой работы;
- реализована техника векторизации кода дубликатов на основе свёртки AST кода дубликатов;
- посчитаны метрики для выбранного метода векторизации и произведено сравнение с аналогами.

Список литературы

[1] Danilo Silva Ricardo Terra Marco Tulio Valente. JExtract: An Eclipse Plug-in for Recommending Automated Extract Method Refactorings. — 2015. — URL: https://arxiv.org/pdf/1506.06086.pdf (дата обращения: 14.12.2018).

[2] Elmar Juergens Florian Deissenboeck, Hummel Benjamin. Code Similarities Beyond Copy Paste. — 2010. — URL: https://www.cqse.eu/publications/2010-code-similariti.. (дата обращения: 14.12.2018).

[3] JetBrains. Analyzing Duplicates. — 2018. — URL: https://www.jetbrains.com/help/idea/analyzing-duplica.. (дата обращения: 14.12.2018).

[4] L. Büch A. Andrzejak. Learning-based recursive aggregation of abstract syntax trees for code clone detection. — 2019. — URL: https://pvs.ifi.uni-heidelberg.de/fileadmin/papers/2019/Buech-Andrzejak-SANER2019.pdf (дата обращения: 25.02.2019).

[5] Michael Pradel Koushik Sen. DeepBugs: A Learning Approach to Name-based Bug Detection. — 2018. — URL: http://software-lab.org/publications/DeepBugs_arXiv_1.. (дата обращения: 14.12.2018).

[6] Miltiadis Allamanis Earl T. Barr Christian Bird Charles Sutton. Suggesting Accurate Method and Class Names. — 2015. — URL: https://homepages.inf.ed.ac.uk/csutton/publications/a.. (дата обращения: 14.12.2018).

[7] Veselin Raychev Martin Vechev Andreas Krause. Predicting Program Properties from Big Code. — 2015. — URL: https://dl.acm.org/citation.cfm?id=2677009 (дата обращения: 14.12.2018).