Санкт-Петербургский государственный университет

Кафедра системного программирования

Шамрай Максим Борисович

# Фильтрация поисковой выдачи дубликатов кода в IntelliJ IDEA

Курсовая работа

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

Санкт-Петербург
2019

# Оглавление

# Введение

Дубликаты исходного кода (клоны кода, дубликаты кода) — это участки кода, имеющие похожую функциональность или повторяющие друг друга полностью. Считается, что наличие клонов кода не только завышают расходы на техническое обслуживание, но и чреваты ошибками, так как непоследовательные изменения в дубликатах могут привести к неожиданному поведению программы [3].

Дубликаты исходного кода разделяют на четыре типа [2]:

1. Фрагменты кода совпадают посимвольно, игнорируя комментарии и пробелы.

2. Аналогичные фрагменты кода за исключением различий в идентификаторах, литералах, типах, последовательности пробелов и комментариях.

3. Подобные фрагменты кода за исключением того, что некоторые операторы могут быть добавлены или удалены в дополнение к изменению идентификаторов, литералов, типов, последовательности пробелов или комментариев.

4. Фрагмента кода, которые выполняют одно и то же вычисление, но реализуются различными синтаксическими конструкциями.

IntelliJ IDEA Ultimate версии предоставляет средства, позволяющие найти дубликаты кода в Java проекте и попытаться заменить найденные фрагменты кода на вызов метода, созданного на основе дубликатов[1] (отметим, что подобного функционала нет в Community версии IntelliJ IDEA).

Не все найденные с помощью IntelliJ IDEA клоны можно выделить в метод, а наивно прогонять алгоритм выделения метода для каждого кандидата также не представляется возможным из-за довольно долгого времени его работы.

[1] https://www.jetbrains.com/help/idea/analyzing-duplicates.html

Тем не менее эту проблему можно решить рассматривая ее как задачу бинарной классификации, ведь в последние годы все чаще появляются новые применения машинного обучения в программной инженерии. Оно успело уже себя проявить в ряде задач, таких как

- Поиск ошибок в исходном коде
- Автоматическое выделение метода
- Деобфускация исходного кода
- Подсказка имен классов и методов

и в многих других.

Таким образом, необходимо разработать, используя машинное обучение, инструмент, позволяющий эффективно фильтровать предполагаемые дубликаты по возможности выделения их в метод.

## Постановка задачи

Целью данной курсовой работы является разработка инструмента фильтрации выходных данных встроенного средства поиска дубликатов в IntelliJ IDEA Ultimate по возможности выделения их в метод.

Для достижения этой цели в рамках работы были сформулированы следующие задачи.

1. Рассмотрение различных способов векторизации кода.
2. Применение методов машинного обучения к векторизованному коду.
3. Анализ результатов.

```
//первый фрагмент
if ( key_states[key_code_mapped] == state )
    repeat = true;
key_states[key_code_mapped] = state;
int key_int_char = character & 0xffff;
putKeyboardEvent(key_code_mapped, state,
                    key_int_char, nanos, repeat);

//=============================================

//второй фрагмент
if ( key_states[mapped_code] == state )
    repeat = true;
key_states[mapped_code] = state;
int key_int_char = character & 0xffff;
putKeyboardEvent(mapped_code, state,
                    key_int_char, nanos, repeat);
```

Рис. 1: Пример невыделяемых дубликатов

# 1. Обзор

## 1.1. Выделение дубликатов в IntelliJ IDEA Ultimate

### 1.1.1. Принцип работы

IntelliJ IDEA версии Ultimate имеет функциональность поиска фрагментов кода, предположительно являющихся клонами, на больших объемах Java кода. Также среда разработки предлагает выделить найденные дубликаты в переиспользуемый элемент.

Это не всегда является возможным. На рис. 1 представлены два фрагмента кода, определенные как дубликаты, но которые не могут извлечь встроенные средства, а так как IntelliJ IDEA Ultimate проприетарное программное обеспечение, мы не можем установить причины этого. Изображенные два фрагмента идентичны с точностью до переменной, то есть это дубликаты 2 типа.

Кроме того, большую часть выявленных дубликатов нельзя выделить в метод с целью переиспользования, далее мы покажем это на

```
<duplicate cost=''26'' hash=''0''>
    <fragment
        file=''file://$PROJECT_DIR$/src/java/org/
            lwjgl/test/opengles/MappedIndexedVBOTest.java''
        line=''238'' start=''7532'' end=''7789''/>
    <fragment
        file=''file://$PROJECT_DIR$/src/java/org/
            lwjgl/test/opengles/FullScreenWindowedTest.java''
        line=''248'' start=''6723'' end=''6984''/>
</duplicate>
```

Рис. 2: Пример разметки дубликатов в XML-файле

наборе данных.

### 1.1.2. Формат выходных данных

После окончания поиска возможен экспорт результатов в XML и HTML файлы, содержащие группы фрагментов, расположенных в одном или в разных файлах кода, а также их расположение, включая начало и конец фрагмента. Эти данные могут быть использованы для фильтрации результатов по различным параметрам, например, для фильтрации предполагаемых дубликатов по возможности выделения их в переиспользуемый элемент, что улучшило бы работу поиска клонов с точки зрения пользователя.

На рис. 2 представлена XML-разметка дубликатов, рассмотренных ранее. В этих файлах сохраняется относительный путь к файлу, а также строку начала стартовый и конечный символ каждого из фрагментов. Это означает, что мы располагаем всем необходимым для того, чтобы вырезать фрагмент из файла.

## 1.2. Модели представления исходного кода

По мере развития машинного обучения возникает большой интерес к тому, чтобы рассматривать программы как данные для алгоритмов обучения. Векторизация кода – это важный этап, который позволяет

нам перейти от кода к собственно методам машинного обучения.

Цель состоит в том, чтобы построить модели для изучения промежуточных, не обязательно интерпретируемых человеком, кодировок кода, и именно от этой модели будет наибольший вклад в точность решения поставленной задачи, так как от нее зависит передача выразительности кода. Они называются моделями представления исходного кода.

В зависимости от подхода, на вход модель представления исходного кода может принимать [4]:

- лексемы (идентификаторы, ключевые слова, фигурные скобки и т.д.)
- признаки (например, количество приватных/публичных полей, n-граммы и т.д.)
- абстрактное синтаксическое дерево (также возможны его модификации)

А результатом ее работы является вектор, представляющий входной исходный код.

### 1.2.1. Подход code2vec

code2vec [5] — опубликованный недавно метод векторизации кода на языке Java, применяющийся для решения задачи предсказания имен методов[2] и демонстрирующий лучшие результаты.

Авторы обучали свою модель[3] на 14 миллионах методах и ее размер составил 1.4 GB, что может быть проблемой при ее интегрировании в IntelliJ IDEA в виде плагина.

### 1.2.2. Bag-of-words

Bag-of-words — модель представления текста, получившая широкое применение в обработке естественного языка и информационном поиске, а также один из наиболее простых методов представления текстов, в

[2]https://code2vec.org/

[3]https://github.com/tech-srl/code2vec#downloading-a-trained-model-14-gb

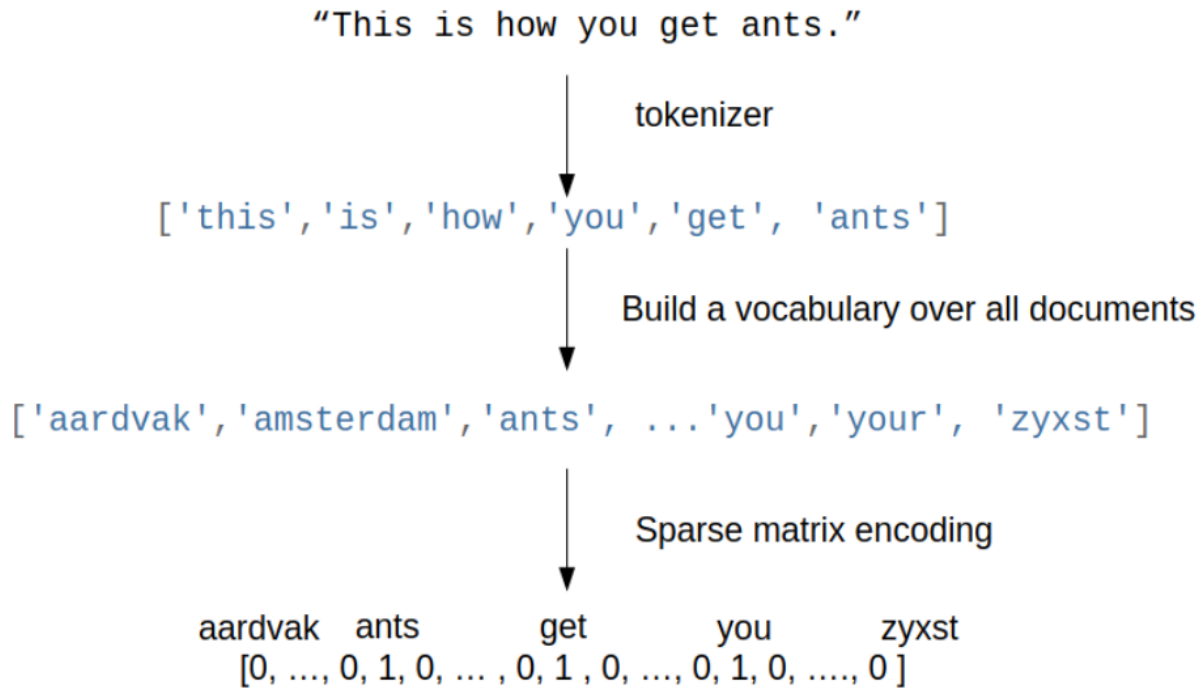


Рис. 3: Представление с помощью bag-of-words

котором игнорируется порядок токенов в рассматриваемом документе, то есть, если два документа отличаются только порядком токенов, то они будут иметь одинаковые векторы.

В bag-of-words каждый отрывок текста представляется вектором, где каждая i-ая компонента является счетчиком встречаемости i-ого токена в рассматриваемом отрывке. На рис. 3 представлен пример работы.

Недостаток этого подхода в возможном совпадении векторов разных отрывков текста, так как порядок слов может играть важную семантическую роль. Кроме того, выходной вектор bag-of-words имеет размерность словаря, что приводит к довольно объемным данным и необходимости сокращения размерности.

### 1.2.3. TF-IDF

TF-IDF (Term Frequency – Inverse Document Frequency) – модификация bag-of-words, главная идея которой заключена в том, что токены

в документе имеют разный вес, и если токен встречается в малом количестве документов, то он важнен для них.

TF — доля документов, в которых присутствует токен (1). IDF — инверсия частоты, с которой некоторый токен встречается во всех документах коллекции (2). Вес токена в документе вычисляется как произведение TF и IDF (3).

(1) $tf(t,d) = \frac{n_t}{\sum_k n_k}$, где $n_t$ есть число вхождений слова $t$ в документ, а в знаменателе – общее число слов в данном документе.

(2) $idf(t,D) = log\frac{|D|}{|\{d_i \in D : t \in d_i\}|}$, где $|D|$ – число документов в коллекции, $|\{\, d_i \in D \mid t \in d_i \,\}|$ – число документов из коллекции $D$, в которых встречается $t$.

(3) $tfidf(t,d,D) = tf(t,d)idf(t,D)$

Теоретически этот подход должен давать лучшие результаты по сравнению с bag-of-words, при этом имея те же недостатки.

### 1.2.4. fastText

fastText[1] – библиотека для быстрой классификации текста, разработанная в Facebook Research, которая работает быстро в основном благодаря своей простоте.

fastText представляет из себя нейронную сеть с одним скрытым полносвязным слоем, на вход которой подаются n-граммы, а перед выходным слоем применяется обобщенная логистическая функция – softmax (см. рис. 4). Сеть обучается, минимизируя функцию потерь (4), где $x_n$ – нормализованные входные данные для n-ого документа, $y_n$ – истинный тэг n-ого документа, $f$ – softmax, $A$ и $B$ – матрицы весов.

(4) $L = -\frac{1}{N}\sum_{n=1}^{N} y_n log(f(BAx_n))$

Преимущество fastText заключается в быстрой скорости работы, в простоте использования, а также в комплексной векторизации и классификации текста.

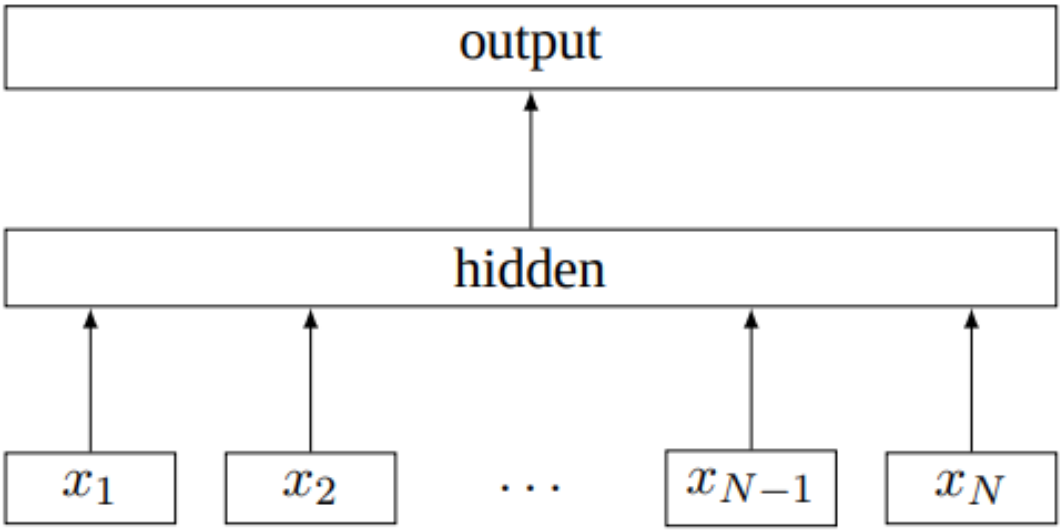


Рис. 4: Архитектура fastText, где $x_i$ – n-граммы

## 1.3. Бинарная классификация

Бинарная или двоичная классификация – это задача разбиения исходного множества на две группы. В нашем же случае надо разбить множество дубликатов на те, которые возможно выделить в метод, и на те, которые невозможно выделить.

### 1.3.1. Фреймворк sklearn

sklearn[4] – простой и эффективный инструмент для анализа данных и машинного обучения. Представляет из себя библиотеку для языка Python и является самым популярным на сегодняшний день. Основное преимущество в представлении множества различных алгоритмов и открытого исходного кода.

### 1.3.2. Фреймворк hyperopt-sklearn

hyperopt-sklearn[5] – библиотека языка Python, основанная на sklearn и созданная для решения проблемы выбора "правильного" классифика-

[4]https://scikit-learn.org/stable/index.html
[5]http://hyperopt.github.io/hyperopt-sklearn/

тора. Авторы считают, что процесс выбора классификатора и подбор гиперпараметров можно и нужно автоматизировать. Данный фреймворк позволяет указать валидационную функцию потерь, согласно которой будет выбран лучший метод бинарной классификации.

## 1.4. Выводы

Таким образом, рассмотрено множество способов векторизации и два фреймворка для последующей классификации. Каждый из них имеет свои достоинства и недостатки. В рамках данной работы было принято решение применить каждый из них.

# 2. Набор данных

Так как область данной курсовой работы довольно узкая, то готового набора данных у нас не было. Тем не менее, коллега разработал инструмент[6] для сбора данных и с его помощью составил набор данных из 19990 групп дубликатов кода на языке Java.

## 2.1. Случайная выборка

К сожалению, так как большинство клонов кода нельзя было выделить, возникла проблема преобладания дубликатов без возможности выделения в метод (14560 дубликатов из 19990). В связи с чем было принято решение использовать технику случайной выборки, которая заключается в случайном удалении представителей из преобладающего класса с целью выровнять мощности классов.

Также существует другие техники борьбы с этой проблемой, например – случайная избыточная выборка, идея которой основана на создании нескольких копий элементов класса меньшинств. Мы решили остановиться на случайной выборке, так как кроме мощности классов мы ничего не теряем, а увеличить количество данных можно всегда.

[6] https://github.com/DedSec256/Medooza.NET

# 3. Векторизация кода

Векторизация кода – важнейший этап анализа кода методами машинного обучения. В этой курсовой работе мы использовали такие модели представления исходного кода как code2vec, bag-of-words и TF-IDF.

Полученные вектора подавались на вход моделям двоичной классификации, и отображался результат лучшей из них.

## 3.1. Среда и язык разработки

Весь код здесь и далее был написан на языке программирования Python в облачном сервисе Google Colab. Он дает возможность работать в популярной на сегодня среде исполнения Jupyter Notebook и предоставляет вычислительные мощности: Intel(R) Xeon(R) CPU @ 2.30GHz, 49Гб дискового пространства и 13Гб оперативной памяти. Также была возможность ускорить свои вычисления с помощью GPU Tesla K80 и TPU.

## 3.2. XML-анализатор

В качестве структуры данных был написан класс Duplicate, который в конструкторе принимает директорию проекта и узел дубликата в XML-файле, после чего заполняет свои поля fragments и exp. Первое поле содержит текстовые фрагменты, а второе возможность выделения дубликата в переиспользуемый элемент (см. рис. 5)

В итоге мы получаем список объектов класса Duplicate, что позволяет нам переходить непосредственно к векторизации.

## 3.3. Векторизация с помощью code2vec

Так как процесс тренировки модели занимает довольно много времени, а авторы предоставляют уже натренированную модель на 14 миллионах методах, было принято решение использовать ее, несмотря на то,


Рис. 5: Схема анализа XML файла дубликатов, fragments — дублирующиеся фрагменты кода, exp — флаг выделяемости в метод

что она применяется авторами на другой задаче – предсказание имен методов.

Тем не менее, авторы не подумали про удобство пользователя и сделали довольно ограниченный интерфейс, согласно которому можно было либо провекторизовать целый файл с кодом и получить на выходе файл с векторами, которые соответствовали бы отрывкам кода из исходного файла, либо провекторизовать один фрагмент кода, записанный во входной файл, и получить вектор на консоль, после чего повторить процесс.

Нами был выбран второй способ, так как в этом случае модель загружалась в память лишь в начале работы, в отличие от первого, где загрузка происходила при каждом запуске, что было нецелесообразно, так как процесс загрузки натренированной модели занимал много времени.

Для этого было обращено внимание на linux-утилиту для автоматизации и тестирования expect, а точнее на ее порт в язык Python – pexpect. Этот инструмент предоставляет удобный интерфейс для взаимодействия с терминалом, а именно запись и чтение, не прерывая процесс выполнения программы в терминале.

Кроме того, некоторые дубликаты состояли из нескольких методов, и code2vec возвращал несколько векторов для них. Для таких клонов считался вектор, состоящий из средних значений всех полученных векторов данного фрагмента.

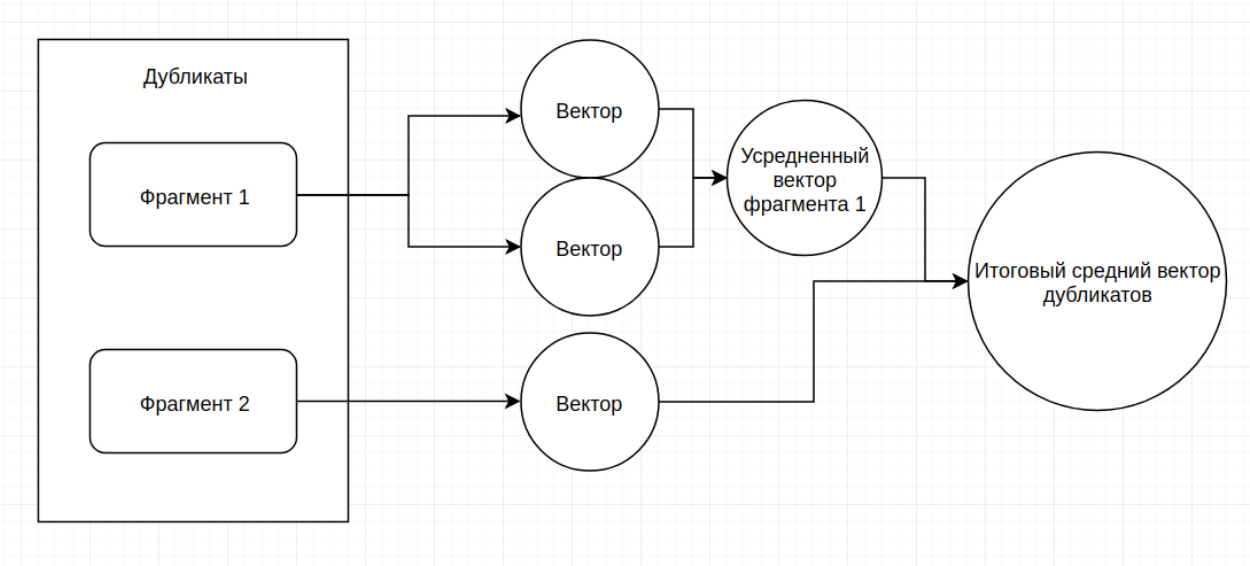


Рис. 6: Векторизация дубликатов с помощью code2vec

После чего также вычислялось среднее значение по всем векторам фрагментов, составляя итоговый вектор дубликата размерностью 385 (см. рис. 6).

Таким образом, для векторизации 5 тысяч групп дубликатов было затрачено 3 часа 30 минут в среде исполнения Google Colab, а на компьютере с 8Гб оперативной памяти, Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz и с Ubuntu 18.04 было затрачено более 8 часов, не завершив векторизацию.

## 3.4. Векторизация с помощью bag-of-words

Основой для векторизации при помощи bag-of-words была выбрана небезызвестная библиотека scikit-learn[7], а точнее ее класс CountVectorizer.

Он предоставляет возможность выбора диапазона n-грамм (bag-of-words – это частный случай n-грамм), а также с его помощью можно в одну строку обучить репрезентативную модель и получить вектора для набора данных.

На вход было решено подавать конкатенированные фрагменты дубликатов, чтобы в векторе отразилась их идентичность.

[7]https://scikit-learn.org/

В итоге была получена разреженная матрица размером 19990x52736, где первая компонента – это количество групп дубликатов, а вторая – размерность векторов/мощность словаря. После чего была сокращена размерность векторов до 65 с незначительной потерей точности, используя класс из sklearn – TruncatedSVD.

## 3.5. Векторизация с помощью TF-IDF

Для векторизации посредством TF-IDF также был выбран класс TfidfTransformer библиотеки scikit-learn.

С его помощью можно посчитать TF-IDF из входных n-грамм. Для чистоты эксперимента TF-IDF была посчитана на векторах bag-of-words.

## 3.6. Векторизация и бинарная классификация с помощью fastText

Так как fastText является фреймворком классификации текста, для него не требуется отдельно векторизовать код, он сам и векторизирует и классифицирует.

Для эксперимента был выбран порт fastText в язык Python, обучение велось с параметрами по умолчанию:

- Количество эпох – 5
- Размер n-грамм – 1
- Размерность скрытого слоя – 100
- Размер шага – 0.1

Все это можно будет подобрать, чтобы получить лучший результат, но сначала мы оценим работу с параметрами по умолчанию.

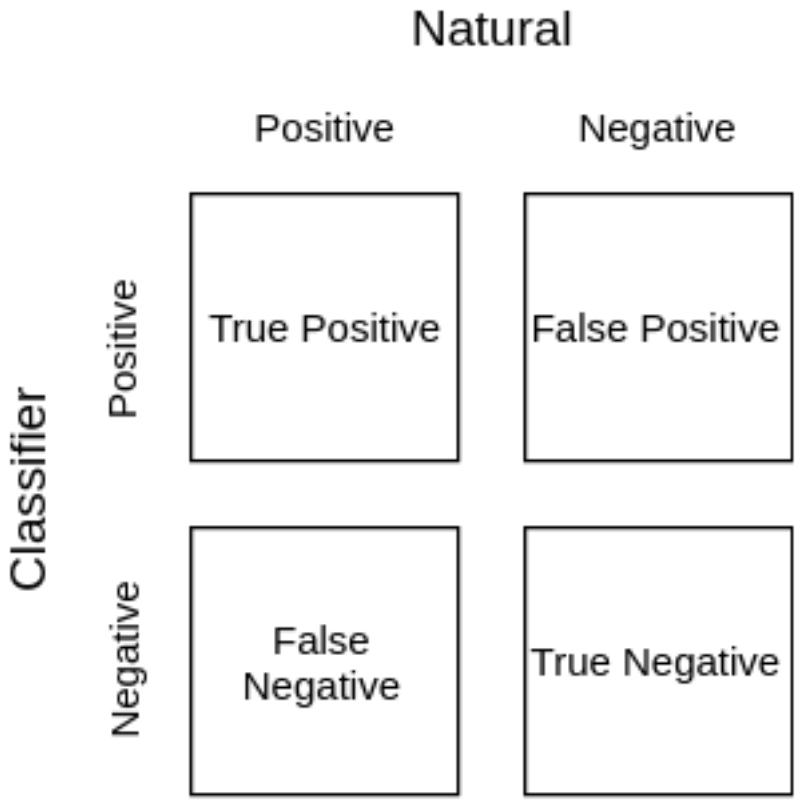


Рис. 7: Confusion matrix

# 4. Критерии оценивания

Поставленную перед нами задачу фильтрации можно переформулировать следующим образом. Требуется определить дубликаты, которые поддаются извлечению в метод, при этом как можно сильнее снизить процент ложно выделяемых дубликатов, то есть тех, которые на самом деле нельзя выделить, но они попали в положительно определенные. Говоря языком машинного обучения, нам нужно снизить количество False Positive (см. рис. 7), так как после классификации все положительно определенные можно проверить достоверным методом – инструментом выделения в метод в IntelliJ IDEA Ultimate (так как он работает довольно долго запускать его сразу на всем наборе дубликатов является нецелесобразным).

Таким образом, получается некое подобие воронки, которая демонстрирует фильтрацию по возможности выделения в метод дубликатов. В верхней ее части весь набор дубликатов, в середине дубликаты после бинарной классификации, но среди которых еще есть некоторый

процент False Positive, а в нижней части остаются только дубликаты, которые точно можно выделить, и их первыми будет видеть пользователь.

## 4.1. Валидационная функция потерь

Для выбора наиболее эффективной модели машинного обучения, обучаемой на векторном представлении кода, было решено использовать валидационную функцию потерь (5), где FNR (False Negative Rate) и FPR (False Positive Rate) – отношения количества False Negative и False Positive соответственно с мощностью набора данных.

(5) $L = FNR + 10 \times FPR$

Эта функция потерь позволила выбрать наиболее подходящий способ бинарной классификации для нашей задачи. Так как весовой коэффициент перед FPR показывает, что мы добиваемся снижения количества False Positive.

Кроме того, были испробованы весовые коэффициенты при FRP 100 и 1000, которые не привели к лучшим результатам, выбирая тот же способ бинарной классификации.

## 4.2. Используемые метрики

Для оценки качества моделей было посчитано несколько метрик (см. рис. 7):

$$FNR = \frac{FN}{n},\ FPR = \frac{FP}{n}$$

$$accuracy = \frac{TP + TN}{n},\ precision = \frac{TP}{TP + FP}$$

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

Кроме этих простых метрик, также посчитаем более сложную. ROC AUC (ROC = Receiver Operating Characteristic, AUC = Area Under the Curve) – площадь под кривой ошибок. Кривая ошибок построена в координатах (FPR, TPR), то есть кривая ошибок – это кривая зависимости

TPR от FPR при варьировании порога для бинаризации. Еще ее часто называют ”честной точностью”, так как она не зависит от выбранного порога.

# 5. Оценка результатов

Проведённые эксперименты показали, что при задании стандартного порога бинаризации ($threshold = 0.5$) результаты невозможно сравнивать, так как значения метрик FNR и FPR разбросаны (см. таблицу 1).

Таблица 1: Результаты экспериментов на различных метриках с $threshold = 0.5$

| | fnr | fpr | accuracy | precision | recall |
|---|---|---|---|---|---|
| tf-idf | 0.19 | 0.1 | 0.71 | 0.76 | 0.62 |
| bow | 0.13 | 0.14 | 0.73 | 0.73 | 0.75 |
| bow + TruncatedSVD | 0.16 | 0.13 | 0.71 | 0.73 | 0.68 |
| code2vec | 0.07 | 0.2 | 0.73 | 0.69 | 0.85 |
| fastText | 0.11 | 0.14 | 0.76 | 0.75 | 0.79 |

В связи с этим было принято решение подобрать значения порога так, чтобы значение FPR равнялось 0.1. Это позволило выбрать лучшую модель, опираясь только на значение FNR (чем меньше, тем лучше).

Таким образом, согласно таблице 2, лучшее решение показывает fastText.

Таблица 2: Результаты экспериментов на различных метриках с провалидированным threshold к значению $FPR = 0.1$

| | threshold | fnr | fpr | accuracy | precision | recall |
|---|---|---|---|---|---|---|
| tf-idf | 0.5 | 0.19 | 0.1 | 0.71 | 0.76 | 0.62 |
| bow | 0.52 | 0.18 | 0.1 | 0.72 | 0.76 | 0.65 |
| bow + TruncatedSVD | 0.503 | 0.20 | 0.1 | 0.69 | 0.75 | 0.60 |
| code2vec | 0.51 | 0.17 | 0.1 | 0.72 | 0.76 | 0.66 |
| fastText | 0.62 | 0.15 | 0.1 | 0.75 | 0.78 | 0.72 |

Кроме этого, для большей надежности и уверенности в выборе, была посчитана метрика ROC AUC (см. таблицу 3), которая также показала,

что лучшим решением из рассмотренных моделей является fastText.

Таблица 3: Значение метрики ROC AUC для различных моделей

| | tf-idf | bow | bow + TruncatedSVD | code2vec | fastText |
|---|---|---|---|---|---|
| roc auc | 0.80 | 0.82 | 0.81 | 0.83 | 0.83 |

# 6. Заключение

В рамках курсовой работы были получены следующие результаты:

- Реализована процедура векторизации исходного кода, основанная на репрезентативных моделях: TF-IDF, bag-of-words, code2vec.
- Обучены модели бинарной классификации, примененные к векторному представлению исходного кода.
- Сделан выбор лучших моделей, основанный на значениях валидационной функции потерь.
- Был применен к поставленной задачи фреймворк fastText.
- Посчитаны метрики и сделаны выводы относительно лучшего решения из рассмотренных.

В результате была проведена исследовательская работа по поиску эффективного метода фильтрации дубликатов по возможности выделения их в метод. Исходный код данной работы можно найти в GitHub репозитории[6].

# Список литературы


[1] Armand Joulin, Edouard Grave, Piotr Bojanowski, Tomas Mikolov. Bag of Tricks for Efficient Text Classification. — 2016. — Режим доступа: https://arxiv.org/pdf/1607.01759.pdf (дата обращения: 18.05.2019).

[2] Stefan Bellon, Rainer Koschke, Giulio Antoniol и др. Comparison and Evaluation of Clone Detection Tools. — 2007. — Режим доступа: https://ieeexplore.ieee.org/document/4288192 (дата обращения: 14.12.2018).

[3] Elmar Juergens, Florian Deissenbeck, Benjamin Hummel. Code Similarities Beyond Copy Paste. — 2010. — Режим доступа: https://www.cqse.eu/publications/2010-code-similarities-beyond-copy-paste.pdf (дата обращения: 14.12.2018).

[4] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, Charles Sutton. A Survey of Machine Learning for Big Code and Naturalness. — 2018. — Режим доступа: https://arxiv.org/pdf/1709.06182.pdf (дата обращения: 18.05.2019).

[5] Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav. code2vec: Learning Distributed Representations of Code. — 2018. — Режим доступа: https://arxiv.org/pdf/1803.09473.pdf (дата обращения: 14.12.2018).

[6] Исходный код данной курсовой работы. — 2019. — Режим доступа: https://github.com/InteIIigeNET/IDEA-code-clones (дата обращения: 19.05.2019).