

Санкт-Петербургский государственный университет

Кафедра системного программирования

Курбатова Зарина Идиевна

Автоматическая рекомендация имен методов в IntelliJ IDEA

Выпускная квалификационная работа

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

Рецензент:
аналитик ООО "Интеллиджей Лабс" Поваров Н.И.

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering

Zarina Kurbatova

Automatic recommendation of method names in IntelliJ IDEA

Graduation Thesis

Scientific supervisor:
Candidate of Engineering Sciences Timofey Bryksin

Reviewer:
analyst, IntelliJ Labs Co. Ltd. Nikita Povarov

Saint-Petersburg
2019

Оглавление

Введение	4
1. Обзор	6
1.1. Алгоритмы рекомендации имен методов	6
1.1.1. ConvAttention	6
1.1.2. Code2vec	7
1.1.3. Code2seq	9
1.2. Метрики для оценки качества предсказания	11
1.3. Компоненты IntelliJ Platform	13
1.3.1. PSI	13
1.3.2. Inspection	15
1.4. Выводы из обзора	15
2. Разработанное решение	17
2.1. Архитектура плагина	17
2.2. Алгоритм работы инспекции	20
2.3. Модуль сбора статистики	23
3. Апробация	24
Заключение	26
Список литературы	27

Введение

При разработке программного обеспечения большое внимание уделяется качеству кода. Важно, чтобы программный код не только решал поставленную задачу, но и был понятен другим разработчикам [5]. Со временем кодовая база растет, каждый раз вникать в логику классов и методов затруднительно, поэтому стоит ответственно подходить к именованию всех идентификаторов, особенно при разработке API (Application Program Interface) библиотек.

В последние годы активно развиваются и внедряются во многие сферы методы машинного обучения. Анализ программного кода является интересной областью для исследований [15], поскольку содержит в себе огромное количество данных и множество различных задач. Например, обнаружение ошибок [7], поиск плагиата [8], предсказание имен идентификаторов [13].

Исследователи разрабатывают алгоритмы предсказания имени идентификатора, основываясь на контексте вокруг него. Эта задача интересна и актуальна, поскольку имена идентификаторов влияют на читаемость кода. Было замечено, что низкое качество имен приводит к дефектам в работе программ [9].

Для повышения качества кода разработчики используют инструменты для анализа кода в интегрированных средах разработки. Некоторые из них работают как автоматические инспекции: в фоновом режиме анализируют код и предоставляют подсказки о том, что можно улучшить. Например, поиск дублирующихся фрагментов кода, поиск опечаток, упрощение синтаксических конструкций. С помощью автоматических инспекций можно значительно упростить работу программистов, частично переложив заботу о поиске недостатков на среду разработки.

На данный момент для среды разработки IntelliJ IDEA нет плагинов, решающих задачу автоматической рекомендации имен методов. В данной работе предлагается решение в виде автоматической инспекции, которая рекомендует имена для методов.

Постановка задачи

Целью работы является разработка плагина для IntelliJ IDEA, который автоматически рекомендует имена для методов. Для достижения этой цели были поставлены следующие задачи:

- провести обзор предметной области;
- выбрать алгоритм рекомендации имен методов;
- разработать архитектуру и реализовать плагин;
- выполнить апробацию.

1. Обзор

1.1. Алгоритмы рекомендации имен методов

Извлечение семантических свойств кода с целью предсказания имени метода, основываясь на его содержимом, является интересной задачей для исследований. В этой главе мы рассмотрим несколько существующих подходов к ее решению. В частности, нас интересуют наилучшим образом себя показавшие недавние решения с открытым исходным кодом.

1.1.1. ConvAttention

В работе ConvAttention [1] описан подход к решению задачи реферирования программного кода. Принимая на вход тело метода, алгоритм генерирует его описание на естественном языке. Возможным применением такого подхода является автоматическая генерация документации к коду, другим применением является предсказание имени метода по его содержимому. В качестве предсказаний алгоритм способен генерировать неологизмы, слова, которых нет в тренировочном корпусе.

Авторы используют сверточную нейронную сеть с использованием механизма внимания [3]. Сверточные сети — это вид нейронных сетей, который состоит из нескольких чередующихся слоев обработки. Идея состоит в разбиении входных данных на отдельные блоки, применении к каждому блоку функции свертки и получении на выходе карты признаков. Использование механизма внимания позволяет обнаружить наиболее значимые фрагменты входных данных путем генерации соответствующих весов для них.

Для предсказания неологизмов авторы основываются на том, что имена методов чаще всего представляют собой конкатенацию нескольких слов. Например, «getLocation» состоит из двух слов: «get» и «Location». Авторы делят имена идентификаторов на составные части, в результате получается множество слов, которые затем используются для создания новых имен идентификаторов. Для генерации неологиз-

ма перебирать все возможные комбинации слов невозможно, потому что существует бесконечное количество вариантов. Для решения этой проблемы авторы используют жадный алгоритм beam search [10], который ищет последовательность токенов с наибольшей вероятностью, на каждом шаге выбирая слово с наибольшей вероятностью.

Описываемый подход демонстрирует хорошие результаты в рамках одного проекта, когда обучение и тренировка проходят на одних и тех же данных, однако он плохо масштабируется. Это является недостатком подхода, поскольку разрабатываемое в рамках данной работы решение не предполагает переобучение модели на проектах пользователей.

1.1.2. Code2vec

В работе [14] авторы предлагают поход к извлечению семантики из фрагмента программного кода, основанный на синтаксической структуре его содержимого. Основная идея состоит в том, чтобы представить код как коллекцию путей в абстрактном синтаксическом дереве и объединить эти пути в единый вектор, который затем использовать для предсказания семантических свойств кода.

На первом шаге строится абстрактное синтаксическое дерево для фрагмента кода (метода), после чего из него извлекаются синтаксические пути. Каждый путь представляет собой последовательность узлов, помеченных стрелками (\uparrow , \downarrow), которые обозначают направление движения (подъем или спуск) по дереву. Авторы вводят ограничение на длину и ширину путей, чтобы ограничить размер тренировочных данных и уменьшить размерность пространства.

Далее составляются тройки вида (x_s, p, x_f) , где x_s - стартовый терминал, x_f - конечный терминал, p - путь между ними. Такие тройки будем называть *контекстом*.

Например, контекст для выражения “`x = 5;`” выглядит как `<x, (NameExpr \uparrow AssignExpr \downarrow IntegerLiteralExpr), 5>`.

На рисунке 1 представлено абстрактное синтаксическое дерево, построенное для фрагмента кода из листинга 1, а также некоторые пути между его листьями.

Листинг 1: Фрагмент кода

```

1 boolean f(Object target) {
2     for (Object elem: this.elements) {
3         if (elem.equals(target)) {
4             return true;
5         }
6     }
7     return false;
8 }

```

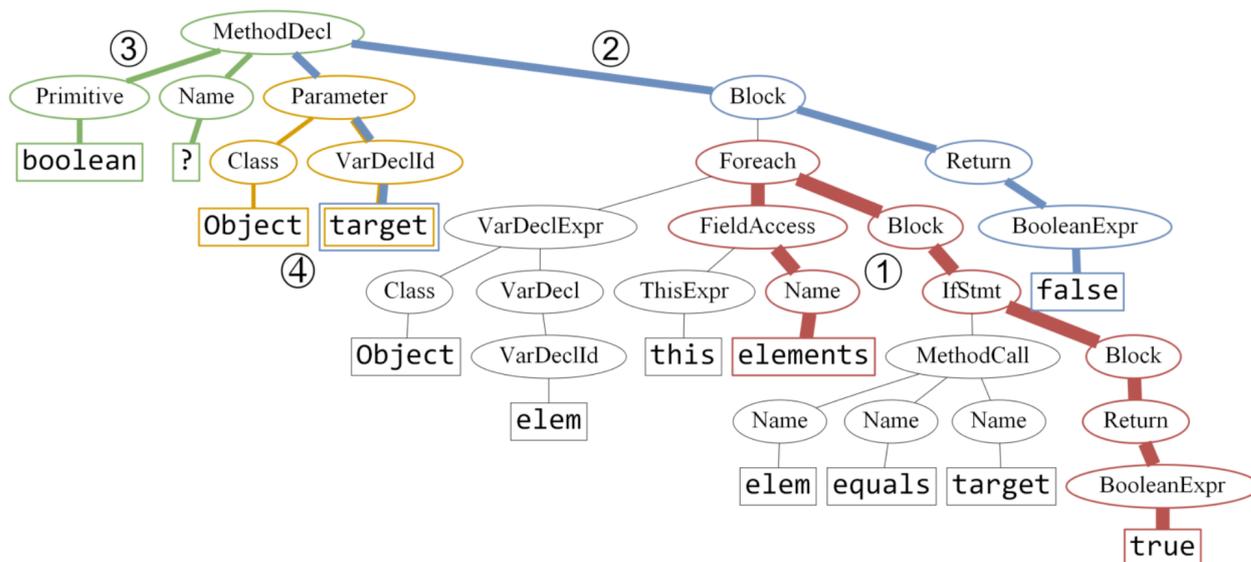


Рис. 1: AST на основе Листинга 1, заимствовано из работы [14]

Каждой компоненте контекста ставится в соответствие вектор, эти векторы конкатенируются в один вектор, к которому затем применяется функция гиперболического тангенса для нормализации, в результате получается *объединенный вектор контекста*.

Для вычисления векторного представления кода используется механизм внимания. В начале параметр внимания выбирается произвольно, а затем меняется во время обучения модели. Принимая на вход объединенные векторы контекстов, веса внимания для каждого вектора вычисляются как нормализованное скалярное произведение между объединенным вектором контекста и параметром внимания.

Так, векторное представление V для фрагмента кода вычисляется как линейная комбинация произведений весов внимания α_i и объеди-

ненных векторов контекстов c_i :

$$V = \sum_{i=1}^n \alpha_i c_i$$

Веса неотрицательны, их сумма равна единице. Полученный вектор V можно использовать для различных задач, например, для задачи предсказания имени метода. На рисунке 2 представлена архитектура описываемого подхода.

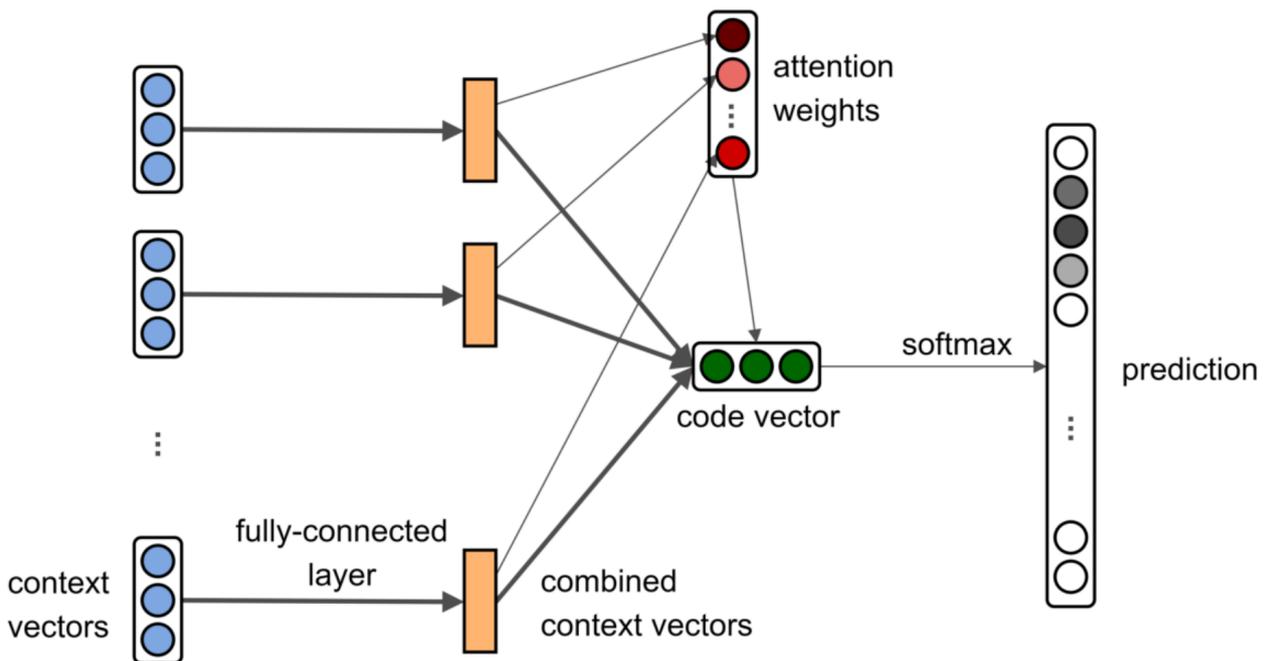


Рис. 2: Архитектура нейронной сети, заимствовано из работы [14]

К недостаткам описываемого подхода можно отнести размер модели (1.5 Gb), который не позволяет использовать ее в рамках данной работы, поскольку модель распространяется вместе с плагином. Еще одним недостатком является то, что модель не способна генерировать неологизмы в качестве предсказаний.

1.1.3. Code2seq

Алгоритм Code2seq [2] основан на той же идее, что и Code2vec: фрагмент кода представляется набором путей в AST, наиболее важные пути выбираются с помощью механизма внимания. Однако в этой работе ав-

торы используют архитектуру кодер-декодер [6], которая нашла применение в задачах машинного перевода [12], распознавания речи [4], генерации описания видео [11]. Идея подхода состоит в отображении данных из пространства одной размерности в пространство меньшей размерности. Кодер отображает входные данные в векторное пространство, а декодер реконструирует выходные данные в соответствии с размерностью второго пространства.

Кодер принимает на вход множество путей в AST и ставит каждому из них в соответствие векторное представление. Поскольку путей может быть много, авторы ограничивают их количество параметром k . В ходе экспериментов было замечено, что оптимальным значением для k является 200. Если рассматривать меньше путей, то результаты модели будут хуже, а увеличение значения не улучшает результаты. Каждый путь представляет собой пару терминалов (стартовый и конечный) в AST и последовательность узлов, соединяющих их. Векторные представления узлов хранятся в матрице. Каждый терминал представляет собой токен, который может состоять из нескольких слов. Например, "ArrayList" состоит из двух слов: "Array" и "List". Векторное представление токена w вычисляется как сумма векторных представлений входящих в него слов:

$$encode\ token = \sum_{s \in split(w)} E_s^{subtokens},$$

где E — матрица векторных представлений для слов.

Векторные представления терминалов и последовательности соединяющих их узлов конкатенируются в один вектор, представляющий собой путь, который затем подается на вход функции гиперболического тангенса для нормализации. Стартовое состояние декодера вычисляется как среднее векторных представлений путей. При генерации используются веса внимания для определения наиболее значимых путей во входной последовательности. Декодер возвращает последовательность сгенерированных токенов.

На рисунке 3 представлена архитектура описываемого подхода. Для

генерации неологизмов авторы используют алгоритм beam search, который ищет последовательность токенов с наибольшей вероятностью.

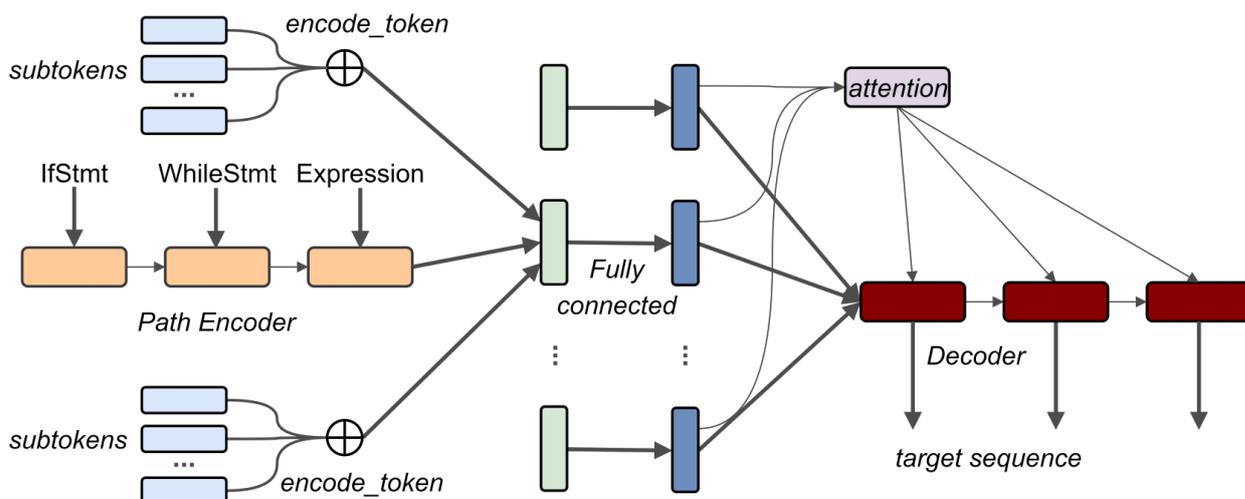


Рис. 3: Архитектура нейронной сети, заимствовано из работы [2]

Авторы предлагают использовать модель для предсказания имен методов и генерации документации к коду. Преимуществом подхода является способность генерировать неологизмы и небольшой размер модели (200 Mb), позволяющий использовать ее в рамках данной работы. Исходный код Code2seq доступен в открытом доступе¹.

1.2. Метрики для оценки качества предсказания

Для оценки качества предсказания модели используют метрики. В таблице 1 представлены четыре возможных варианта предсказания. Таким образом, есть два вида ошибок: False negative (ложный пропуск) и False positive (ложное срабатывание).

Таблица 1: Матрица ошибок классификации

	$y = 1$	$y = 0$
$\hat{y}=1$	True positive (TP)	False positive (FP)
$\hat{y}=0$	False negative (FN)	True negative (TN)

\hat{y} – предсказание модели

y – истинная метка объекта

¹<https://github.com/tech-srl/code2seq>

Точность (*precision*) показывает, сколько полученных от классификатора положительных ответов являются правильными. Чем выше точность, тем меньше ложных срабатываний. Точность вычисляется по следующей формуле:

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

Полнота (*recall*) показывает, какая часть положительных объектов была выделена классификатором. Чем выше полнота, тем меньше ложных пропусков. Полнота вычисляется по следующей формуле:

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

Для удобства хотелось бы иметь одну метрику, значение которой бы оптимизировалось во время обучения. Например, в роли такой метрики служит F1 — гармоническое среднее точности и полноты:

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

У метрики F1 есть свойство: ее значение близко к нулю тогда, когда хотя бы один из аргументов близок к нулю.

1.3. Компоненты IntelliJ Platform

IntelliJ Platform — платформа для создания интегрированных сред разработки². Например, с ее помощью созданы среды разработки IntelliJ IDEA, Rider, WebStorm. В этой секции будут описаны компоненты платформы, которые необходимы для релизации плагинов.

1.3.1. PSI

IntelliJ Platform предоставляет PSI (Program Structure Interface) — набор интерфейсов для работы с абстрактным синтаксическим деревом, построенным на основе кода, с добавлением семантики. Узлы дерева представлены классами, реализующими общий для всех элементов интерфейс `PsiElement`. На рисунке 4 представлен пример такого дерева.

В интерфейсе `PsiElement` определены методы для получения предка, потомков, соседей, диапазон текста в документе, занимаемый элементом. Одним из классов, реализующих этот интерфейс, является `PsiMethod`, позволяющий получить полезную информацию о методе: его имя, тип возвращаемого элемента, набор параметров, модификатор доступа, тело метода.

При изменении кода меняется соответствующее PSI-дерево. Платформа позволяет сохранять указатель на меняющиеся PSI-элементы дерева с помощью `SmartPsiElementPointer`. Помимо этого, платформа позволяет менять PSI-деревья: добавлять, заменять и удалять PSI-элементы.

Поиск PSI-элементов в дереве осуществляется с помощью класса `PsiTreeUtil`. Например, он позволяет получить информацию об иерархии PSI-дерева, посчитать его глубину, найти всех потомков конкретного типа. Для представления классов и интерфейсов внутри дерева служит `PsiClass`, с его помощью можно проверить, является ли класс интерфейсом, получить его внутренние классы и входящие в него методы.

²http://www.jetbrains.org/intellij/sdk/docs/intro/intellij_platform.html

```
private int getSum(int a, int b) {
    return a + b;
}
```

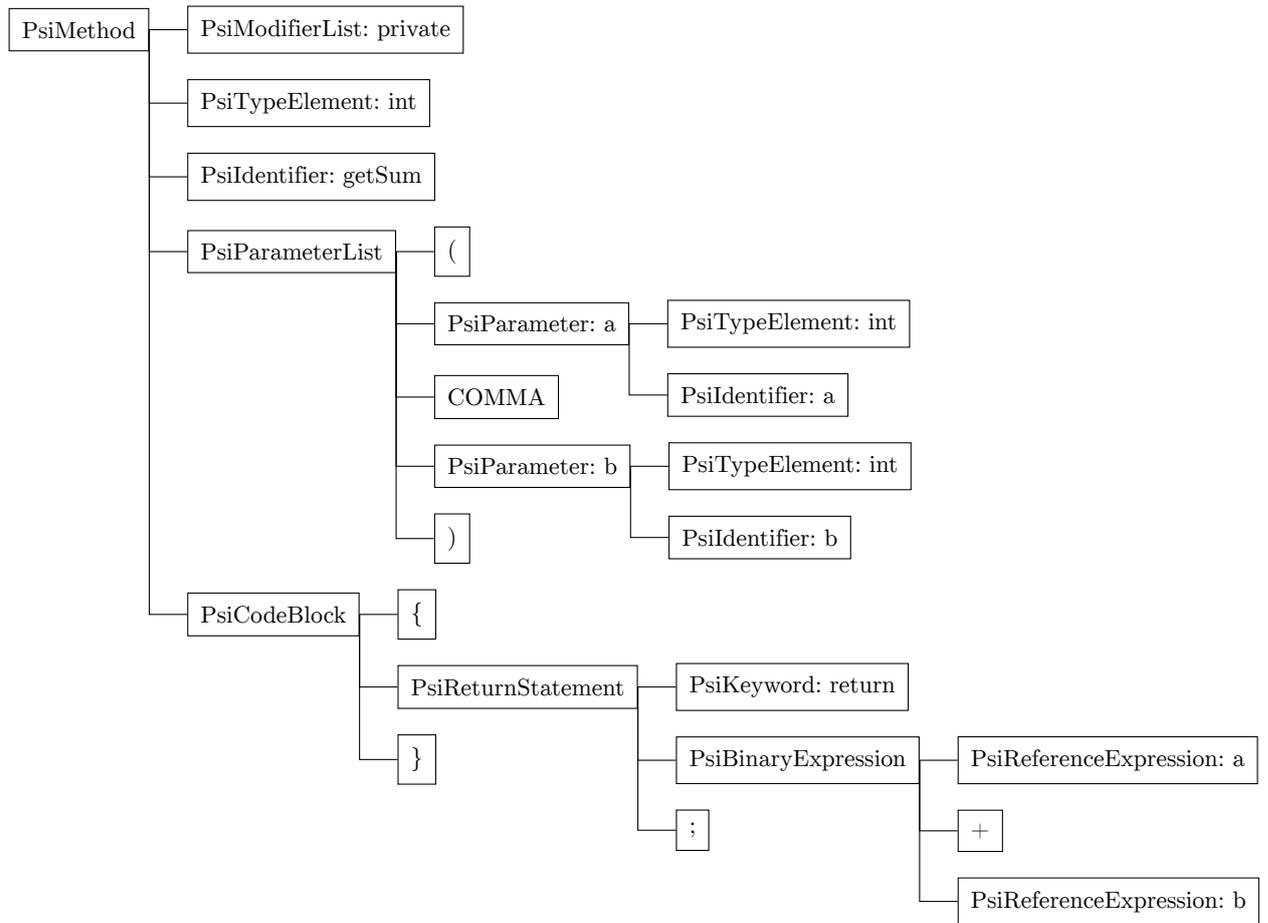


Рис. 4: PSI-дерево

1.3.2. Inspection

IntelliJ Platform предоставляет возможность анализировать код с помощью автоматических инспекций³. Встроенные в среду разработки IntelliJ IDEA инспекции находят синтаксические ошибки до компиляции и предлагают исправления. Помимо этого, инспекции помогают находить неиспользованные фрагменты кода, опечатки в словах, утечки памяти.

```
29     if (id == null) {
30         return;
31     }
32
33     if (id != null) {
34
35     } else {
36         UserInfo user = getUserById(id);
37         String userName = user.getName();
38         names.add(userName);
39     }
40
--
```

Рис. 5: Пример инспекции в IntelliJ IDEA

Инспекции сообщают об уровне серьезности обнаруженных проблем с помощью разных способов подсветки кода. На рисунке 5 представлен пример инспекции, которая сообщает разработчику о том, что в блоке **if-else** условие всегда истинно. Таким образом, фрагмент кода, выделенный на рисунке синим прямоугольником, всегда недостижим.

Для реализации инспекции необходимо отнаследоваться от класса `JavaElementVisitor` и переопределить метод для посещения интересных PSI-элементов дерева.

1.4. Выводы из обзора

Поскольку существует огромное разнообразие возможных имен идентификаторов, нам важно использовать алгоритм, способный в ка-

³<https://www.jetbrains.com/help/idea/code-inspection.html>

честве предсказаний генерировать неологизмы. Из рассмотренных алгоритмов под этот критерий подходят ConvAttention и Code2seq.

В таблице 2 представлены результаты сравнения алгоритмов по метрикам Precision, Recall и F1, которые взяты из работы [2]. Самые высокие результаты демонстрирует алгоритм Code2seq, по этой причине он был выбран в рамках данной работы для генерации имен методов.

Таблица 2: Сравнение алгоритмов по метрикам

Model	Precision	Recall	F1
ConvAttention	60.71	27.60	37.95
Code2vec	48.15	38.40	42.73
Code2seq	64.03	55.02	59.19

2. Разработанное решение

В рамках данной работы был разработан плагин на языке Kotlin⁴ для среды разработки IntelliJ IDEA с двумя режимами работы: генерация рекомендаций по запросу пользователя и автоматическая инспекция, которая в фоновом режиме анализирует код и предлагает рекомендации. Помимо этого, была добавлена автоматическая инспекция для поиска больших условных выражений и модуль сбора статистики использования плагина.

2.1. Архитектура плагина

На рисунке 6 представлена архитектура предлагаемого решения, разбитая на три компоненты: IntelliJ Platform, TensorFlow и astrid. В компоненте IntelliJ Platform представлены классы, используемые при разработке плагина. Компонента astrid содержит основные классы, реализованные в рамках данной работы.

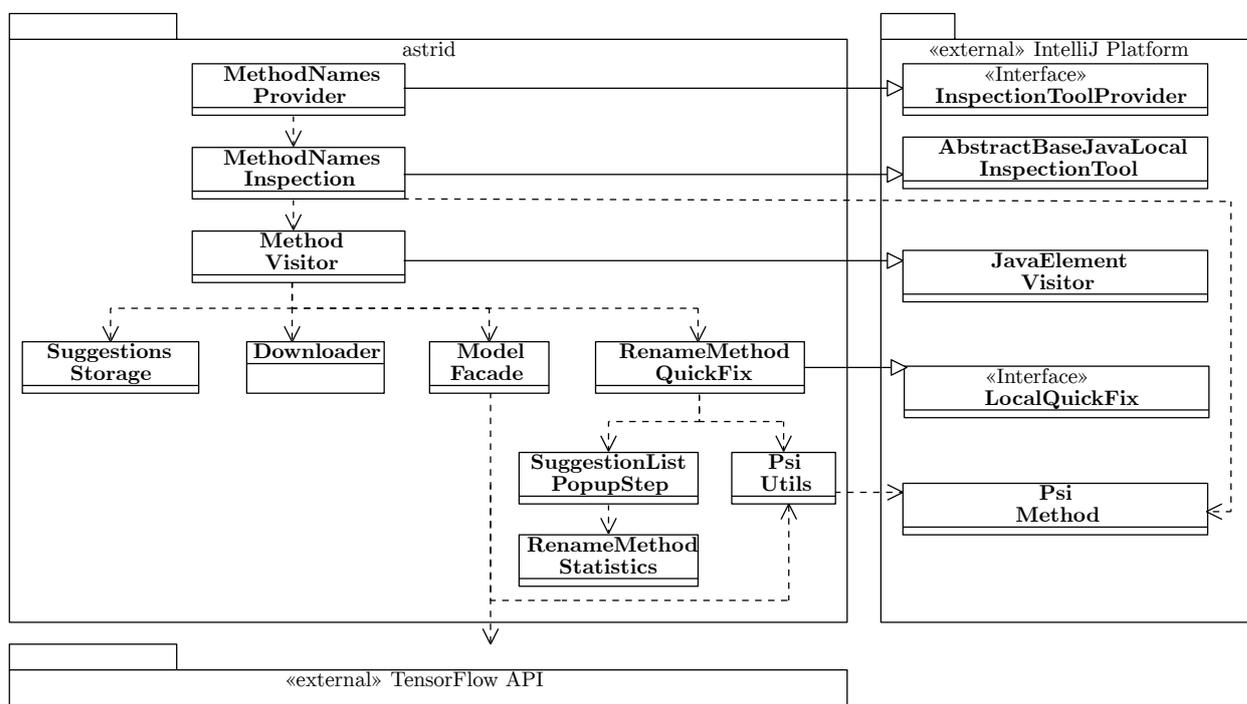


Рис. 6: Архитектура плагина

⁴<http://kotlinlang.org>

Библиотека TensorFlow предоставляет Java API⁵ для загрузки и запуска моделей, написанных на языке Python, в приложениях на Java. Саму модель было решено разместить на Dropbox, при установке плагина запускается процесс скачивания модели в файловую систему пользователя. У пользователя есть возможность наблюдать за процессом загрузки с помощью индикатора прогресса или же отложить загрузку на следующий запуск плагина. Индикатор прогресса представляет собой объект класса `ProgressIndicator`, входящий в состав IntelliJ Platform. Загрузка модели проходит в фоновом режиме, не блокируя работу пользователя.

На рисунке 7 представлена диаграмма последовательности, описывающая процесс работы плагина. После открытия проекта инспекция `MethodNameInspection` обходит все методы в текущем файле по очереди и обращается к `SuggestionStorage`: если для метода уже посчитаны рекомендации, то возвращается список посчитанных ранее рекомендаций, в противном случае запускается генерация рекомендаций. Если тело метода было изменено, то для него пересчитываются рекомендации.

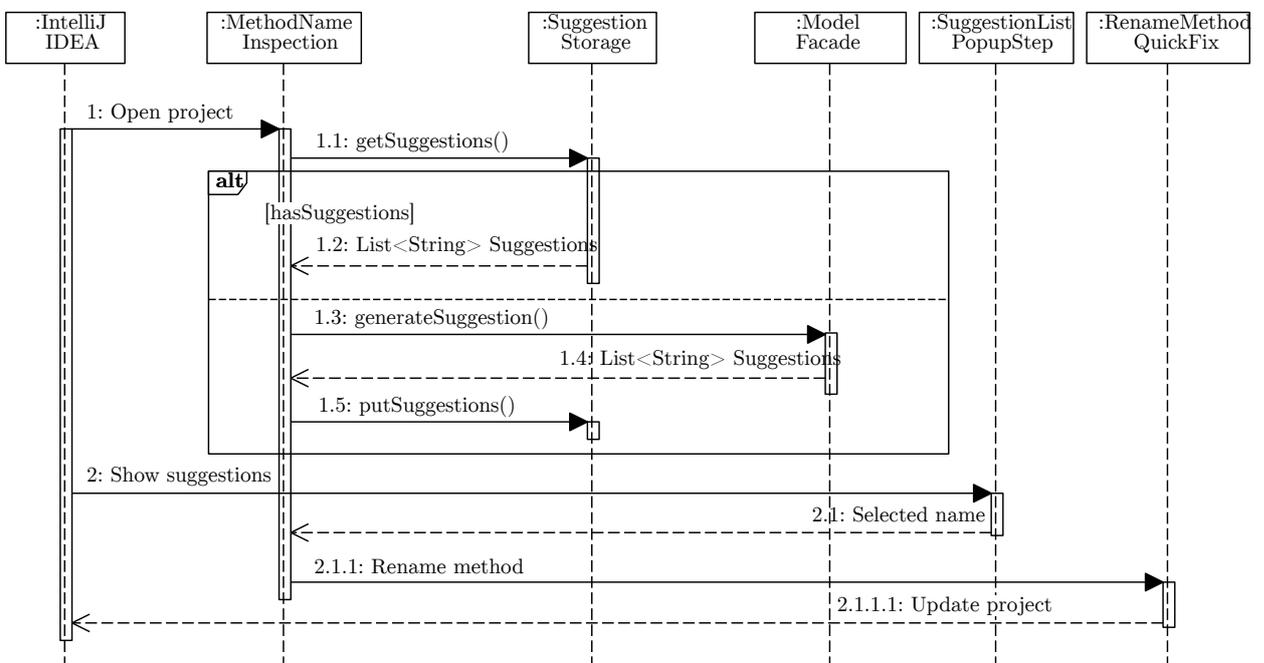


Рис. 7: Схема работы плагина

⁵https://www.tensorflow.org/install/lang_java

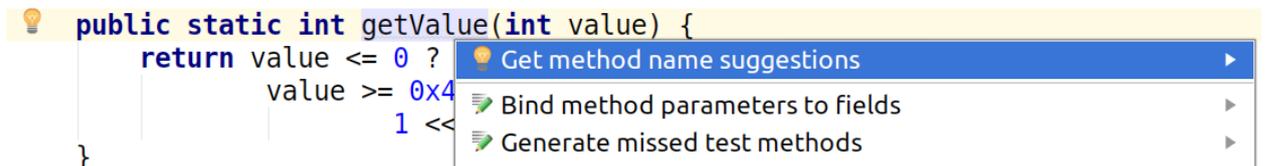
Инспекция игнорирует метод, если он:

- является конструктором;
- переопределяет метод наследуемого класса;
- реализует метод интерфейса;
- не содержит в себе кода.

`ModelFacade` представляет интерфейс для работы с моделью: принимает на вход тело метода и возвращает список посчитанных рекомендаций для метода. Если список рекомендаций не содержит в себе текущее имя метода, то сигнатура метода подсвечивается, и появляется подсказка о том, что плагин сгенерировал рекомендации для метода.

При нажатии на подсказку появляется список имен, предоставленный классом `SuggestionListPopupStep`, за переименование метода и всех его вхождений в проект отвечает класс `RenameMethodQuickFix`.

На рисунке 8 представлен внешний вид подсказки.



```
public static int getValue(int value) {  
    return value <= 0 ?  
        value >= 0x4  
        1 <<  
}
```

Рис. 8: Пример подсказки

На рисунке 9 представлен пример сгенерированных рекомендаций для метода быстрого возведения двойки в степень. Если выбрать последний вариант в списке рекомендаций, то инспекция будет игнорировать метод.

```
public static int getValue(int value) {  
    return value <= 0 ?  
           value >= 0x4  
           1 << LeadingZeros(i: value - 1));  
}
```

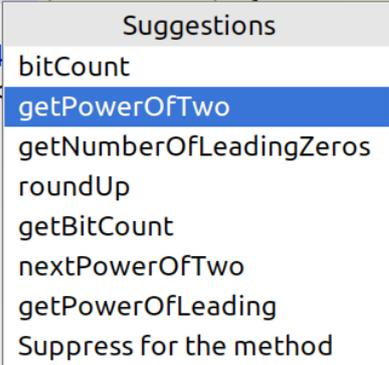


Рис. 9: Пример рекомендаций

2.2. Алгоритм работы инспекции

На рисунке 10 представлен алгоритм работы автоматической инспекции. Для каждого метода строится абстрактное синтаксическое дерево, из него извлекаются синтаксические пути между всеми терминалами, получившееся множество путей подается на вход модели Code2seq. Модель возвращает список предсказаний, из которого удаляются короткие имена и специальные символы, которые не несут смысловой нагрузки. Если список рекомендаций не содержит текущее имя метода, то регистрируется подсказка, и текущее имя метода подсвечивается в среде разработки.

У пользователя есть возможность ознакомиться со списком рекомендаций и либо выбрать одну из них, либо отключить подсказку для данного метода. После выбора имени из списка рекомендаций осуществляется переименование метода и всех его вхождений в проект.

Помимо инспекции для рекомендации имен методов, была реализована инспекция для поиска больших условных выражений. Считаем условное выражение большим, если его длина составляет более ста

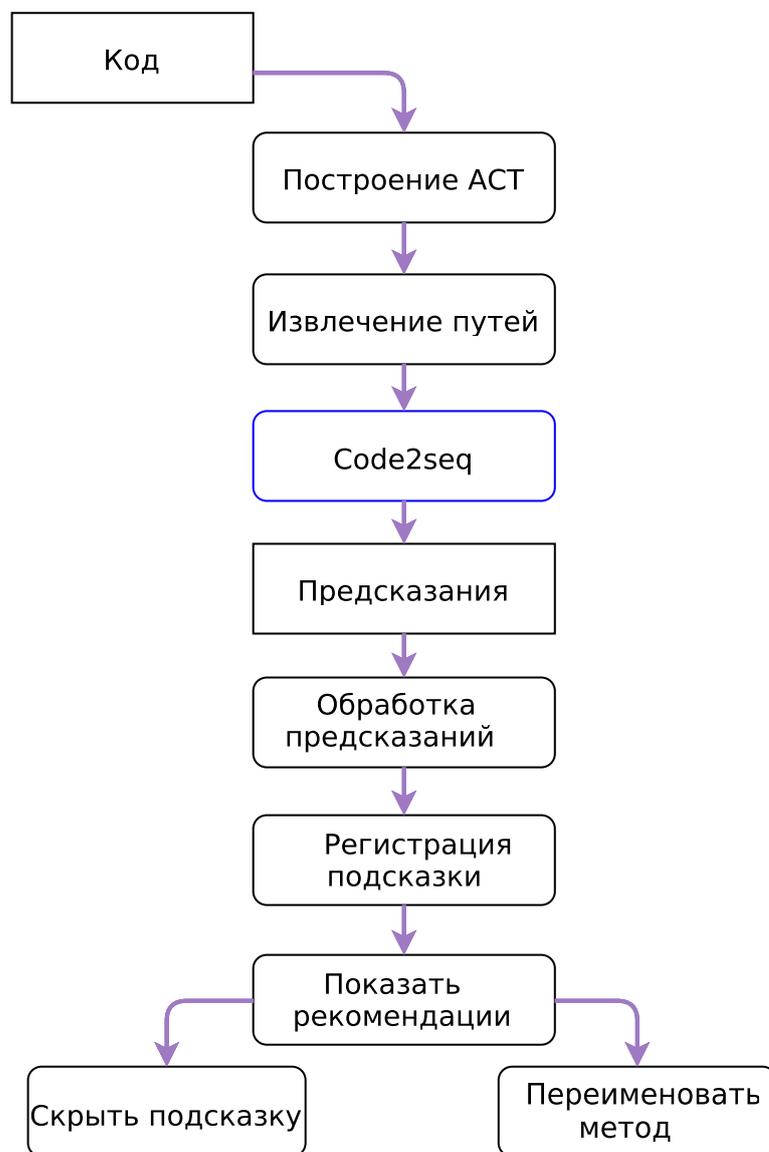


Рис. 10: Алгоритм работы инспекции

символов, этот критерий был подобран эмпирически. Так, если условное выражение большое, то для повышения читаемости кода его стоит выделить в отдельный метод с подходящим названием, а на прежнее место добавить вызов нового метода.

Инспекция подсвечивает все найденные большие условные выражения. После того, как пользователь нажмет на подсвеченное выражение, алгоритм работы выглядит следующим образом:

1. извлечь условное выражение;
2. сгенерировать для него список рекомендаций;
3. из рекомендаций убрать имя, если в текущем классе есть метод с таким же;
4. в качестве имени для нового метода выбрать имя с наибольшей вероятностью;
5. создать в текущем классе приватный метод, возвращающий булево значение;
6. условное выражение поместить в тело нового метода;
7. на прежнее место условного выражения добавить вызов нового метода.

Реализованные в рамках данной работы инспекции настраиваемые: их можно отключить или подключить по желанию. При этом возможность сгенерировать рекомендации для имени метода по запросу есть всегда, она доступна по нажатию `Alt+Enter` на имени интересующего метода.

2.3. Модуль сбора статистики

В данной работе реализован модуль сбора статистики использования плагина. Статистика хранится для всех проектов в целом без разделения по проектам.

Возможность сохранять данные между запусками среды разработки предоставляет класс `PersistentStateComponent`, входящий в состав IntelliJ Platform. Он позволяет сохранять примитивные типы, коллекции, строки, ассоциативные массивы в формате XML. Статистика хранится локально в файловой системе пользователя.

Для каждого применения или отклонения рекомендации сохраняется длина соответствующего метода (количество строк) и логарифм вероятности предсказания модели. Также сохраняются счетчики применений и отклонений рекомендаций.

Поскольку вероятность имени метода вычисляется как произведение условных вероятностей слов, входящих в состав имени, получаем очень маленькие числа, поэтому удобнее работать с их логарифмами. За сериализацию данных в формат XML отвечает класс `RenameMethodStatistics`.

3. Апробация

В ходе апробации плагин был протестирован шестью разработчиками на их собственных проектах, написанных на языке Java. После тестирования были собраны оценки полезности плагина по пятибалльной шкале. Результаты опроса представлены на рисунке 11. Видно, что в целом плагин оценили положительно и отметили полезность рекомендаций.

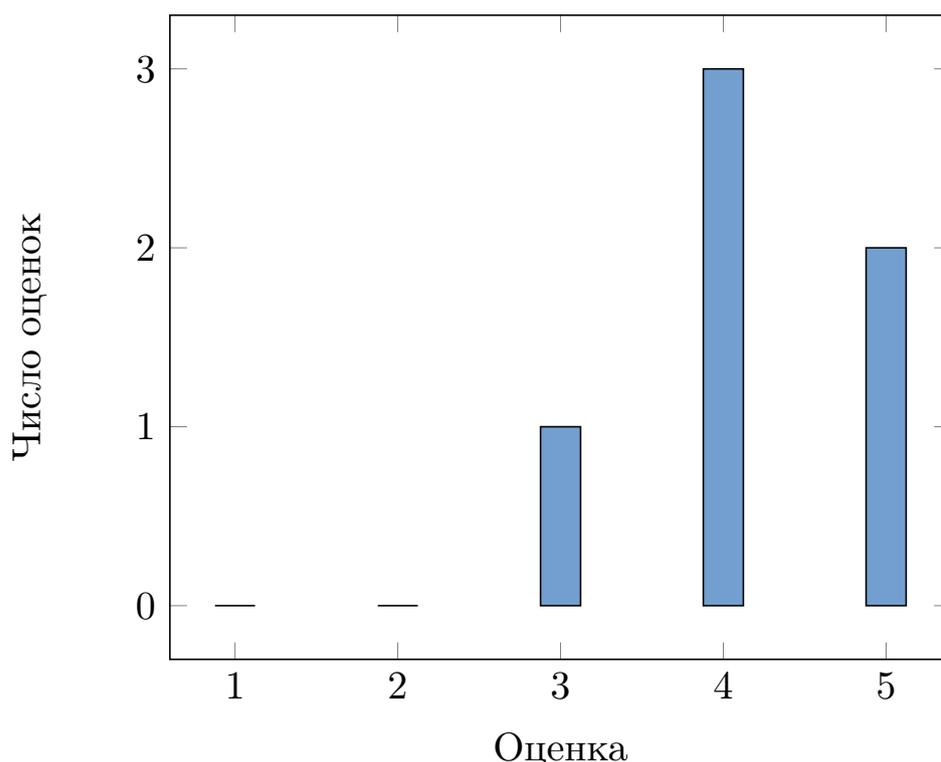


Рис. 11: Оценка полезности плагина

После тестирования была собрана статистика использования плагина. На рисунке 12 представлен график, построенный на основе собранной статистики. Синие точки соответствуют одобренным пользователем рекомендациям, красные — отклоненным. По оси x указана длина метода, по y — логарифм вероятности рекомендации, предсказанной моделью.

В ходе апробации было выявлено, что разработанный в рамках данной работы плагин приносит пользу разработчикам в их работе. Анализ статистики использования плагина установил зависимость между

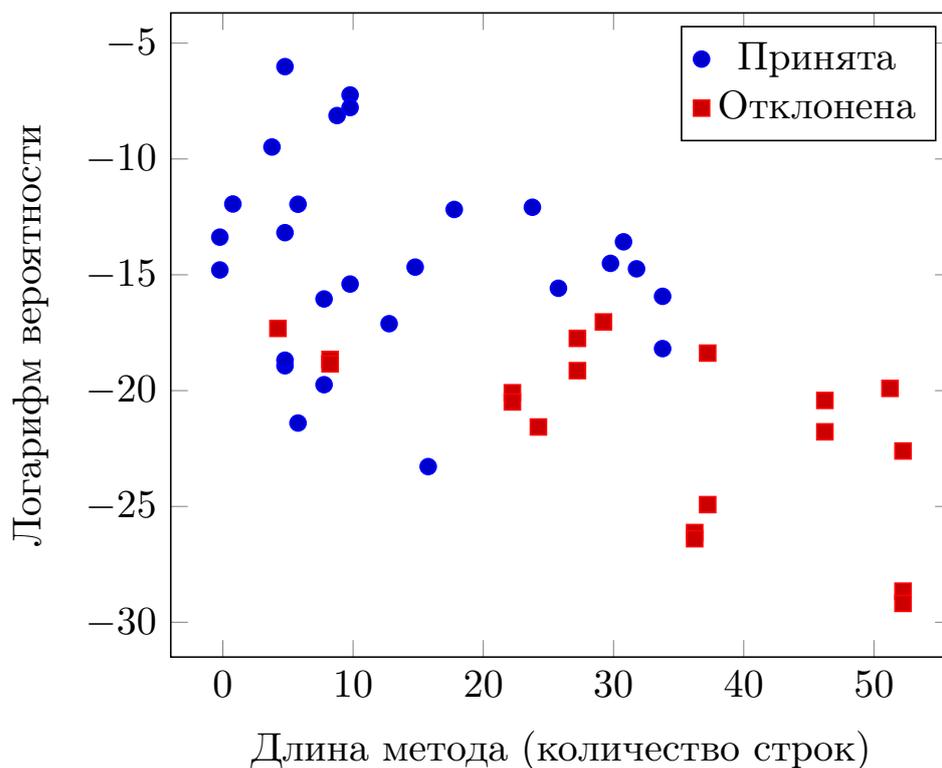


Рис. 12: Зависимость релевантности рекомендаций от длины метода

релевантностью предложенных плагином рекомендаций и длиной метода, для которого рекомендации были посчитаны. Чем больше тело метода, тем сложнее выделить наиболее важную с точки зрения семантики информацию. Тем не менее, на методах длиной менее тридцати строк рекомендации плагина пользователи сочли релевантными.

Заключение

В ходе работы были достигнуты следующие результаты.

- Проведен обзор предметной области. Рассмотрены алгоритмы рекомендации имен методов, описаны метрики для оценки их качества и проведено сравнение описанных алгоритмов.
- Выбран алгоритм рекомендации имен методов, демонстрирующий высокое качество предсказаний.
- Разработана архитектура и выполнена реализация плагина к IntelliJ IDEA. Плагин в фоновом режиме анализирует код и предлагает рекомендации имен для методов.
- Проведена апробация плагина. Плагин протестирован пользователями, которые в основном дали положительную оценку. Проведен анализ собранной статистики использования и было замечено, что на методах короче тридцати символов плагин генерирует наиболее релевантные рекомендации.

Таким образом, в результате данной работы был создан плагин для среды разработки IntelliJ IDEA, который рекомендует имена для методов. Код проекта доступен по ссылке⁶.

⁶<https://github.com/ml-in-programming/astrid>

Список литературы

- [1] Allamanis Miltiadis, Peng Hao, Sutton Charles. A convolutional attention network for extreme summarization of source code // International Conference on Machine Learning. — 2016. — P. 2091–2100.
- [2] Alon Uri, Levy Omer, Yahav Eran. code2seq: Generating sequences from structured representations of code // arXiv preprint arXiv:1808.01400. — 2018.
- [3] Bahdanau Dzmitry, Cho Kyunghyun, Bengio Yoshua. Neural machine translation by jointly learning to align and translate // arXiv preprint arXiv:1409.0473. — 2014.
- [4] A Comparison of Sequence-to-Sequence Models for Speech Recognition. / Rohit Prabhavalkar, Kanishka Rao, Tara N Sainath et al. // Interspeech. — 2017. — P. 939–943.
- [5] Fowler Martin. Refactoring: improving the design of existing code. — Addison-Wesley Professional, 2018.
- [6] Learning phrase representations using RNN encoder-decoder for statistical machine translation / Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre et al. // arXiv preprint arXiv:1406.1078. — 2014.
- [7] Pradel Michael, Sen Koushik. DeepBugs: A Learning Approach to Name-based Bug Detection // arXiv preprint arXiv:1805.11683. — 2018.
- [8] Ragkhitwetsagul Chaiyong, Krinke Jens, Clark David. A comparison of code similarity analysers // Empirical Software Engineering. — 2018. — Vol. 23, no. 4. — P. 2464–2519.
- [9] Relating identifier naming flaws and code quality: An empirical study / Simon Butler, Michel Wermelinger, Yijun Yu, Helen Sharp // Reverse

Engineering, 2009. WCRE'09. 16th Working Conference on / IEEE. — 2009. — P. 31–35.

- [10] Russell Stuart, Norvig Peter, Intelligence Artificial. A modern approach // Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs. — 1995. — Vol. 25, no. 27. — P. 79–80.
- [11] Sequence to sequence-video to text / Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue et al. // Proceedings of the IEEE international conference on computer vision. — 2015. — P. 4534–4542.
- [12] Sutskever Ilya, Vinyals Oriol, Le Quoc V. Sequence to sequence learning with neural networks // Advances in neural information processing systems. — 2014. — P. 3104–3112.
- [13] Vasilescu Bogdan, Casalnuovo Casey, Devanbu Premkumar. Recovering clear, natural identifiers from obfuscated JS names // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering / ACM. — 2017. — P. 683–693.
- [14] code2vec: Learning Distributed Representations of Code / Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav // arXiv preprint arXiv:1803.09473. — 2018.
- [15] A survey of machine learning for big code and naturalness / Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, Charles Sutton // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 4. — P. 81.