**Санкт–Петербургский государственный университет**

***Пословская Элеонора Дмитриевна***

## Выпускная квалификационная работа

# *Использование сетей глубокого обучения в задаче локализации ошибок в исходном коде*

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5005.2017 «Прикладная математика, фундаментальная информатика и программирование»
Профиль «Математическое и программное обеспечение вычислительных машин»

Научный руководитель:
кандидат физ.-мат. наук, кафедра математической теории игр и статистических решений, доцент Панкратова Ярославна Борисовна

Соруководитель:
Руководитель практики в IntelliJ Labs, JetBrains,
Хворов Александр Александрович

Рецензент:
кандидат эконом. наук, кафедра математического моделирования и прикладной информатики, доцент Казьмина Олеся Александровна

Санкт-Петербург
2021 г.

# Содержание

**Введение** . . . . . 3

**Постановка задачи** . . . . . 5

**Термины** . . . . . 6

**Обзор литературы** . . . . . 8

**Глава 1. Данные** . . . . . 13

1.1. Разметка данных . . . . . 14

1.2. Оценочные данные . . . . . 17

1.3. Обработка данных . . . . . 17

**Глава 2. Реккурентные нейронные сети** . . . . . 20

2.1. RNN . . . . . 20

2.2. LSTM . . . . . 21

**Глава 3. Эксперименты над данными** . . . . . 24

3.1. Базовый алгоритм . . . . . 24

3.2. Модель-конкурент . . . . . 24

3.3. Предлагаемая модель . . . . . 24

3.4. Ранжирование . . . . . 26

3.5. Результаты . . . . . 29

**Выводы** . . . . . 32

**Заключение** . . . . . 33

**Список литературы** . . . . . 34

# Введение

Область искусственного интеллекта, стремительно развивающаяся в последние несколько лет, успешно применяется везде, где возникает потребность в обработке информации. Но если обработка естественного языка сейчас является хорошо изученным разделом машинного обучения, то пусть и похожие на них языки программирования имеют другую структуру, более сложные связи между оперируемыми сущностям и совершенно иной спектр прикладных задач, для которых привычные для NLP методы необходимо грамотно адаптировать. Поэтому программирование само по себе является не только инструментом для создания нейросетевых моделей, но и областью, где такие модели могут быть востребованы. Так, в процессе создания качественного программного обеспечения время, затраченное на написание кода и поиск и исправление ошибок, примерно сопоставимо между собой. Желание упростить одну из этих частей сводится к исследованию двух обширных задач: задачи автоматической генерации программного кода и более простой, но не менее важной – задачи поиска ошибок в полученном коде.

Задача автоматической генерации корректного кода кажется более привлекательной с точки зрения экономии трудозатрат, однако она еще не была качественно решена по ряду причин, самой трудной из которых является сложность формализации требований от человека к результирующей программе. Но существует вспомогательная к ней задача – задача поиска ошибок в исходном коде программы, которая могла бы помочь автоматически исправлять ошибки в программах, тем самым пусть и не автоматизируя, но существенно упрощая процесс написания и валидации кода. Научить модель искать ошибки по какому-то определенному алгоритму вполне реально при помощи статических анализаторов кода, однако они не учитывают более сложные и глубоко заложенные ошибки, часто возникающие непосредственно при исполнении программы. Человек может выявить такие ошибки при ручном методе отладки программы, но это довольно сложно задать в виде каких-либо инструкций. Поскольку в процессе компиляции и работы программ при возникновении ошибок происходит вы-

вод стека вызовов, где происходит вывод сообщений об ошибках и путей до методов, где эта ошибка вскрылась, можно использовать эту информацию для ограничения области поиска ошибок в программе. Таким образом, совместно используя информацию как об исходном коде программы, так и зная последовательность вызванных функций до обнаруженной ошибки, можно построить модель, которая будет указывать на метод, содержащий в себе некорректный код.

## Постановка задачи

Основной задачей этой работы является создание модели, позволяющей упростить и ускорить нахождение ошибок в исходном коде программ на основе информации об исходном коде и стеках вызовов. Поскольку данные о стеках вызовов были предоставлены компанией JetBrains и взяты из проекта IntelliJ IDEA, разработка которой ведется преимущественно на языке программирования Java, модель будет ориентирована на обработку данных именно на этом языке.

Решить поставленную задачу предполагается через выполнение следующих этапов:

- ознакомиться с литературой и узнать подробнее о существующих подходах в области локализации ошибок
- собрать данные и привести их в вид, пригодный для подачи на вход модели, разделить на обучающее и оценочное множества
- оценить работу лучшей и наиболее похожей по используемым данным модели-конкурента
- разработать архитектуру модели и реализовать ее
- обучить модель на тренировочной части существующих данных и оценить на отложенной выборке
- оценить модель на общеизвестных данных, чтобы сравнить качество с существующими алгоритмами

## Термины

Поскольку область, в которой предлагается решать поставленную задачу имеет специфичный набор обозначений для некоторых сущностей, в данном разделе будет небольшой экскурс в суть данных, с которыми предстоит вести работу.

Задача классификации методов на содержащие и не содержащие ошибку сводится к обычной постановке задачи бинарной классификации.

**Определение**. Пусть у нас есть множество объектов $X$ и множество классов $Y$, которое конечно. Имеется выборка $X^l = (x_i, y_i)_{i=1}^l$, $y_i = y^*(x_i)$. Необходимо построить такой классифицирующий алгоритм $a : X \to Y$, аппроксимирующий зависимость $y^*(x)$ на всем множестве $X$.

В данной работе, множество $X$ состоит из методов стеков вызовов, а множество $Y$ – из двух классов 0 и 1, обозначающих соответственно наличие и отсутствие ошибки.

Объекты пространства $X$ могут быть описаны различными способами, чаще всего представляющими собой многомерный вектор признаков. В некоторых случаях, когда объект представляет собой категориальную переменную, возникают трудности с его кодированием в исходном пространстве - например, для текстов в таком случае применяют такие методы как Bag Of Words или TF-IDF, что в результате дает вектор размерностями в десятки тысяч элементов. Большинство алгоритмов страдают от проклятья размерности[1], что влечет за собой определенные неудобства в работе с ними. Поэтому часто используются методы понижения размерности векторов в пространства более низкой размерности, а сами вектора в таком случае называют **эмбеддингами**.

**Определение**. Эмбеддинг - это векторное представление объекта в пространстве более низкой размерности чем исходное, сохраняющее свойства объекта.

Для облегчения понимания состава данных, с которыми далее будет вестись работа, небольшой экскурс в само понятие стектрейса. Как известно, на низком уровне исполнения программ, необходимо сохранять информацию о последовательности вызовов и возвратов в функции для

соблюдений правильной структуры исполнения программы. Для решения этой задачи используется простая структура Last In First Out, называемая иначе стеком, в который при каждом новом вызове функции поступает адрес возврата данных и при окончании работы функции по этому адресу данные возвращаются, и функция уходит со стека. При успешном исполнении программы стек должен оказаться пуст, но при ошибочном - можно извлечь информацию о методах оставшихся в стеке. Для каждого падения программы компилятор, а именно JVM - Java Virtual Machine, выводит информацию о методах, встречаемых в стеке вызовов до метода, вызвавшего падение программы и сгенерированное исключение с текстом ошибки (либо базовое, либо настроенное программистом вручную для такой ситуации), что уже в свою очередь называется **стектрейсом**.

```
java.lang.Exception: No runnable methods
    at org.junit.runners.BlockJUnit4ClassRunner.validateInstanceMethods(BlockJUnit4ClassRunner.java:169)
    at org.junit.runners.BlockJUnit4ClassRunner.collectInitializationErrors(BlockJUnit4ClassRunner.java:104)
    at org.junit.runners.ParentRunner.validate(ParentRunner.java:355)
    at org.junit.runners.ParentRunner.<init>(ParentRunner.java:76)
    at org.junit.runners.BlockJUnit4ClassRunner.<init>(BlockJUnit4ClassRunner.java:57)
    at org.junit.internal.builders.JUnit4Builder.runnerForClass(JUnit4Builder.java:10)
    at org.junit.runners.model.RunnerBuilder.safeRunnerForClass(RunnerBuilder.java:59)
    at org.junit.internal.builders.AllDefaultPossibilitiesBuilder.runnerForClass(AllDefaultPossibilitiesBuilder.java:26)
    at org.junit.runners.model.RunnerBuilder.safeRunnerForClass(RunnerBuilder.java:59)
    at org.junit.internal.requests.ClassRequest.getRunner(ClassRequest.java:26)
```

**Рис. 1:** Пример вывода стектрейса в консоль. Вызываемые методы перечислены в порядке, обратном их вызову в программе.

Поскольку часто решение обнаруженных программистами или пользователями ошибок не входит в их компетенции, они должны передать как можно более полную информацию о них программистам, отвечающим за поддержку программного обеспечения. Такие сообщения об ошибках называются **баг-репортами**(в английской литературе также issue) и они содержат в себе информацию о неполадке, но могут также включать в себя пути воспроизведения бага или вывод программы при падении.

## Обзор литературы

Область исследования исходных текстов программ довольно обширна и нашла свое отражение во множестве работ, в том числе посвященных задаче локализации ошибок в исходном коде. В основном были предприняты подходы к работе с текстом программ теми же методами, что и с текстами на естественных языках, но более поздние статьи начинают помимо этого учитывать также структурные особенности кода: абстрактные синтаксические деревья (AST) и потоки данных между отдельными структурами.

Один из первых вариантов работы с AST программ, встречаемых в статья по локализации некорректного кода, включал в себя обход построенных деревьев[2], из которого в свою очередь получается корпус путей для встречаемых в них сущностей языка, таких как упоминания методов и переменных. Основная работа здесь ведется в рамках решения задачи локализации и исправления ошибок из небольшого заданного класса, а датасет собран из реальной кодовой базы с искусственно внесенными в нее ошибками. На этом корпусе далее обучается двунаправленная Long Short-Term Memory модель с выходом в полносвязные слои, где происходит оценка кандидатов на наличие бага и ведется поиск наиболее близкого кандидата на исправление. Такие жестко заданные классы ошибок не подходят к цели нашей работы - поскольку мы хотим находить любые виды ошибок в исходном коде, но ограниваем поиск только методами из стека вызовов. Однако идея работы с LSTM для обработки кода также может быть перенесена на использование этой архитектуры для обработки последовательности методов в стектрейсе.

Уже через год вышла статья[3] результатом которой стала библиотека code2vec, позволяющая без переобучения модели на своих данных получать векторное представление кода программы. Для обучения модели был собран корпус из 14M различных методов на языке Java, а в качестве прикладной задачи для оценки качества получаемых эмбеддингов кода была выбрана задача предсказания имени метода по его содержимому. Как и в предыдущей статье, здесь используется представление метода через контексты-пути в AST, в которых он встречается и для каждого кон-

текста в результате образуется триплет (метод$_{in}$, путь между методами, метод$_{out}$). Контексты при помощи словарей для методов и путей через методы домножаются на обучаемую матрицу, в результате чего получается эмбеддинг для контекста. Пропуская контексты через полносвязную сеть, авторы добиваются комбинирования компонент контекста и далее к ним применяется механизм внимания[4] - также обучаемый слой, коэффициенты которого помогают сети правильно усреднить все контексты в один вектор. Итоговый вектор является репрезентативным для всего кода метода. Исходный код библиотеки распространяется свободно на GitHub[1] и поэтому полученную модель легко можно использовать для других задач, требующих эмбеддинги кода. Минусом однако является большой вес модели и низкая скорость работы, что однако было исправлено в следующей версии очень близкой к этой модели – code2seq от тех же авторов. Также очень неплохим плюсом в сторону модели является расширения экстрактора от JetBrains Research, позволяющие работать не только с Java, но и с другими популярными языками программирования.

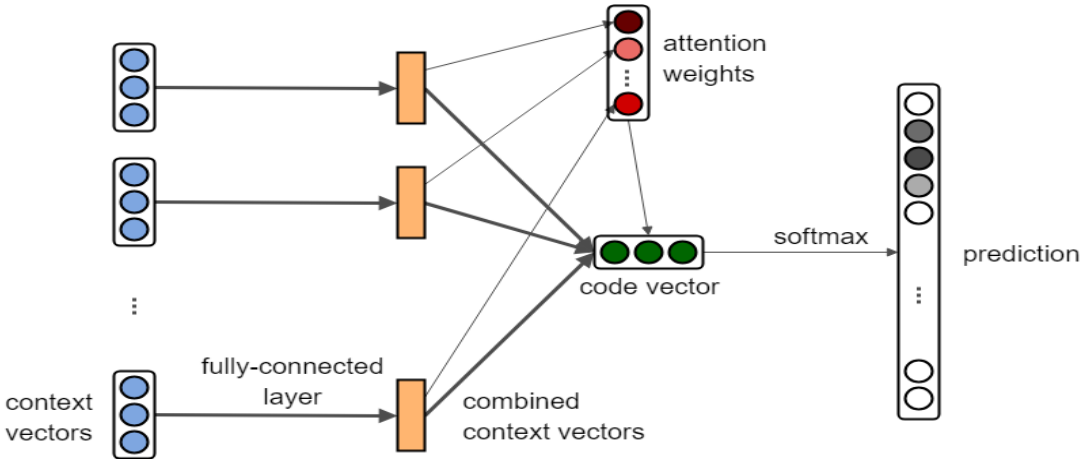


**Рис. 2:** Архитектура code2vec

[1] `https://github.com/tech-srl/code2vec`

Улучшением предыдущей архитектуры в code2seq[2][5] стало использование BiLSTM в качестве кодирующей модели для последовательности методов. Тем самым создатели добились лучшего улавливания моделью контекста методов и это позволило им от задачи суммаризации кода перейти к гораздо более сложной задаче генерации кода. Одновременно с этим модель стала экономней в плане потребления ресурсов и более легко обучаемой и сохранила возможность использования экстракторов для других языков. Эти факторы стали для нас ключевыми при выборе между code2vec и code2seq, поскольку очень важна возможность переучить модель на другой язык и запустить ее на обычном компьютере.

Поскольку методов локализации ошибок довольно много и они не слишком отличаются друг от друга в особенностях архитектуры, мы остановимся на тех статьях, авторы которых во-первых предоставляют открытый исходный код своих работ и во-вторых оцениваются на одном датасете, что существенно упрощает будущее сравнение полученных результатов с моделями-конкурентами.

В задачах локализации ошибок в свою очередь даже сейчас не всегда используются методы с применением глубокого обучения. Часто они основаны на Spectrum Based Bug Localization и используют статистические методы анализа исходного кода и встречаемости в нем ошибок. Предложенный в статье[6] метод предлагает использовать SBBL совместно с построением языковой модели на основе подсчета вероятности встретить похожий код в заданном месте по некоторым формулам, самыми известными из которых являются Tarantula, Jaccard, Ochiai и формулы для их рассчета приведены в той же статье. Также на полученных оценках они обучают решающий лес для ранжирования результатов, что является хорошей практикой и часто встречается в подобных работах.

Более практически понятный метод оценки исходного кода на наличие ошибок это Historical Spectrum Based Bug Localization[3][**?**]. Помимо оценки исходного кода, анализируется также история взаимодействия разработчика с кодом рассматриваемой программы. Поскольку современная

[2] `https://github.com/tech-srl/code2seq`

[3] `https://github.com/justinwm/HSFL`

разработка практически полностью завязана на системах контроля версий, идея авторов использовать историю посылок и изменений исходного кода вполне разумна. Для каждого метода помимо оценок SBBL как ранее также рассчитывается скор изменения кода, который вносит корректировку в итоговый результат и улучшает работу обычного SBBL подхода на несколько процентов. Из этой работы наиболее полезной кажется идея использования истории взаимодействия авторов кода с ним, поскольку частоизменяемый код действительно может более вероятно содержать в себе ошибку.

Интересен также подход, при котором строится последовательность вызова методов во время тестирования программы и из них собирается большой граф вызова методов[8]. Авторы предлагают работать с ним по той же схеме, что используется при обходе графов ссылок в мировой сети, а именно использовать известный алгоритм PageRank. Поскольку при помощи обычного SBBL метода можно оценить каждый метод на вероятность нахождения в нем ошибки, то полученные вероятности инициализируют оценки PageRank перед началом работы алгоритма и уточняются им в процессе работы. Данная модель существенно улучшает результаты работы SBBL метода, но довольна тяжела для использования в больших проектах с огромными графами связей.

Поскольку использование стектрейсов потенциально является неплохим дополнением к данным об исходном коде, то есть и подход[4], частично использующий эту информацию[9]. Немного меняется использование метода SBBL – теперь оценка вероятности некорректности метода базируется на том, насколько он похож на сообщение об ошибке, указанное человеком, в этом описании также может фигурировать содержимое стектрейса. Похожесть двух текстов вычисляется путем рассчета и перемножения их TF-IDF векторов, далее оценка похожести векторов уточняется несколькими способами. Если в баг-репорте упоминается стектрейс, то соответственно входящие в него файлы прибавляют к оценке обратное к номеру позиции число, также подсчитывается и нормализуется длина метода, являющаяся значимым фактором для вероятности допустить ошибку. На собранных

[4] `https://github.com/LeeeWee/BugLocator`

данных обучается метод опорных векторов и результаты ранжируются по его оценкам. Этот метод не слишком новый, но он прост в реализации, использует похожие данные и авторы делятся хорошими метриками качества работы алгоритма. В связи с этим, было решено обучить на предоставленных данных эту модель в качестве проверки как качества созданной нами модели, так и оценить насколько мы доверяем датасету на котором модель оценивалась авторами.

# Глава 1. Данные

Данные для работы над проектом были предоставлены компанией JetBrains, которые были собраны в процессе корпоративной разработки ПО IntelliJ IDEA, в связи с чем не представляется возможным выложить их в открытый доступ. Разработка такого крупного проекта ведется при помощи системы контроля версий Git и соответственно выкладывается в приватное хранилище - репозиторий на сайте GitHub, откуда в дальнейшем будет браться дополнительная информация. Предложенные данные включают в себя набор из множества файлов и связующей их с кодом в репозитории таблицей, а именно:

1. Таблица, содержащая отображение идентификатора проблемы с кодом issue_id в идентификатор report_id, в свою очередь являющимся номером стека вызовов, полученного в результате вызова некорректного кода

2. Коллекцию файлов формата .json определенной структуры, содержащих информацию о последовательности методов, вызванных до того как в программе произошла ошибка

При разработке и тестировании IntelliJ IDEA разработчики часто встречаются с падениями программы, откуда и была накоплена информация о стектрейсах. Она записывается в удобном формате JSON, позволяющем легко десериализовать записанные данные в словарь объектов.

Рассмотрим подробнее структуру распарсенного и сохраненного стектрейса.

1. Поле id совпадает с названием файла и является также идентификатором report_id, упомянутым ранее

2. Дата создания стектрейса timestamp в unixtime формате

3. Следующее поле class отвечает за хранение выброшенных программой исключений - они не обязательно единственны

4. В поле commit может указываться commit кода, выкинувшего данное исключение, но часто это поле остается пустым

5. Поле frames задано массивом, содержащим информацию о каждом методе, входящем в стектрейс

   (a) Полное название метода method_name в стандартном для Java формате через точку, включает в себя имя пакета, класс и имя метода (в случае лямбда-функций - специальный символ $)

   (b) Имя файла file_name с исходным кодом

   (c) Номер позиции метода в файле

<table>
<tr><td>id</td><td colspan="2">338337</td></tr>
<tr><td>timestamp</td><td colspan="2">1316517478452</td></tr>
<tr><td>class</td><td colspan="2">[java.lang.IndexOutOfBoundsException]</td></tr>
<tr><td rowspan="4">frames</td><td>method_name</td><td>com.intellij.editor.LineSet.findLineIndex</td></tr>
<tr><td>file_name</td><td>LineSet.java</td></tr>
<tr><td>line_number</td><td>571</td></tr>
<tr><td colspan="2">...</td></tr>
</table>

**Таблица 1:** Пример части стектрейса

В ходе работы мы хотим решать задачу бинарной классификации методов в стеке вызовов, а именно наличие (1) или отсутствие (0) ошибки в методе, предсказав это для каждого встречаемого в стеке метода. Однако на текущем этапе не имеется абсолютно никакой разметки методов по классам, а передача данных в подобном сыром виде на вход модели не представляется возможной.

## 1.1 Разметка данных

Для разметки данных необходимо было задействовать дополнительную информацию о записанных и исправленных багах в IntelliJ на GitHub. Нас интересуют две основные вещи - во-первых, необходимо пометить метод, содержащий ошибку и во-вторых – для всех файлов, участвующих в

стектрейсе нам понадобится их исходный код. Поскольку в нашем датасете присутствуют стеки вызовов только для исправленных на текущий момент ошибок, то можно по issue_id найти последнее измененное состояние проблемы. Теперь можно отталкиваться от предположения, что в предыдущем до этого состоянии баг еще не был исправлен, а в конечном напротив содержится корректная версия кода. После сравнения двух версий файлов, метод, чей код изменился, и будет признан ошибочным.

Поскольку программы в отличие от текста имеют жестко связанную структуру, нельзя просто смотреть на соответствующие строчки и брать первое попавшееся имя метода как указание, что метод с таким именем изменился. Более того, необходимо вообще определить в коде такую сущность как метод, функция или конструктор. В связи с этим было принято решение написать парсер исходного кода в абстрактное синтаксическое дерево (AST) до уровня методов. Моя версия парсера основана на выделении ветвей дерева в зависимости от открытия/закрытия фигурных скобок, при открывающей скобке мы спускаемся вниз по дереву и поднимаемся обратно если встречаем закрывающую. Для разметки ветвей используются регулярные выражения, которые проходят по соответствующему куску кода между двумя скобками и ищут по заданным шаблонам имена методов, функций, конструкторов классов или имена самих классов, поскольку нам необходимо знать полное имя метода во избежание коллизий частовстречаемых имён. Далее построенные AST сравниваются между собой по путям в ветвях и по коду в листьях, где несовпадение указывает на метод с багом. Полный путь до метода в дереве сохраняется и далее полученное название будет сопоставлено с методами в стектрейсе, помечая некорректный код.

Одновременно с разметкой были собраны актуальные на момент наличия бага версии файлов для методов, также участвующих в стектрейсе. Но не для всех – часть файлов не может быть собрана так как соответствующие им методы входят в стандартную библиотеку либо предоставляются сторонними разработчиками и не представляется возможным получить доступ к их исходному коду.

Также для обучения деревьем градиентного бустинга необходимо собрать числовые и категориальные признаки кода. Использование эмбед-

дингов не поддерживается стандартными алгоритмами обучения деревьев, а значит необходимо представить код для них иным образом. Для этого были собраны дополнительные данные об исходном коде: длина методов, длина названия, наличие лямбда функций и параллелизации, позиция метода в файле, метка типа ошибки. Также были добавлены метаданные с Git, а именно даты изменения файлов и количество изменений. Есть основания полагать, что так модель сможет отличать старые файлы, которые скорее всего не содержат ошибку, от новых и частоизменяемых.

Поскольку в будущем на вход модели необходимо подавать последовательности фиксированного размера, а сами стектрейсы могут содержать тысячи методов, было решено определить отсечку по количеству методов, в пределах которых 99% стектрейсов будут содержать ошибку. Построив предложенную кумулятивную сумму, оказалось, что в 47 первых фреймах содержится 95% ошибок, а в окне в 80 фреймов их 99%.

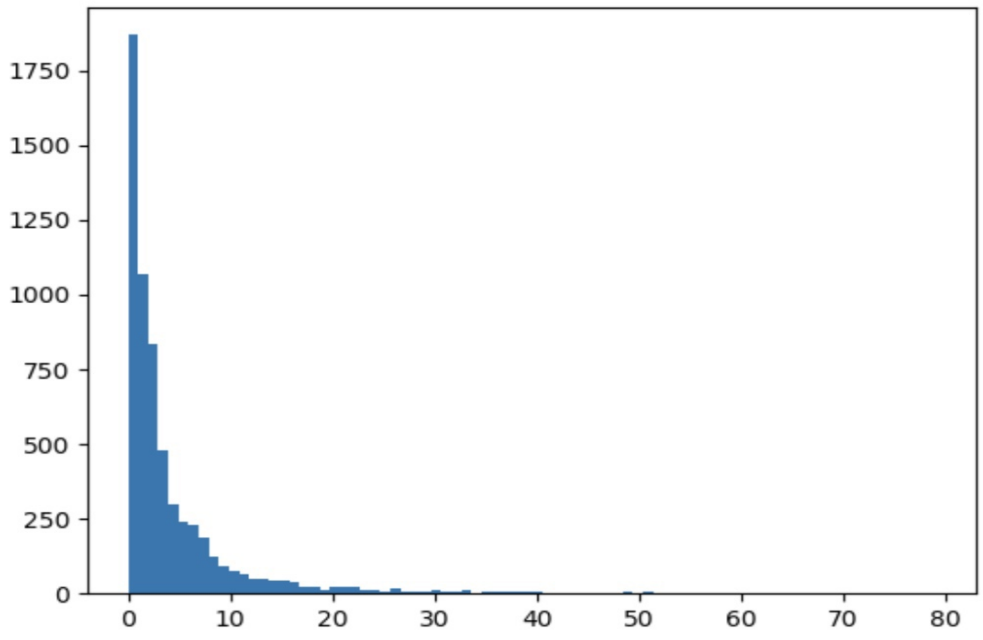


**Рис. 3:** Количество стектрейсов с ошибками в методах в зависимости от соответствующей позиции метода в стектрейсе

## 1.2 Оценочные данные

Всего результирующий датасет состоит из 6253 примеров, для каждого примера есть хотя бы один файл с кодом, в среднем на каждый пример приходится 8 файлов. Длина стектрейсов ограничена 80 первыми входящими в него файлами и каждый из методов имеет метку о наличии или отсутствии бага.

В связи с большим разнообразием моделей, нацеленных на поиск ошибок в исходном коде, имеется необходимость оценить разработанную модель на общеизвестных данных. Самым популярным датасетом для оценки алгоритмов локализации багов является Defects4j[10], состоящий из нескольких опенсорс-проектов, для которых собрана такая информация, как:

- Описание ошибки (человеком)
- Вывод упавшей программы со стектрейсом и общая информация (время, коммит в GitHub)
- Код, который привел к падению программы
- Код, который исправляет ошибку

Скрипты по разметке и обработке данных для Intellij IDEA были немного преобразованы и применены к этим данным. Таким образом, получился практически идентичный датасет, который был использован далее для финальной оценки качества работы модели.

## 1.3 Обработка данных

Любая модель нейронного обучения нуждается в представлении данных в числовом виде – в противном случае, невозможно было бы использовать никакой алгоритм обучения нейронных сетей, ведь все они основаны на принципе минимизации эмпирического риска и соответствующих функций потерь. В связи с этим, мы использовали для перевода исходного кода в векторный вид библиотеки code2vec и code2seq.

Ранее уже были упомянуты архитектурные особенности и возможные недостатки code2vec, но в нашей работе мы все равно провели эксперимент по использованию эмбеддингов от этой модели. Поскольку code2seq задумывался как улучшение этой модели, то интересно было проверить, насколько различные результаты они могут дать. При обработке данных экстрактор путей code2vec оказался сильно медленнее обновленного и часто падал с ошибкой – возможно, так как в репозитории IntelliJ IDEA довольно много неизвестных экстрактору названий и аннотаций.

Принцип работы code2seq заключается в использовании связки энкодер-декодер на последовательности входных токенов. Токены получаются из обхода AST подаваемого кода, где для каждого элемента набирается по возможности несколько разных путей из которых он достижим. Далее последовательности токенов кодируются через BiLSTM, а сами токены делятся на сабтокены(например, ArrayList -> Array List), для них достается эмбеддинг из обученной матрице эмбеддингов и результат суммируются как представление исходного токена. Так, для пропускаемого в момент времени $t$ через BiLSTM токена мы получаем текущее состояние выхода $a_t$ с учетом контекста предыдущих и следующих токенов и результат усредняется в один общий контекстный токен $c_t$. Входные обработанные данные пропускаются через многослойный перцептрон и на выходе получается вероятность получения следующего токена для всех токенов в словаре. Но поскольку нас интересует задача не столько генерации кода, сколько его представления, то мы для представления кода возьмем конечный контекстный токен $c_n$. Его размер 384 элемента, что довольно много, но все еще близко к популярной практике выбора размерности эмбеддингов в 300. Размер вектора code2vec тоже близок к этому числу и составляет 320 элементов.

На выходе теперь code2seq выдает таблицу, в которой каждому методу соответствует определенный характеризующий его вектор. Достаточно только выбрать те методы, вектора которых нам необходимы для дальнейшей работы со стеком вызовов. Методы, для которых векторов нет, получают нулевой вектор той же размерности, все последовательности методов обрезаются до 80 первых методов.

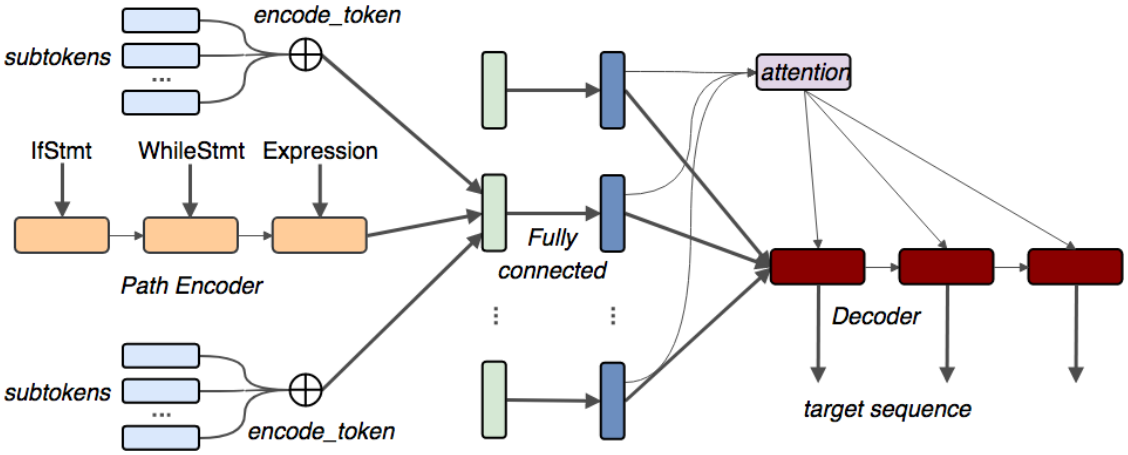


**Рис. 4:** Архитектура code2seq

# Глава 2. Реккурентные нейронные сети

## 2.1 RNN

Поскольку данные, о которых ведется речь в работе, представляют собой направленный список методов для каждого стектрейста, то работать с ними с ними можно как с последовательностью. Для этого лучше всего подходят реккурентные нейронные сети глубокого обучения, архитектура которых позволяет распространять сигнал от входных нейронов для каждого элемента входной последовательности.

Классический полносвязный слой[11] выглядит как

$$y = Wx + b$$

где $x$ - входной вектор размерности $n_{in}$, $y$ - выходный вектор размерности $n_{out}$, $W$ - матрица весов и $b$ - вектор размерности $n_out$, ответственный за смещение. Самая простая архитектура нейронной сети может основываться на нескольких полносвязных слоях, но она никак не будет учитывать контекст подаваемых данных.

В отличие от полносвязных слоев, слои реккурентной нейронной сети (RNN)[11] представлены в виде

$$y_t = \sigma(W_{yh}^t h_t + c_t)$$

$$h_t = \sigma(W_{hh} h_{t-1} + W_{hx} x_t + b_h)$$

$x_t$ - вход слоя в момент времени $t$, $y_t$ - выход слоя в момент $t$, ($W_{hx}$, $W_{hh}$, $W_{yh}$) - матрицы параметров из входного слоя во внутренний, из внутреннего во внутренний и из внутреннего в выходной слой, $h_t$ - вектор скрытого внутреннего состояния и $b_t$, $c_t$ - снова вектора смещения для обновления внутреннего и внешнего слоя. Таким образом внутреннее состояние слоя может передавать информацию о предыдущих входах нейронной сети для уточнения прогнозов выхода слоя. Есть также несколько различных архитектур RNN с различным числом входов и выходов, варьируемые в зависимости от решаемой задачи, но для нашей работы лучше всего подходит

архитектура многие-ко-многим, так как мы классифицируем каждый элемент в стеке вызовов.

Однако такой тип нейронных сетей на практике часто сталкивается с несколькими проблемами. Реализуемая в них память является кратковременной и заменяется новой спустя несколько итераций обучения. Также они страдают от проблемы взрывающихся/затухающий градиентов - явления, когда обучение сети ведет себя нестабильно на длинных последовательностях из-за частых перемножений больших или напротив маленьких значений. Для решения этой проблемы архитектура RNN была улучшена – внутри реккурентных компонентов не используется функция активации, что позволяет избавиться от размывания градиента во времени. Контроль за информацией, используемой ячейкой, осуществляется через фильтры, которые состоят из сигмоидальной нейронной сети и поточечного умножения. Новая архитектура сетей получила название Long short-term memory[12] так как способна запоминать информацию как на долгий срок, так и учитывать ближний контекст.

## 2.2 LSTM

Новая информация в LSTM-ячейке проходит несколько внутренних фильтров-слоев. Первый слой получает информацию о внутреннем состоянии предыдущей ячейке $h_{t-1}$ и входные данные $x_{t-1}$, и выбирает какую информацию пропустить дальше

$$f_t = \sigma(W_f \dot{[}h_{t-1}, x_t] + b_f)$$

значения близкие к единице будут пропущены далее. Затем слой входного фильтра

$$i_t = \sigma(W_i \dot{[}h_{t-1}, x_t] + b_i)$$

решает какую информацию необходимо обновить, а слой с $tanh$ отвечает за построение новых значений

$$\hat{C}_t = tanh(W_C \dot{[}h_{t-1}, x_t] + b_C)$$

которыми можно обновить состояние ячейки

$$C_t = f_t C_{t-1} + i_t \hat{C}_t$$

.

Выходное состояние ячейки также проходит через внутренние фильтры, основываясь на текущем состоянии ячейки. Значения предыдущего выхода ячейки $h_{t-1}$ и текущий вход $x_t$ проходят через сигмоидальный слой, теперь чтобы определить обновляемые значения выводимые из ячейки.

$$o_t = \sigma(W_o \dot{[}h_{t-1}, x_t] + b_o)$$

В это время значения состояния ячейки проходят через $tanh$ слой и на выходе перемножаются с $o_t$

$$h_t = o_t \tanh(C_t)$$

передавая только необходимую информацию.

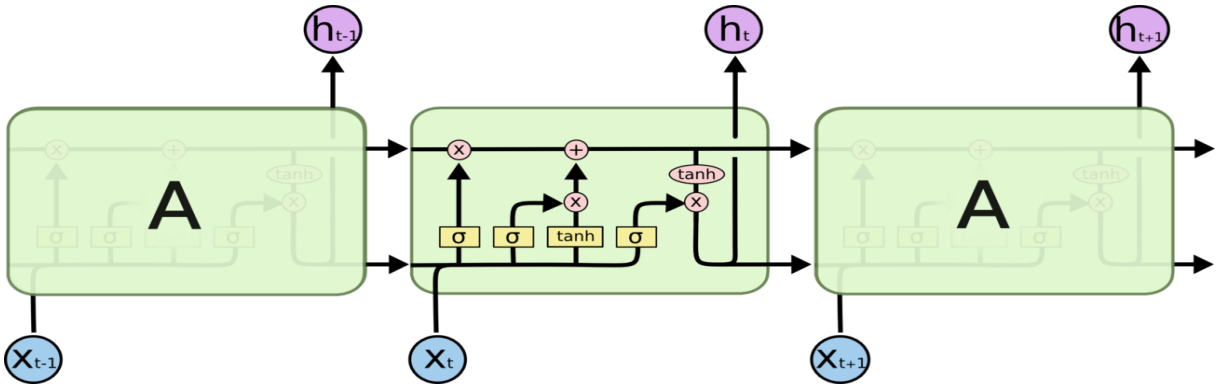


**Рис. 5:** Пример LSTM сети. Строение компонента LSTM.

Двунаправленные (Bidirectional LSTM) версии этой архитектуры содержит удвоенную версию слоев внутри реккурентной ячейки и способна проводить сигнал в обе стороны. Через каждую компоненту сети входная последовательность теперь будет проходить как в прямом, так и в обратном порядке. Они показали себя более эффективными в задачах, где стоит учитывать не только предыдущих, но и следующих контекст элемента внутри последовательности(например, машинный перевод).

Обучения же реккурентных нейронных сетей аналогично обычным полносвязным сетям и использует те же функции потерь и метод обратного рапространения ошибки, поэтому для различных задач можно использовать как функции потерь, заточенные под классификацию (например, кросс-энтропия, KL-дивергенция), так и функции для задач регрессии(например, функция среднеквадратичных потерь).

# Глава 3. Эксперименты над данными

## 3.1 Базовый алгоритм

Поскольку необходимо от чего-то отталкиваться и следить, чтобы качество модели не упало ниже определенного уровня качества, при котором само ее использование окажется излишним, необходимо назначить некоторый простой алгоритм предсказания методов с ошибкой. Если учесть график 3, то совершенно очевидно, что таким простым алгоритмом следует взять предсказание первых $n$ методов, где $n$ варьируется в зависимости от дальнейших метрик.

## 3.2 Модель-конкурент

Для того, чтобы оценивать нашу модель по сравнению с моделью, разработанной для тех же целей и работающую примерно на тех же данных, была выбрана и реализована модель BugLocator на основе алгоритма ее работы из статьи. Так как входными данными для BugLocator являются баг-репорты, то за их отсутствием и желанием предсказывать ошибочный метод без использования описания ошибки человеком, в качестве баг-репортов были приняты описания стектрейсов – в авторских баг-репортах для BugLocator они так же присутствовали.

## 3.3 Предлагаемая модель

Учитывая, что в поставленной задаче речь идет о работе с последовательностями элементов, где важно учитывать контекст, то было принято решение работать с популярной в обработке естественного языка (NLP) архитектурой BiLSTM. Задача разметки методов на наличие багов или их отсутстсвие очень приближена к классической задаче NLP по разметке слов в предложении по частям речи, поэтому идея была взята из решения подобных задач. Сеть состоит из одного LSTM модуля с выходом в полносвязный нейронный слой, который сводит вывод LSTM в слой с двумя нейронами – по одному для каждого класса. К последнему слою будет

применена функция активации softmax, чтобы ограничить значения на нем отрезком $[0; 1]$.

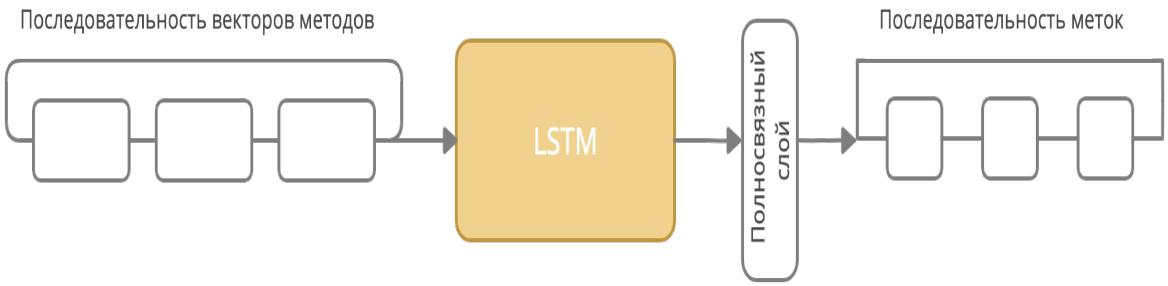


**Рис. 6:** Архитектура используемой модели.

В области нейронных сетей в задаче классификации обычно используется такая функция потерь как перекрестная энтропия. Перекрестная энтропия определяется как:

$$L(t) = -\frac{1}{N}\sum_{i=1}^{N}[t_i log(p_i) + (1 - t_i) log(1 - p_i)]$$

где всего $N$ точек, $t_i$ – значение класса для $i$-ой точки, $p_i$ - значение softmax для $i$-ой точки.

Наша нейронная сеть была обучена с перекрестной энтропией как функцией потерь. В качестве метрики для отслеживания качества предсказания сети во время обучения был использован accuracy или иначе говоря количество верно распознанных методов, содержащих ошибку. В текущей реализации подсчета метрик также поддерживается метрика accuracy @ k, где допускается нахождение метода с ошибкой среди топ k методов по наибольшей оценке забагованности.

Для тренировки модели использовался подход с перебором таких параметров как:

- Число эпох
- Шаг обучения
- Размер скрытого слоя LSTM

- Число эпох и коэффициент сокращения шага обучения

Лучшего всего из оптимизаторов показал себя Adam[16].

Поскольку все еще не была решена задача заполнения пропущенных данных, то в модель был включен специальный обучаемый эмбеддинг, который подставлялся в модель вместо нулевых векторов. После его добавления модель стала учиться гораздо лучше, поскольку перестала зануляться информация от отсутствующего метода и контекст стал учитываться намного эффективнее.

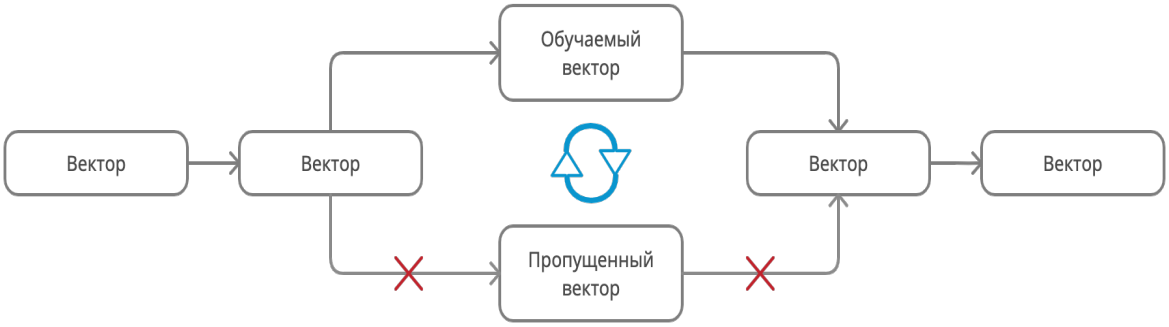


**Рис. 7:** Замена пропущенных эмбеддингов.

Весь код написан с использованием фреймворка PyTorch[17], созданного для быстрого и удобного создания нейронных сетей самой разнообразной архитектуры. Код модели воспроизводим и находится в репозитории[5], где также находятся скрипты для обработки данных и различные эксперименты по изменению архитектуры модели.

## 3.4 Ранжирование

Во многих упомянутых в обзоре работах фигурировало ранжирование результатов на основании полученных методами оценок и собранных дополнительно признаков. Поскольку ничего не ограничивает нас в использовании подобного подхода и во время разметки данных была собрана дополнительная информация как о самом коде, так и о его изменениях, было принято решение обучить модель градиентного бустинга[13] на предсказаниях модели LSTM.

[5] `https://github.com/lissrbay/bug_ml`

Деревья градиентного бустинга – классический подход машинного обучения, все еще являющийся одним из лучших подходов для восстановления сложных зависимостей. Суть алгоритма градиентного бустинга заключается построении композиции базовых алгоритмов, где каждый новый алгоритм корректирует ошибки предыдущих.

$$a_n = \sum_{i=0}^{n} b_i(x)$$

$a_n$ - итоговая композиция алгоритмов, $b_i$ - базовый алгоритм, в нашем случае являющийся решающим деревом. Пусть задана некоторая функция потерь $L$ и к некоторому моменту обучены $n-1$ базовых алгоритмов $b_1, ..., b_{n-1}$. К следующей композиции добавляется еще один алгоритм $b_n$, при этом он должен минимизировать ошибку композиции:

$$\sum_{i=1}^{n} L(y_i, a_{n-1}(x_i) + b_i) \to min_b$$

или иначе говоря какие значения должен принимать вектор $b_n(x_i) = s_n$ для минимизации ошибки на выборке.

$$F(s) = \sum_{i=1}^{n} L(y_i, a_{n-1}(x_i) + s_i) \to min_s$$

Так как направление наискорейшего убывания функции $F(s)$ задается вектором антиградиента, то его можно принять в качестве вектора $s$, а обучение $b_n$ сводится к приближению к значениям вектора $s$ на обучающей выборке по квадратичной функции ошибки:

$$b_n(x) = argmin_b \frac{1}{n} \sum_{i=1}^{n} (s_i - b(x_i))^2$$

CatBoost – известный и легкий в освоении фреймворк для обучения деревьев градиентного бустинга. Он известен тем, что практически не требует перебора параметров от пользователя для процесса обучения, так как

прекрасно работает на стандартных настройках и сам кодирует категориальные переменные. Но в применении к нашим данным обычное обучение задаче классификации на наличие бага или его отсутствие показало себя хуже, чем LSTM-модель.

Исходную задачу легко переложить на термины задачи ранжирования, где у нас есть коллекция запросов $Q = q_1, ..., q_n$ - это наши стектрейсы и информация о них, и набор документов $D = d_1, ..., d_n$ - это методы, встречаемые в стектрейсе. Упорядочить по релевантности в новых терминах значит отранжировать методы по вероятности быть забагованными. Таким образом, данные были немного преобразованы в подходящий для новой модели формат и она была обучена с ранжирующей функцией потерь QuerySoftMax[14].

$$-\frac{\sum_{q \in Q} \sum_{i \in q} w_i t_i log(\frac{w_i e^{\beta \alpha_i}}{\sum_{j \in q w_i e^{\beta \alpha_j}}})}{\sum_{q \in Q} \sum_{i \in q} w_i t_i}$$

$w_i$ - вес запроса $i$, $t_i$ - релевантность документа запросу, $\beta$ - параметр масштабирования и $a_i$, $a_j$ - предсказания модели. Эта функция потерь хорошо показывает себя на небольшом объеме данных и не требует больших вычислительных ресурсов.

Результаты применения только LSTM-модели были улучшены на несколько процентов и было принято решение оставить комбинированную модель.

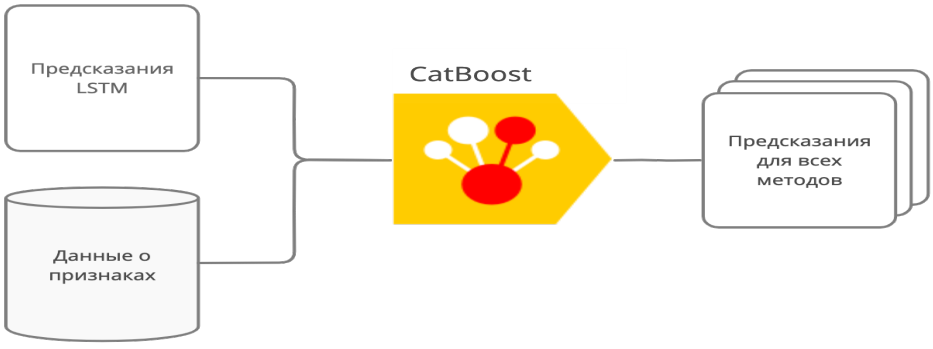


**Рис. 8:** Схема работы ранжирующей части модели.

## 3.5 Результаты

Начнем с результатов работы базового алгоритма – их легко измерить и они показывают, что наиболее выгодно предсказывать первые N методов стектрейса, что довольно логично.

| Accuracy | Топ N |
|---|---|
| 1 | 30.5% |
| 2 | 48.2% |
| 3 | 62.9% |
| 4 | 70.1% |
| 5 | 73.2% |

**Таблица 2:** Результаты работы базового алгоритма на данных IntelliJ IDEA

Метрика, которую мы используем в нашей работе - $accuracy@k$. В отличии от простой метрики точности предсказаний, мы считаем, что если среди k предложенных методов один содержал ошибку - значит, считаем предсказание верным.

Ниже представлена таблица сравнения результатов как на нашем датасете, так и на датасете Defects4j. Поскольку для нас целью было нахождение метода, с наибольшей вероятностью содержащего в себе ошибку, то мы ограничились измерением качества по метрике $accuracy@1$. Но в большинстве статей авторы используют более широкий охват методов и считают качество алгоритма по $accuracy@5$, так как разработчики редко обращают внимание дальше чем на первые 5 методов стектрейса[15], поэтому на Defects4j мы тоже оценивали модель учитывая пять методов с наибольшей оценкой.

Замена нулевых эмбеддингов на обучаемый вектор помог нейросети лучше учитывать контекст методов и улучшила итоговый результат по сравнению с простым применением LSTM. Сбор дополнительных данных и ранжирование результатов также немного повысило итоговое качество модели.

Что касается использования code2seq вместо code2vec, то по результатам оно полностью себя оправдывает. В то время как метрики модели,

| Модель | IntelliJ Accuracy@1 |
|---|---|
| Baseline | 30.5% |
| code2vec + LSTM | 39.7% |
| code2vec + LSTM + обуч. эмбед. | 44.5% |
| code2seq + LSTM | 56.15% |
| code2seq + LSTM + обуч. эмбед. | 62.4% |
| CatBoost + LSTM предсказания | 66.5% |
| BugLocator | 25.4% |

**Таблица 3:** Результаты работы моделей

обученной на векторах code2vec едва ли на 10% отличаются от метрик базового алгоритма, то code2seq показывает практически двоекратный прирост качества предсказаний. Добавление обучаемого эмбеддинга обеим моделям помогает примерно одинаково. В итоге code2seq существенно опережает code2vec в нескольких характеристиках, и от использования code2vec пришлось отказаться.

| Модель | Defects4j Accuracy@5 |
|---|---|
| Baseline | 74.1% |
| code2seq + LSTM + обуч. эмбед. | 83.8% |
| CatBoost + LSTM предсказания | 86.7% |
| BugLocator | 49.6% |
| SBBL | 57.2% |
| SBBL + PageRank | 46.1% |
| Historical SBBL | 36.0% |

**Таблица 4:** Результаты работы моделей

Как видно из таблиц, мы добились не только улучшения качества по сравнению с базовым алгоритмом и моделью-конкурентом, но и существенно обогнали конкурентов по качеству даже на общем датасете. То, что результаты конкурентов обгоняются даже базовым алгоритмом может быть основано на том, что они не предполагают работу со стектрейсами, а следовательно круг поисков ошибочных методов для них сильно выше, или как в случае с BugLocator - стектрейсы учитываются на довольно простом уровне. Можно заметить, что BugLocator показывает довольно низ-

кое качество на нашем датасете - это связано с отсутствием описания багов человеком, которое часто входит в багрепорты и использование которого предполагается авторами метода. Поскольку в этой работе предполагается уход от использования данных, полученных непосредственно от человека(кроме исходного программ), то в описании ошибки участвовали только стектрейсы и классы ошибок.

## Выводы

В рамках работы над решением задачи были сделаны следующие выводы:

- Реккурентные нейронные сети глубокого обучения могут быть успешно использованы в задачах векторного представления исходного кода и локализации ошибок
- Использование стектрейсов как дополнительного источника данных позволяет ограничить пространство поиска ошибочных методов и получить хороший результат путем использования других архитектур, учитывающих их структуру
- Скрипты для обработки данных практически независимы от проекта и при наличии в проекте стектрейсов, можно легко дособирать данные и получить предсказания модели, обученной на датасете IntelliJ
- Разработанная модель может быть использована в IntelliJ IDEA как часть инструмента для отладки программного кода на Java

# Заключение

Был произведен полный цикл разработки модели машинного обучения, включающий в себя сбор данных, обучение и оценку модели. Для нее была использована архитектура LSTM с улучшением в виде обучаемого эмбеддинга вместо отсутствующих векторов. На ее предсказаниях и дополнительных признаках были также обучены деревья градиентного бустинга, что дало дополнительный прирост качества предсказаний. Используемые скрипты легко адаптируются, что позволит использовать их в дальнейшем в продукте компании JetBrains. Итоговая модель показывает хорошие результаты как на предоставленных компанией данных, так и на общедоступном датасете Defects4j.

Дальнейшие исследования будут включать в себя добавление языка Kotlin, поскольку он тоже широко применяется в кодовой базе IntelliJ IDEA, создание и отладку пайплайна по автоматизированному обучению и использованию модели, а также выпуск статьи по теме после проведения дополнительных экспериментов.

# Список литературы


[1] Marimont, R., Shapiro, M., «Nearest Neighbour Searches and the Curse of Dimensionality»IMA J Appl Math, 1979

[2] Devlin J., Uesato J., Singh R., Kohli P.«Semantic Code Repair using Neuro-Symbolic Transformation Networks». arXiv:1710.11054, 2017

[3] Alon U., Zilberstein M., Levy O., Yahav E.«code2vec: Learning distributed representations of code »., arXiv:1803.09473, 2018

[4] Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L.«Attention Is All You Need». NIPS 2017, 2017

[5] Alon U., Brody S., M., Levy O., Yahav E. «code2seq: Generating Sequences from Structured Representations of Code». arXiv:1808.01400, 2019

[6] Chakraborty S., Li Y. , Irvine M., Saha R., Baishakhi R. «Entropy Guided Spectrum Based Bug Localization Using Statistical Language Model». preprint arXiv:1802.06947, 2018

[7] M. Wen et al. «Historical Spectrum based Fault Localization». in IEEE Transactions on Software Engineering, IEEE Transactions on Software Engineering, 2019

[8] He H., Ren J., Zhao G.«Enhancing Spectrum-Based Fault Localization Using Fault Influence Propagation ». IEEE Access 8, 2020

[9] Wong C., Xiong Y., Zhang H., Hao D., Zhang L., Mei H. «Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis». IEEE, 2014

[10] Just R., Jalali D., Ernst M. «Defects4j: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs ». ISSTA 14, 2014

[11] Николенко С., Кадурин Аю, Архангельская Е. «Глубокое обучение. Погружение в мир нейронных сетей». — «Питер», 2018

[12] Hochreiter S., Schmidhuber J.«Lons Sort-Term Memory». Neural Computation 9(8): pp. 1735-1780, 1997

[13] Friedman J., «Greedy Function Approximation: A gradient boosting machine ». The Annals of Statistics, 2001

[14] Bruch S., Wang X., Bendersky M., Najork M.«An Analysis of the Softmax Cross Entropy Loss for Learning-to-Rank with Binary Relevance», ICTIR 19, 2019

[15] Pavneet S. K., Xin X., Lo D., and Shanping L. «Practitioners' expectations on automated fault localization». In Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, 2016, pp. 165–176.

[16] Kingma D., Ba J.«Adam: A method for stochastic optimization». //arXiv preprint arXiv:1412.6980. – 2014.

[17] Paszke A., Gross S., Massa F., Lerer A. et al.«PyTorch: An Imperative Style, High-Performance Deep Learning Library»2019, arXiv:1912.01703