Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Тема: **Применение искусственных нейронных сетей для поиска клонов в исходном коде программ**

Студент гр. 23541/3 А.Г. Зорин

Санкт-Петербург
2018

Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Работа допущена к защите
зав. кафедрой

_____ В.М. Ицыксон

«___» _____ 2018 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Тема: **Применение искусственных нейронных сетей для поиска клонов в исходном коде программ**

Направление: 09.04.01 – Информатика и вычислительная техника
Магистерская программа: 09.04.01.15 – Технологии проектирования системного и прикладного программного обеспечения

Выполнил студент гр. 23541/3 _____ А.Г. Зорин

Научный руководитель,
к. т. н., доц. _____ В.М. Ицыксон

Консультант по нормоконтролю,
к. т. н., доц. _____ А.Г. Новопашенный

Санкт-Петербург
2018

РЕФЕРАТ

Отчет, 55 стр., 6 рис., 3 табл., 30 ист., 1 прил.

МАШИННОЕ ОБУЧЕНИЕ, ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ, ПОИСК КЛОНОВ

Магистерская работа посвящена исследованию возможности применения искусственных нейронных сетей в задаче поиска клонов в исходном коде программ. Исследование было проведено на основе проектов с открытым исходным кодом, которые написаны на языке программирования Java. В ходе проведения данного исследования был разработан прототип инструмент для поиска дубликатов в исходном коде программы.

Полученный прототип был протестирован на специальной выборке - BigCloneBench. Тестирование показало ...

ABSTRACT

Report, 55 pages, 6 figures, 3 tables, 30 references, 1 appendicies

CODE CLONE DETECTION, MACHINE LEARNING, NEURAL NETWORKS

This work is dedicated to analysis of artificial neural network usage for detection clones in source code of a program. The AI-based approach was proposed.

СОДЕРЖАНИЕ

СПИСОК ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ

API	Application Program Interface
AST	Abstract Syntax Tree
CBOW	Continuous Bag of Words
CFG	Control Flow Graph
IDE	Integrated Development Environment
LOC	Lines of Code
LSTM	Long Short-Term Memory
PDG	Program Dependency Graph
PSI	Program Structure Interface
XML	eXtensible Markup Language
НС	Нейронная сеть
ПО	Программное обеспечение
РНС	Рекуррентная Нейронная Сеть

ВВЕДЕНИЕ

Одной из актуальных проблем разработки и сопровождения программного обеспечения (ПО) является наличие клонов в исходном коде программы. Фрагмент кода, который был скопирован и вставлен с незначительными или большими изменениями называется дубликатом ПО или клоном. По данным различных исследований современной ПО насчитывает до 30% клонов [23].

Клоны могут появляться в программных системах по разным причинам. Это может быть копирование участков когда. Такой подход может стать разумной отправной точкой для написания программы. Другая причина - разработка уже существующего участка кода другим программистом. Также причиной появления клонов может служить невыразительность используемого языка программирования или API (Application Program Interface - программный интерфейс приложения).

Среди многих проблем, связанных с наличием клонов в исходном коде программы, одной из главных является ее последующее сопровождение. В том случае, когда имеется ошибка в одном из дублированных фрагментов, ее необходимо найти и справить во всех идентичных участках, а не только в одном.

Обнаружение клонов является важной проблемой для поддержки ПО. Однако, несмотря на тот факт, что в предыдущих исследованиях были показаны отрицательное воздействие клонировния, это не всегда так. Постоянная необходимость изменения клонов также отсутствует. Тем не менее, возможность обнаружения двух идентичных участков кода критична во многих областях, например, упрощение и совместимость кода, определение плагиата, нарушение авторских прав, обнаружение вредоносного ПО и т.д.

Целью данной работы является разработка метода поиска клонов основанного на использовании нейронных сетей. На основе данного метода планируется разработка и тестирование прототипа инструмента поиска клонов.

РАЗДЕЛ 1. ОБЗОР И СРАВНИТЕЛЬНЫЙ АНАЛИЗ СУЩЕСТВУЮЩИХ ПОДХОДОВ

В данном разделе вводятся основные определения, связанные с предметной областью, приводятся различные классификации. Рассматриваются существующие подходы к обнаружению клонов и приводятся их основные характеристики.

Также, в данном разделе приводится сравнительный анализ архитектур нейронных сетей

1.1. Основные определения и классификация

Для рассмотрения существующих подходов к обнаружению клонов, в первую очередь необходимо дать основные определения и классификации, связанные с предметной областью.

- **Фрагмент кода** - часть исходного кода, необходимая для запуска программы. Такая часть может содержать в себе функцию или метод, блоки или последовательности операторов.
- **Клон кода** - две и более части кода, которые аналогичны друг другу в соответствии с типами клонов.
- **Клоновый класс** - максимальное множество фрагментов кода, в котором любые два фрагмента являются клонами.
- **Кандидат в клоны** - пара фрагментов кода, которые были определены как клон.

С точки зрения идентичности дублированных фрагментов, клоны делятся на следующие типы:

- I: Полностью идентичные фрагменты программы без учета различий разметки и комментариев [22] [1].
- II: Клоны идентичные клонам первого типа, в которых также не учитваются различия идентификаторов, типов и литералов [22] [1].

- III: Клоны, идентичные клонам второго типа, в которых также не учитываются изменения, добавления и перемешивания операторов [22] [1].
- VI: Фрагменты программы, решающие схожую задачу, но реализованные различными способами (семантические клоны) [22] [1].

При рассмотрении клонов с точки зрения их размера, выделяют следующие виды:

- С фиксированной гранулярностью: идентичные фрагменты кода фиксированного размера (методы классы и т.д.).
- С производной гранулярностью: идентичные фрагменты произвольного размера

1.2. Причины возникновения

Клоны в коде не появляются сами по себе. Существует несколько причин, которые могут подтолкнуть разработчиков ко внедрению клонов в систему. Далее приведены некоторые причины их появления.

1.2.1. Стратегия разработки

Клоны в программных системах могут появляться из-за повторного использования или programming approach. Например:

Повторное использование

Повторное использование кода, логики, дизайна и/или всей системы являются основными причинами возникновения дубликатов кода.

Повторное использование за счет копирования/вставки. Повторное использование кода посредством копирования/вставки (с незначительными изменениями или без них) самый простой и распространенный вид механизма повторного использования в процессе разработки. Такой способ является быстрым способом повторного использования надежных синтаксических и семантических конструкций [10].

Ответвление. Термин ответвление был использован Каспером и Годфри [14] для того, чтобы обозначить повторное использование подобных решений с их возможным расхождением. Например, при разработке драйвера для семейства аппаратных устройств, похожее семейство устройств может уже иметь драйвер. Таким образом, этот драйвер может быть использован с незначительными изменениями. Аналогично, клоны могут появляться при переносе ПО с одной платформы на другую.

Подход к программированию

Клоны, также могут появляться во время разработки ПО. Например:

Слияние двух похожих систем. Иногда две программные системы с похожей функциональностью объединяются с целью создания новой. Также, такие системы могут быть разработаны разными командами, и, тогда, клоны при объединении появляются из-за реализации похожих функциональностей в обеих системах.

Разработка системы с помощью генеративного подхода. Генерация кода может породить огромное количество клонов из-за использования одинаковых шаблонов для генерации одинаковой или похожей логики.

1.2.2. Преимущества поддержки ПО

Еще одна из причин появления клонов - получение некоторых выгод обслуживания.

Чистая и понятная архитектура ПО. Иногда клоны намеренно внедряются в ПО для поддержания чистой и понятной архитектуры [14].

Высокая стоимость вызовов функций. В программах реального времени вызовы функций могут быть слишком затратными. В том случае, когда компилятор не предлагает встроить код автоматически, это приходится делать вручную, что непременно влечет за собой появление клонов.

1.2.3. Преодоление ограничений

Клоны могут появиться из-за различных ограничений в языках программирования.

Ограничения языка

Значительные усилия при написании повторноиспользуемого кода. Написание кода который потом можно будет повторно использовать слишком трудо- и времязатратно. Поэтому, в неоторых случаях, проще создать клон, чем тратить силы на написание общего кода.

Отсутствие механизма повторного использования в языке программирования. Иногда, в языках программирования не хватает механизмов абстракции, например наследования, общих типов или передачи параметров. В следствии этого разработчикам приходится применять их как идиомы. Такие повторяющиеся действия могут создавать возможно небольшие и потенциально часто используемые клоны [11] [3].

Ограничения в работе программистов

Сложности в понимании больших систем. Обычно сложно понять большую программную систему. Такие сложности толкают программистов к использованию пример-ориентированного программирования путем адаптации уже существующего кода.

Временные ограничения. Одними из основных причин появления клонов являются временные рамки доступные разработчикам. В большинстве случаев, разработчикам отведено ограниченное количество времени для завершения проекта или его части. Из-за таких ограничений разработчики ищут легкие пути решения проблем и, следовательно, ищут похожие существующие решения.

Неправильный способ измерения продуктивности. Иногда, продуктивность разработчиков измеряется в количеством написанных строк кода в час. В таких случаях, программисты фокусируются на увеличении количества строк кода и, следовательно, пытается использовать уже существующий код.

1.2.4. Случайное клонирование

В конечном итоге, клоны могут случайно появляться в ПО.

Протоколы взаимодействия с API и библиотеками. Для использования API обычно требуются серии вызовов функций и/или другие последовательности команд. Например, при создании кнопки с помощью Java SWING API, серии команд включают в себя: создание кнопки, добавление ее в контейнер и назначение обработчиков событий [13].

1.3. Преимущества обнаружения клонов

Помимо очевидного понимания возможностей улучшения качества кода с помощью рефакторинга клонов, есть и другие преимущества поиска клонов в коде. Например:

Выявление потенциальных библиотечных методов. В своих работах Дэвей [7], Берд и Мунро [4] отметили, что многочисленное использование скопированного кода доказывает его удобство использования. Такой код может быть включен в библиотеку и беспрепятственно повторно использоваться в дальнейшем.

Помощь в понимании программы. В том случае, когда функциональность программы осмысленна, возможно создать полную картину о других файлах, содержащих похожие копии этого фрагмента. Например, если фрагмент кода отвечает за управление памятью, можно сделать вывод, что все файлы, содержащие копию, должны иметь реализацию структуры с динамическим распределением память [20].

Поиск шаблонов. В том случае, когда все дублированные фрагменты одного исходного участка могут быть найдены, может быть обнаружена функциональная схема использования этого фрагмента [20].

Поиск плагиата и нарушения авторских прав. Поиск схожего кода может быть полезным в поиске плагиата и нарушения авторских прав [29].

Уменьшение размера кода. Методы поиска клонов позволяют сделать исходный код более компактным.

1.4. Критерии сравнения

При анализе подходов к обнаружению дублированных участков необходимо выделить основные характеристики для их сравнения. Ключевыми характеристиками метода можно считать:

- типы обнаруживаемых клонов
- полнота получаемых результатов
- точность получаемых результатов

В таком случае, основное требование к подходу - обнаружение клонов первых трех типов.

При сравнении архитектур нейронных сетей, необходимо принимать во внимание область применения таких сетей и их эффективность в этой области.

1.5. Способы обнаружения клонов

Как правило, процесс обнаружения клонов состоит из двух этапов: трансформации и сравнения. На первом этапе код преобразуется в промежуточное внутреннее представление, которое позволяет использовать более эффективные и специализированные алгоритмы сравнения. В то же время, выбор промежуточного представления накладывает ограничения на используемые алгоритмы и во многом определяет качество итоговых результатов.

Методы поиска клонов могут быть классифицированы в зависимости от способа внутреннего представления кода:

- на основе анализа текста
- на основе анализа токенов
- на основе анализа синтаксических деревьев
- на основе анализа графов
- на основе программных метрик
- смешанные методы

1.5.1. Метод на основе анализа текста

Подходы такого типа производят малое (или не производят вообще) количество изменений или нормализаций перед фактическим сравнением. В большинстве случаев используется непосредственно сам исходный код программы, представленный в виде последовательности строк. В полученной последовательности производится поиск одинаковых фрагментов наибольшей длины. Поиск таких фрагментов осуществляется при помощи методов анализа данных или строковых алгоритмов. К таким относятся методы сравнения отпечатков строк [13], поиска по шаблону [9], нахождения частой последовательности [6] и т.д.

Однако, высокая чувствительность к изменениям в исходной программе является главным недостатком подходов такого типа. Без применения модификаций такое представление позволяет обнаруживать только клоны первого типа. Для обнаружения клонов второго типа необходимо применять различного рода модификации: литералы и идентификаторы заменяются на специальные константы. Благодаря описанным модификациям, при обнаружении схожих фрагментов, такие различия не учитываются.

1.5.2. Метод на основе анализа токенов

В данном семействе подходов исходный код обрабатывается и представляется в виде последовательности, так называемых, токенов (уникальных последовательностей символов). Следующим этапом применяются строковые алгоритмы для поиска дубликатов в последовательности токенов. В отличии от методов на основе анализа текстов, такие методы позволяют использовать более эффективные и устойчивые к изменениям программы методы. Эффективность таких методов достигается за счет более компактного внутреннего представления, а их устойчивость - за счет фильтрации и нормализации токенов.

1.5.3. Метод на основе анализа синтаксических деревьев

В рамках данного подхода исходный код программы представляется в виде дерева разбора или абстрактного синтаксического дерева (AST). Такой подход использует структурную информацию

о программе, что позволяет обнаруживать клоны первых трех типов. Однако, семантика программы не учитывается, что приводит к невозможности обнаружения клонов четвертого типа, а также клонов с измененным порядком операторов.

Для обнаружения дублированных фрагментов кода с помощью анализа деревьев, используются алгоритмы поиска одинаковых поддеревьев. Такие методы, как правило, базируются на алгоритмах динамического программирования [30], либо пытаются свести задачу к поиску одинаковых подстрок [5]. Еще один подход, относящийся к рассматриваемому семейству методов - дополнение узлов дерева метриками, которые характеризуют соответствующие поддеревья [15]. Такой прием сильно упрощает задачу, позволяя найти ее решение за время, пропорциональное длине исходного кода.

При конвертации AST в XML (eXtensible Markup Language) и использовании технологий глубинного анализа данных, появляется возможность нахождения точные и параметризованные клоны на более абстрактном уровне. Во избежании сложностей, связанных с полным сравнением AST, поддеревья представляются в виде сериализованных последовательностей токенов (суффиксного дерева), что позволяет более эффективно находить синтаксические клоны.

1.5.4. Метод на основе анализа графов

Методы, основанные на анализе графов углубляются в абстракцию исходного кода за счет учета семантической информации, заключенной в графах зависимостей, захватывающих информацию управления и потока данных. Для поиска схожих подграфов используется алгоритм изоморфизма для подграфов, применяемый на графе программных зависимостей (PDG). Один из ведущих способов поиска клонов, основанного на данном семействе методов, - поиск изоморфных подграфов PDG с помощью (обратного) разбиения программы на элементы [16].

1.5.5. Метод на основе программных метрик

Еще одно семейство методов основано на сборе различных метрик, связанных с фиксированными фрагментами кода. Такой подход позволяет после сбора метрик сравнить их между собой,

вместо сравнения исходного кода напрямую. В различных предложенных методах использовались различные программные метрики для поиска клонов. Например, в качестве метрики можно принимать количество строк исходного кода (LOCs), количество вызовов функций, количество ребер графа потока управления (CFG). Такие метрики рассчитываются для каждого функционального элемента программы. Те элементы у которых схожие метрики считаются клонами. Рассматриваемый подход позволяет отыскать клоны на уровне функций, но не справляется с поиском клонов меньших размеров [18]

1.5.6. Смешанные методы

Помимо рассмотренных методов, существует такая группа, которая использует смешанный подход. Иными словами, используются несколько из рассмотренных выше структур для создания внутреннего представления программы. Одним из самых популярных гибридных подходов является получение и сериализация синтаксического дерева в последовательность токенов [25]. Такое смешивание позволяет анализировать структурную информацию, получаемую из AST, используя эффективные строковые алгоритмы.

1.6. Искусственные нейронные сети

На текущий момент искусственные нейронные сети (НС) достигли высокой производительности в таких задачах поиска схожих элементов, как, например, поиск одинаковых изображений, фотографий и текста. Данное достижение - одна из основных причин, по которой было решено использовать искусственные нейронные сети.

Для того, чтобы определить, какая из архитектур НС больше всего подходит для решения поставленной задачи, необходимо провести сравнительный анализ.

1.6.1. Сеть прямого распространения

Как следует из названия, такая сеть передает информацию только в одном направлении, от входа к выходу. НС состоят из

полносвязных слоев (каждый нейрон из одного слоя связан с каждым нейроном следующего слоя), однако, сам слой между собой никак не связан. Каждый слой состоит из входных, скрытых или выходных ячеек [21].

Круг применения таких сетей весьма узок, их можно применять, например, для простых задач классификации или предсказаний.

1.6.2. Нейронная сеть Хопфилда

Рассматриваемая сеть является полносвязной НС с симметричной матрицей связей. Каждый узел, в такой сети, является входным до начала процесса обучения, скрытым - во время и выходным после обучения. Обучение сети производится с помощью установления желаемого значения нейрона, после чего могут быть рассчитаны веса [12].

Как только веса заданы, обученная сеть становится способной "распознавать" входные сигналы - то есть, определять, к какому из запомненных образцов они относятся.

1.6.3. Сверточная нейронная сеть

Сверточная НС отличается от остальных. Основной ее задачей является обработка изображений. Довольно типичной задачей для нее является классификация или обнаружение объектов на изображениях. Как правило, такие сети начинают свою работу с, так называемого, входного "сканера", который не пытается анализировать все входные данные разом, а анализирует их небольшими фрагментами, соответствующими его размерам [2].

1.6.4. Рекуррентная нейронная сеть

В данном виде НС связи между нейронами образуют своеобразный направленный граф. Благодаря такому строению появляется возможность обработки серии событий во времени или последовательные цепочки. Однако узким местом таких сетей является проблема исчезающего градиента, то есть информация со временем теряется [17].

1.7. Сравнительный анализ

1.7.1. Сравнение методов поиска клонов

В данной главе были рассмотрены подходы, применяемые в задачах поиска программных клонов, приведены основные определения, связанные с предметной областью. В рамках обзора рассмотрены основные характеристики, преимущества и недостатки методов.

В таблице 1.1 представлен сравнительный анализ подходов на основе следующих характеристик:

- гранулярность - размер минимального клона
- тип клонов, которые возможно обнаружить с помощью данного подхода
- полнота обнаружения
- точность обнаружения

Таблица 1.1

Сравнительный анализ подходов

Представление	Гран.	Тип клонов	Полнота	Точность
Текст	Свободная	I	Низкая	Высокая
Токены	Свободная	I-II	Высокая	Низкая
Синт. деревья	Свободная	I-III	Низкая	Высокая
Графы	Свободная	I-IV	Средняя	Высокая
Метрики	Фикс.	I-IV	Средняя	Средняя

Учитывая предъявляемые требования, наиболее подходящими методами можно считать методы с внутренним представлением в виде синтаксических деревьев, графов или метрик. В эту же группу добавляются и гибридные методы. Однако, у методов, основанных на представлении кода в виде синтаксических деревьев, полнота обнаружения клонов в коде очень низкая. Следовательно, такой метод в чистом виде не подходит.

1.7.2. Нейронные сети

В предыдущем разделе были приведены основные виды архитектур НС. Рассмотрены области их применения, их положитель-

ные и отрицательные стороны. Основываясь всем вышеперечисленном, можно сделать вывод о наиболее подходящей архитектуре НС. Так как основная задача НС в предлагаемом методе - анализирование кода, то наиболее подходящей архитектурой сети будет являться рекуррентная нейронная сеть.

РАЗДЕЛ 2. ПОСТАНОВКА ЗАДАЧИ И ВЫБОР ПУТИ РЕШЕНИЯ

В данном разделе рассматриваются основные требования, предъявляемые к методу обнаружения клонов. Приводится постановка и подробное описание задач, решаемых в рамках данной работы, в том числе:

- задача разработки метода обнаружения клонов, основанного на использовании искусственных нейронных сетей
- задача разработки прототипа инструмента обнаружения

2.1. Задача разработки интеллектуального метода обнаружения клонов

Целью данной работы является разработка метода для обнаружения клонов. Главное требование, которое предъявляется к разрабатываемому методу - использование искусственных нейронных сетей. Кроме того, разрабатываемый метод интеллектуального обнаружения клонов должен обеспечивать решение следующих задач:

- обнаружение клонов I-III типов;
- максимизация полноты и точности обнаружения;
- извлечение информации о клонах в виде клоновых классов.

Несмотря на тот факт, что разрабатываемый метод должен быть интеллектуальным, в качестве предварительной обработки необходимо привести исходный код к одному из подходящих видов внутреннего представления. В предлагаемом подходе было решено использовать несколько представлений (гибридный метод). Этими представлениями являются AST и последовательность токенов.

Использование AST позволяет сохранить информацию о структуре исходного текста программы, именно благодаря этому становится возможным увеличить полноту и точность результатов.

Использование токенов в качестве представления исходного кода позволяет, в свою очередь, значительно сократить размер входных данных.

Общую структуру внутреннего представления кода можно увидеть на рис. 2.1

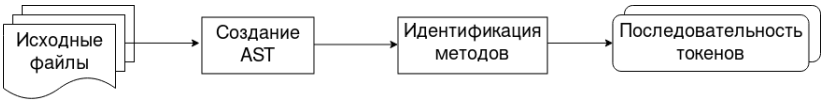

Рис. 2.1. Предлагаемый метод обнаружения клонов

2.2. Задача разработки прототипа инструмента

Одна из задач данной работы заключается в разработке и тестировании прототипа на базе предложенного метода. Учитывая выбранный подход, задача реализации прототипа состоит из двух частей:

1. Разработка инструмента, производящего преобразование исходного кода в выбранное представление;
2. Разработка инструмента, использующего нейронные сети для поиска клонов.

2.2.1. Разработка инструмента для создания внутреннего представления

К разрабатываемому инструменту предъявляются следующие требования:

- получение исходного кода на языка Java из заданной директории
- построение AST для полученного кода
- фильтрация AST
- преобразование AST в последовательность токенов

В качестве целевого языка был выбран язык Java, так как на протяжении многих лет он является одним из самых распространенным и используемым языком программирования (согласно TIOBE) [26].

2.2.2. Разработка инструмента для поиска программных клонов

Вторая важная часть разработки прототипа - реализация инструмента поиска клонов. К такому инструменту предъявляются следующие требования:

- использование рекуррентных нейронных сетей для поиска клонов
- поиск клонов I-III типов
- высокая точность и полнота поиска

Для данного инструмента, в качестве целевого языка программирования был выбран язык Python. Такой выбор был сделан из-за популярности и частым использованием его в области разработки нейронных сетей. К тому же, для данного языка программирования существует множество библиотек для реализации и обучения нейронных сетей. Одна из них - Tensorflow, которая и была использована в данной работе [27].

РАЗДЕЛ 3. РАЗРАБОТКА МЕТОДА ИНТЕЛЛЕКТУАЛЬНОГО ОБНАРУЖЕНИЯ КЛОНОВ

В данном разделе приводится подробное описание предлагаемого интеллектуального метода обнаружения клонов. А именно, приводится общая структура процесса обнаружения клонов, описываются основные этапы предложенного метода, раскрываются задачи, решаемые на данных этапах, приводятся необходимые алгоритмы.

3.1. Общая схема алгоритма

Предлагаемый метод состоит из следующих основных этапов:

1. предобработка
2. преобразование
3. работа НС
    - обучение НС
    - обнаружение клонов
4. постобработка

В рамках данного метода, на первом этапе, исходный код программы представляется в виде синтаксического дерева. Далее из него извлекаются поддеревья, связанные с функциями или методами классов, которые затем преобразуются в последовательности токенов. Перед последующим преобразованием осуществляется фильтрация и нормализация поддеревьев. Иными словами, удаляются не интересующие нас элементы.

На следующем этапе производится создание векторного представления токенов. С его помощью производится последующее обучение нейронной сети и анализ исходного кода на наличие программных клонов.

Далее следует процесс обучения нейронной сети. В том случае, когда сеть уже обучена, на этом этапе будет производиться поиск клонов с помощью этой сети.

На последнем этапе из всех обнаруженных клонов формируются клоновые классы, которые, в последствии и будут представлены пользователю. Общая структура процесса обнаружения клонов представлена на рисунке 3.1.

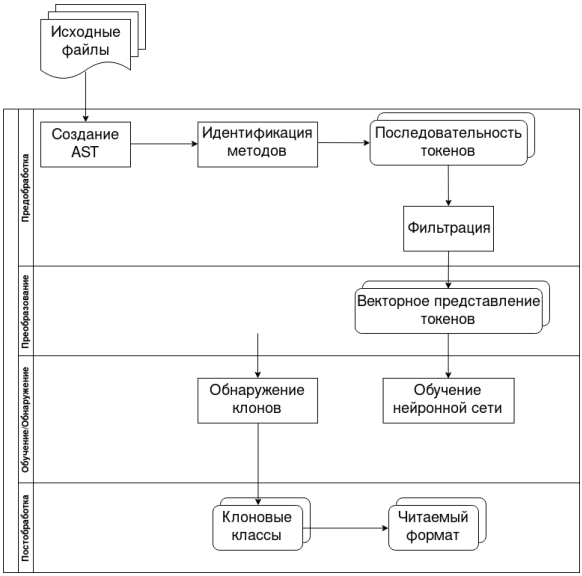

Рис. 3.1. Общая структура процесса обнаружения клонов

3.2. Предобработка

На данном этапе, с помощью анализатора, основанного на части среды разработки (IDE Intellij IDEA community), производится представление исходного кода программы в виде AST. После чего, из полученного дерева выделяются только те поддеревья, которые относятся к методам или функциям. Такой поиск осуществляется посредством обхода в глубину всех вершин AST. В случае обнаружения вершины искомого типа, дальнейший спуск в данную ветку не осуществляется.

Следующим этапом полученные поддеревья преобразуются в

последовательности токенов путем того же обхода в глубину всех их вершин. После чего производится фильтрация этих последовательностей. Производимая фильтрация позволяет исключить влияние незначительных отличий фрагментов исходного кода друг от друга. В рамках данного подхода фильтрации подвергаются токены следующих типов:

- комментарии различного типа
- элементы форматирования
- списки параметров

Обычно, для обнаружения клонов выше клонов I типа, необходимо производить анонимизацию токенов. Однако, в нашем методе, нейронная сеть будет сравнивать типы токенов. Из этого можно сделать вывод о ненадобности анонимизации. Таким образом, процесс фильтрации - последний процесс на этапе предобработки.

3.3. Преобразование

На данном этапе производится преобразование полученных последовательностей токенов в их векторное представление. Данное преобразование производится с помощью модели Миколова - Word2Vec [8]. Данная модель включает в себя набор алгоритмов расчета векторных представлений слов, предполагая семантическую близость слов используемых в похожих контекстах.

Рассматриваемая модель в своей работе использует нейронную сеть прямого распространения. В Word2Vec существуют два алгоритма обучения: CBOW (Continuous Bag of Words) и Skip-gram (рис. 3.2).

CBOW и Skip-gram — это нейросетевые архитектуры, которые описывают, как именно нейросеть «учится» на данных и «запоминает» представления слов. Принципы у обоих архитектур разные. Принцип работы CBOW — предсказывание слова при данном контексте, а Skip-gram наоборот — предсказывается контекст при данном слове.

В данном подходе используется алгоритм Skip-gram. Как уже говорилось ранее, цель обучения Skip-gram модели - найти такие

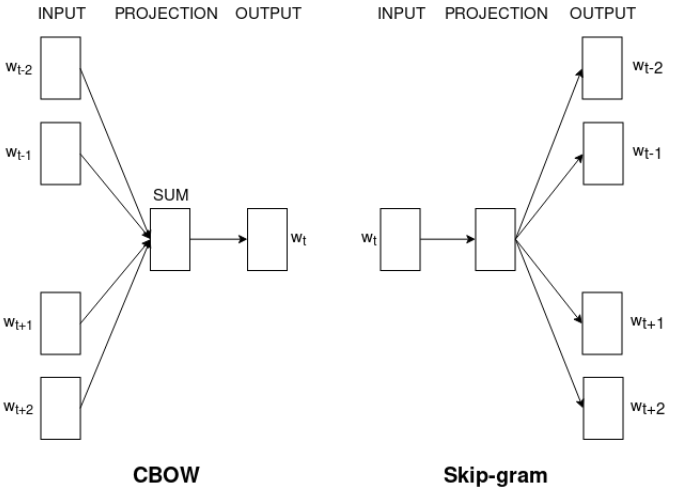

Рис. 3.2. Подходы к обучению Word2Vec

представления слов, которые будут полезны для предсказания контекста в предложении или документе. Формально, при заданной последовательности обучающих слов $w_1, w_2, w_3, ..., w_T$ целью обучаемой модели является максимизация средней логарифмической вероятности (3.1)

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_t) \tag{3.1}$$

, где c - размер обучающего контекста (который может быть функцией центрального слова w_t) [8].

Таким образом, после выполнения данного этапа, были получены векторные представления для каждого токена.

3.4. Обучение и использование нейронной сети

3.4.1. Выборка данных

В рамка предлагаемого метода обнаружение программных клонов осуществляется при помощи нейронных сетей. Для корректного использования сетей их необходимо обучить. В данной секции будет рассматриваться этап обучения нейронной сети.

Одним из самых важных элементов в обучении нейронных сетей является набор обучающих данных. В открытом доступе, для данной задачи, подходящих наборов данных практически нет. Использование результатов других утилит, в качестве набора данных, не позволит объективно оценить предлагаемый подход, так как такое решение приведет к копированию функциональности таких инструментов. В связи с чем было принято решение разработки простого «мутатора» для генерации обучающего набора данных.

Основная задача разрабатываемого «мутатора» заключается в генерации различных программных клонов I-III типа. При этом, сгенерированные клоны не обязаны быть корректными с точки зрения компилятора.

Общая структура работы «мутатора» приведена на рисунке 3.3. Разработанный «мутатор» позволяет сгенерировать обучающую выборку данных. Однако, качество такой выборки не самое высокое и использовать ее можно как второстепенную.

В качестве основной обучающей и тестовой выборки был выбран набор данных BigCloneBench [28]. Данная выборка представляет из себя коллекцию, состоящую из 8 млн проверенных программных клонов и 25 тыс. Java систем с открытым исходным кодом. BigCloneBench содержит как внутрипроектные, так и межпроектные программные клоны четырех типов [28].

Главным преимуществом данной выборки является ее независимое создание от инструментов поиска клонов. Создание выборки таким образом позволяет избежать копирования функциональности существующих инструментов.

Как было упомянуто ранее, в предлагаемом методе будут использоваться рекуррентные нейронные сети. Однако, они обладают недостатком, который заключается в потере информации со временем. Такой недостаток может быть легко устраним при использова-

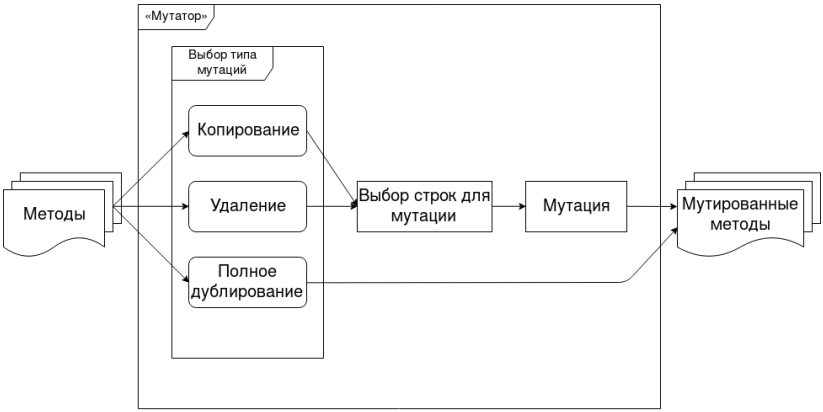

Рис. 3.3. Общая структура «мутатора»

нии сетей с долгой краткосрочной памятью (LSTM). В LSTM-сетях внутренние нейроны «оборудованы» сложной системой так называемых ворот (gates), а также концепцией клеточного состояния (cell state), которая и представляет собой некий вид долгосрочной памяти. Ворота же определяют, какая информация попадет в клеточное состояние, какая сотрется из него, и какая повлияет на результат, который выдаст РНС на данном шаге [19].

3.4.2. Сиамская нейронная сеть

В качестве архитектуры основной нейронной сети была выбрана, так называемая, Сиамская архитектура РНС. Такая архитектура применяется для оценки семантической схожести предложений. Используемая архитектура состоит из двух LSTM сетей со связанными весами, каждая из которых обрабатывает одно из предложений в заданной паре (рис. 3.4). Данная модель использует LSTM для чтения векторных представлений слов каждого входного предложения и использует свое окончательное скрытое состояние в качестве векторного представления предложений. Впоследствии, сходство между этими представлениями используется как предиктор семантического подобия [19].

Обучающая выборка для Сиамской сети состоит из триплетов

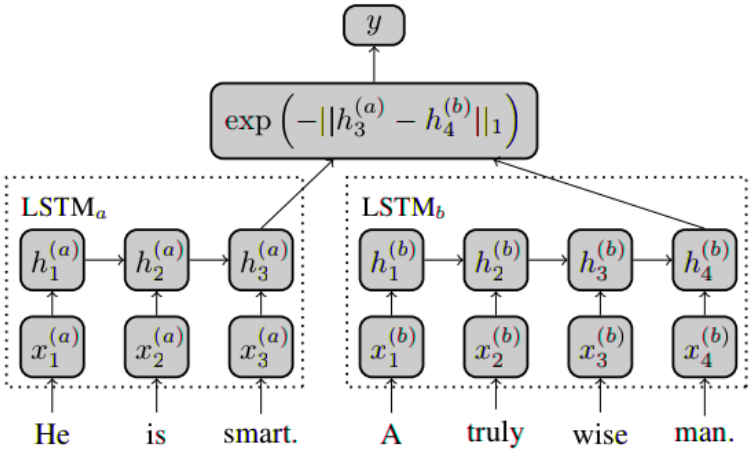

Рис. 3.4. Сиамская нейронная сеть

x_1, x_2, y, где x_1 и x_2 представляют из себя сравниваемые последовательности, а $y \in \{0, 1\}$ определяет сходство x_1 и x_2 ($y = 1$) или их различие ($y = 0$). Целью обучения является уменьшение дистанции между схожими парами и ее увеличение между различными.

Обучение сети производится с помощью использования контрастного коэффициента потерь и квадратичной регуляции $l_2 - norm$, что приводит к следующей целевой функции обучения:

$$L(w) = \frac{\lambda}{2}\|w\|_2^2 + \frac{1}{2N}\sum_{(i,j)\in D} y_{ij}d_{ij}^2 + (1 - y_{ij})max(1 - d_{ij}^2, 0) \quad (3.2)$$

, где w - веса НС, D - набор обучающих пар, d_{ij}^2 - квадратичное l_2 расстояние между i и j последовательностями (рассчитанное между двумя последними слоями Сиамской НС), а $y_{ij} \in 0, 1$ - как и было рассмотрено ранее - значение отвечающее за сходство/различие последовательностей [19].

3.4.3. Sequence-to-Sequence

В предлагаемом подходе программные клоны будут определяться на уровне методов. Так как методы могут быть разных размеров, то и их векторные представления могут быть также разных размеров. Существует два способа приведения представлений методов к единому размеру:

- Выбор максимального размера и дополнения нулями до него
- Использование модели sequence-to-sequence (seq2seq)

В рассматриваемом методе была использована модель seq2seq. Данная модель пользуется большим успехом в различных задачах, например: распознавание речи, машинный перевод, обобщение текстов и т.п.

Суть такой модели заключается в использовании LSTM для считывания входной последовательности небольшими фрагментами до получения большого векторного представления с фиксированной размерностью. Затем используется другая LSTM для извлечения выходной последовательности из этого вектора (рис. 3.5) [24]. В качестве необходимого представления с фиксированным размером было использовано значение скрытой ячейки на последнем этапе развертывания кодировщика (thought vector на рис. 3.5).

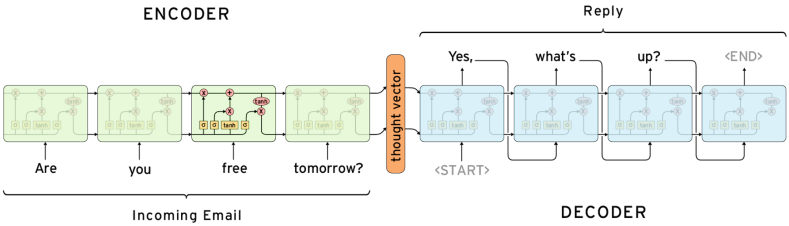

Рис. 3.5. Sequence-to-sequence

Несмотря на тот факт, что в предлагаемом методе используется только часть модели seq2seq, необходимо проводить ее полное обучение. В качестве входных данных кодировщику и декодировщику подаются одинаковые вектора. Таким образом, seq2seq пытается

привести входные данные кодировщика к фиксированному вектору, а затем - этот вектор к данным декодировщика. Иными словами, суть обучения заключается в увеличении логарифмической вероятности правильного перевода T с учетом исходного предложения S:

$$\frac{1}{|D|} \sum_{(T,S) \in D} log_p(T|S) \tag{3.3}$$

, где D - обучающий набор данных [24].

3.5. Постобработка

Финальный этап предлагаемого метода - постобработка. Суть данного этапа заключается в приведении результатов в компактном и воспринимаемом человеком формате. Без данного этапа результат работы сети выглядит как набор векторных представлений с бинарным ответом о схожести двух методов. Таким образом, данный этап состоит из двух частей:

- Создание клоновых классов (Компактность)
- Вывод пути метода и его расположение в файле (номера строк начала метода и его конца) (Воспринимаемость)

3.6. Итоги раздела

В данном разделе был представлен метод интеллектуального обнаружения клонов, основанный на использовании рекуррентных нейронных сетей. Рассмотрена общая структура процесса обнаружения, а также основные этапы метода.

РАЗДЕЛ 4. РАЗРАБОТКА ПРОТОТИПА ИНСТРУМЕНТА ОБНАРУЖЕНИЯ КЛОНОВ

В данном разделе приводится подробное описание реализации прототипа инструмента обнаружения клонов. А именно, рассматривается общая структура прототипа, реализация «мутатора» и нейронных сетей.

4.1. Общая структура прототипа

Разрабатываемый прототип инструмента обнаружения клонов состоит и двух модулей:

- Инструмент построения AST
- Инструмент поиска клонов с помощью НС

Как было описано ранее, языком реализации инструмента для построения AST и представления его в виде токенов был выбран Java.

Реализация инструмента поиска клонов была выполнена с помощью популярной программной платформы от Google - Tensorflow. Языком реализации является Python.

4.2. Реализация инструмента построения AST

Данный инструмент состоит из нескольких основных программных пакетов:

- trees - реализует построение AST из заданного исходного кода;
- preproc - обеспечивает преобразование AST в последовательность токенов. Также отвечает за векторное представление токенов;

4.2.1. Состав пакета trees

Данный пакет содержит классы, обеспечивающие преобразование исходного кода в абстрактное дерево. На самом деле, данный пакет создает PSI (Program Structure Interface), что можно рассматривать как расширение AST. В данный пакет входят два класса:

- PsiGen - класс, отвечающий за анализ исходного кода программы и создания из него PSI. Помимо создания PSI, рассматриваемый класс преобразует PSI в последовательность токенов.
- ASTEntry - представляет из себя реализацию токенов, которые используются в дальнейшем. Также данный класс содержит в себе различные методы взаимодействия с токенами.

Далее будет приведено описание класса ASTEntry.

ASTEntry

Как было описано выше, данный класс представляет из себя реализацию токена, входящего в PSI. Данный класс включает в себя следующие элементы класса:

- nodeName - элемент класса, содержащий в себе имя узла;
- sourceStart - элемент класса, определяющий строку, где начинается данный токен;
- sourceEnd - элемент класса, определяющий строку, где заканчивается данный токен;
- parent - элемент класса, указывающий на родителя токена;
- children - список всех дочерних токенов для рассматриваемого;
- filePath - содержит в себе путь файла из которого был извлечен данный токен;

и следующие методы:

- removeSpaces - метод, производящий удаление неинтересующих нас токенов из списка дочерних;

- getAllTokensList - метод, который обходит список всех дочерних токенов и возвращает только их имена в виде списка;
- filePath - содержит в себе путь файла из которого был извлечен данный токен;
- mutate - метод, производящий мутации различного типа;

4.2.2. Состав пакета preproc

В пакет preproc входят классы отвечающие за создание векторных представлений токенов и мутации этих токенов:

- TreeMutator - данный класс организует работу с исходным кодом. А именно - вызывает создание PSI, выполняет преобразование PSI в последовательность токенов, производит ее фильтрацию и, при необходимости - мутацию;
- Embedding - реализует создание векторных представлений токенов и записывает полученные результаты в файлы. Таким образом, данный класс организует создание выборки данных для последующего обучения сетей.

Далее рассматривается класс Embedding, т.к. он представляет наибольший интерес.

Embedding

Рассматриваемый класс содержит в себе реализацию word2vec с помощью которой производится создание векторных представлений. В данный класс входят следующие методы:

- train - метод, который осуществляет обучение модели word2vec на заранее выбранном проекте. В нашем случае, в качестве данного проекта был выбран openjdk. В данном методе, также, производится сериализация обученной модели с целью ее повторного использования.
- createEmbedding - метод, отвечающий за создание векторного представления заданных токенов и сохранение их в файл.

4.3. Реализация инструмента обнаружения программных клонов

Данный инструмент был написан с использованием платформы Tensorflow. Он состоит из нескольких файлов, в которых описаны различные классы и методы:

- clonesRecognition - основной файл, в котором организуется работа с инструментом. Вызываются различные методы инициализации, обучения, вычисления и т.д.;
- model - файл, в котором описываются модели нейронных сетей, методы их обучения и использования;
- helpers - вспомогательный файл, в котором описываются различные методы преобразования информации, сохранения, чтения и т.п.;
- clones - файл, описывающий клоновые классы.

4.3.1. Состав файла model

Как уже было описано, в данном файле содержатся два класса, описывающих разные НС:

- Seq2seq - класс, реализующий seq2seq алгоритм;
- SiameseNetwork - класс, реализующий Сиамскую архитектуру НС.

Seq2seq

В рассматриваемом классе присутствуют следующие элементы:

- scope - содержит в себе название предела, в котором будут существовать все переменные (необходим для корректной сериализации и десереализации обученной модели);
- sess - сессия внутри которой будут проходить обучение и вычисления;
- encoder_cell - ячейка кодировщика (отвечает за попытки привести информацию к вектору фиксированного размера);

- decoder_cell - ячейка декодировщика;
- vocab_size - размер словаря (в нашем случае - длинна векторного представления токена);
- input_embedding_size - размер входной последовательности.

Помимо указанных элементов, в данном классе присутствуют методы обучения (train) и вычисления (get_encoder_status).

SiameseNetwork

Так как в данном классе присутствуют элементы схожие с элементами предыдущего класса, в этом подразделе они рассмотрены не будут. Рассмотрим отличающиеся элементы:

- input_x1 - переменная первой последовательности для сравнения;
- input_x2 - переменная второй последовательности для сравнения;
- input_y - переменная указывающая на сходство/различие первых двух (0 - схожи, 1 - различны);
- sequence_length - переменная, указывающая длину последовательности (результат работы seq2seq)
- layers - количество слоев для обучения

4.3.2. Состав файла helpers

В файле helpers содержатся только вспомогательные методы. Среди них:

- batch - создает серии значений из заданной последовательности;
- siam_batches - представляет заданные параметры для обучения Сиамской сети в виде тройки значений $\{x_1, x_2, y\}$;
- random_sequences - производит генерацию случайных последовательностей для обучения seq2seq модели;
- save_model - производит сохранение модели в указанный файл;

- load_model - производит загрузку модели из указанного файла.

4.3.3. Состав файла clonesRecognition

Как упоминалось ранее, данный файл является входной точкой рассматриваемой утилиты. В нем содержатся следующие методы:

- main - входная точка утилиты. В ней производится инициализация основных значений, выбор типа работы (обучение, расчет или полный);
- seq2seq_train - обучение seq2seq модели;
- siam_train - обучение Сиамской НС;
- eval - производит полный расчет;

4.4. Итоги раздела

В данном разделе была рассмотрена реализация прототипа инструмента обнаружения клонов, основанного на предложенном интеллектуальном методе анализа. Приведена общая структура прототипа и описаны его основные составляющие: инструмент предобработки и инструмент обнаружения клонов.

РАЗДЕЛ 5. ТЕСТИРОВАНИЕ И АНАЛИЗ РЕЗУЛЬТАТОВ

В данном разделе проводится исследование характеристик прототипа на примере его использования для обнаружения клонов. Тестирование проводилось на нескольких различных выборках из BigCloneBench.

5.1. Тестирование прототипа обнаружения клонов

5.1.1. Описание тестовых данных

Для тестирования разработанного прототипа из выборки BigCloneBench были выделены партии с разным количеством методов:

- приблизительно 4000 методов
-
- приблизительно 11000 методов

Выбранные фрагменты набора можно считать программами среднего размера.

5.1.2. Описание тестовой платформы и конфигурации прототипа

Все эксперименты в рамках данного тестирования были проведены на двух различных машинах со следующими конфигурациями:

1. ПК
    - ОС ArchLinux;
    - CPU Intel(R) Core(TM) i7-7700K CPU 4.20ГГц;
    - 64 Гб RAM;
    - NVidia Quadro P4000 8Гб;

2. DGX-1

   - ОС Ubuntu 16.04 LTS
   - Intel(R) Xeon(R) CPU E5-2698 v4 2.20 ГГц x2;
   - 512 ГГб RAM
   - NVidia Tesla V100 16Гб x8.

5.1.3. Исследование показателей прототипа

Главной задачей данной работы является разработка метода интеллектуального обнаружения клонов. Основными требованиями, предъявляемыми к такому методу, являются увеличение точности поиска и определение клонов первых трех типов. Данные характеристики исследуются на примере использования разработанного прототипа для обнаружения клонов. Результат анализа прототипа представлен в таблице 5.1, где:

- $KLOC$ - количество строк исходного кода (в тысячах);
- $N_{methods}$ - количество исследуемых методов;
- N_{clones} - количество найденных клонов;
- Rec - полнота результата;
- $Prec$ - точность результата;
- t_w - время расчета (без учета преобразований и обучения);
- t_f - полное время работы (без учета обучения сетей);

Так как тестирование производилось на двух абсолютно разных машинах, в таблице указано худшее время работы.

Таблица 5.1

Результаты тестирования

$N_{methods}$	KLOC	N_{clones}	Rec	Prec	t_w	t_f
4092	228	1697	0.94	0.97	02:13 мин	04:42 мин
8270	768	5453	0.88	0.94	04:25 мин	06:55 мин
10892	3192	32041	0.88	0.93	07:22 мин	09:44 мин
19711	2830	24797	0.90	0.76	10:51 мин	14:43 мин

Все анализируемые метрики весьма тривиальные, за исключением *Recall* и *Precision*. Для их пояснения необходимо ввести некоторые понятия, а именно - *True positives* (TP), *True negatives* (TN), *False positives* (FP), *False negatives* (FN), где:

- *True positives* - истинно-положительный результат;
- *True negatives* - истинно-отрицательный результат;
- *False positives* - ложно-положительные результаты;
- *False negatives* - ложно-отрицательные результаты.

Такие метрики рассчитываются согласно табл. 5.2.

Таблица 5.2

Вспомогательные метрики

	Клон	Не клон
Клон (CD)	TP	FP
Не клон (CD)	FN	TN

Итак, после введения дополнительных метрик, рассчет *Precision* и *Recall* будет намного нагляднее. Для расчета основных метрик оценки работы нейронных сетей используются следующие формулы:

$$Recall = \frac{True\ positives}{True\ positives + False\ negatives} \tag{5.1}$$

$$Precision = \frac{True\ positives}{True\ positives + False\ positives} \tag{5.2}$$

ЗАКЛЮЧЕНИЕ

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Akhin Marat, Itsykson Vladimir. Clone detection: Why, what and how? — 2010. — 10.
2. Backpropagation Applied to Handwritten Zip Code Recognition / Y. LeCun, B. Boser, J. S. Denker et al. // Neural Comput. — 1989. — Dec. — Vol. 1, no. 4. — P. 541–551.
3. Basit H. A., Rajapakse D. C., Jarzabek S. Beyond templates: a study of clones in the STL and some general implications // Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005. — 2005. — May. — P. 451–459.
4. Burd Elizabeth, Munro Malcolm. Investigating the Maintenance Implications of the Replication of Code // Proceedings of the International Conference on Software Maintenance. — ICSM '97. — Washington, DC, USA : IEEE Computer Society, 1997. — P. 322–.
5. Clone detection in source code by frequent itemset techniques / Vera Wahler, Dietmar Seipel, Jürgen Wolff V. Gudenberg, Gregor Fischer // In SCAM. — 2004. — P. 128–135.
6. Cordy James R., Dean Thomas R., Synytskyy Nikita. Practical Language-Independent Detection of Near-Miss Clones // In proceedings of the 14th IBM centre for advanced studies conference (CASCON'04). — IBM Press, 2004. — P. 1–12.
7. The Development of a Software Clone Detector / Neil Davey, Paul Barson, Simon Field et al. // International Journal of Applied Software Technology. — 1995.
8. Distributed Representations of Words and Phrases and their Compositionality / Tomas Mikolov, Ilya Sutskever, Kai Chen et al. // Advances in Neural Information Processing Systems 26. — Curran Associates, Inc., 2013. — P. 3111–3119.
9. Lightweight detection of duplicated code — a language-independent approach : Rep. / Institute of Applied Mathematics and Computer Science. University of Berne ; Executor: Stéphane Ducasse, Oscar Nierstrasz, Matthias Rieger : 2004.
10. An ethnographic study of copy and paste programming practices in OOPL / Miryung Kim, L. Bergman, T. Lau, D. Notkin // Em-

pirical Software Engineering, 2004. ISESE '04. Proceedings. 2004 International Symposium on. — 2004. — Aug. — P. 83–92.
11. Extending software quality assessment techniques to Java systems / J. F. Patenaude, E. Merlo, M. Dagenais, B. Lague // Proceedings Seventh International Workshop on Program Comprehension. — 1999. — P. 49–56.
12. Hopfield John J. Neural networks and physical systems with emergent collective computational abilities // Proceedings of the National Academy of Sciences. — 1982. — Vol. 79, no. 8. — P. 2554–2558.
13. Howard Johnson J. Identifying redundancy in source code using fingerprints. — 1993. — 01. — P. 171–183.
14. Kapser C., Godfrey M. W. "Cloning Considered Harmful" Considered Harmful // 2006 13th Working Conference on Reverse Engineering. — 2006. — Oct. — P. 19–28.
15. Koschke Rainer, Falke Raimar, Frenzel Pierre. Clone Detection Using Abstract Syntax Suffix Trees // Proceedings of the 13th Working Conference on Reverse Engineering. — WCRE '06. — 2006. — P. 253–262.
16. Krinke J. Identifying similar code with program dependence graphs // Proceedings Eighth Working Conference on Reverse Engineering. — 2001. — P. 301–309.
17. L. Elman Jeffrey. Finding Structure in Time // Cognitive Science. — 1990. — Vol. 14, no. 2. — P. 179–211.
18. Mallaiah Sudhamani, Rangarajan Lalitha. Code Similarity Detection by Computing Block Level Feature Metrics // nternational Journal of Computer Application (2250-1797). — 2018. — Mar-Apr. — Vol. 2.
19. Mueller Jonas, Thyagarajan Aditya. Siamese Recurrent Architectures for Learning Sentence Similarity // Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. — AAAI'16. — Phoenix, Arizona : AAAI Press, 2016. — P. 2786–2792.
20. Rieger Matthias. Effective clone detection without language barriers /. — 2005. — 01.
21. Rosenblatt Frank. The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain // Psychological Review. — 1958. — P. 65–386.
22. Roy Chanchal, R. Cordy James. A Survey on Software Clone De-

tection Research. — 2007. — 01.
23. Roy Chanchal, R. Cordy James. An Empirical Study of Function Clones in Open Source Software. — 2008. — 10. — P. 81–90.
24. Sutskever Ilya, Vinyals Oriol, Le Quoc V. Sequence to Sequence Learning with Neural Networks // Advances in Neural Information Processing Systems 27 / Ed. by Z. Ghahramani, M. Welling, C. Cortes et al. — Curran Associates, Inc., 2014. — P. 3104–3112.
25. Tairas Robert, Gray Jeff. Phoenix-based clone detection using suffix trees // In ACM-SE. — 2006. — P. 679–684.
26. TIOBE index for Java, TIOBE. The software quality company. — Access mode: `https://tiobe.com/tiobe-index/java/` (online; accessed: 19.05.2018).
27. TIOBE index for Java, An open source machine learning framework for everyone. — Access mode: `https://www.tensorflow.org/` (online; accessed: 19.08.2017).
28. Towards a Big Data Curated Benchmark of Inter-project Code Clones / J. Svajlenko, J. F. Islam, I. Keivanloo et al. // 2014 IEEE International Conference on Software Maintenance and Evolution. — 2014. — Sept. — P. 476–480.
29. Walenstein Andrew, Lakhotia Arun. The Software Similarity Problem in Malware Analysis. — 2007. — 01.
30. Yang Wuu. Identifying Syntactic Differences Between Two Programs // Software - Practice and Experience. — 1991. — Vol. 21. — P. 739–755.

ПРИЛОЖЕНИЕ 1
ЛИСТИНГИ

Листинг 1.1. Код на Java

```
1  private static boolean changesLine(final GenericTreeNode
       patternTree, int reportLine) {
2    LineNumberFetcher fetcher = new LineNumberFetcher();
3    try {
4      fetcher.visit(patternTree);
5      return fetcher.lines.contains(reportLine);
6    } catch (Exception e) {
7      return false;
8    }
9  }
10
11 private static class LineNumberFetcher implements TreeVisitor {
12   Set<Integer> lines = new TreeSet<>();
13
14   @Override
15   public void visit(GenericTreeNode genericTreeNode) throws
         Exception {
16     for (GenericTreeNode matchingNode : genericTreeNode.
           getMatchNodes()) {
17       lines.add(matchingNode.getLine());
18     }
19     for (GenericTreeNode child : genericTreeNode.getChildren()) {
20       child.acceptVisitor(this);
21     }
22   }
23 }
```