МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«ТЮМЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ НАУК
Кафедра программного обеспечения

Заведующий кафедрой
к.т.н., доцент
М.С. Воробьева

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**
магистра

# РАЗРАБОТКА СИСТЕМЫ ПРОВЕРКИ СХОЖЕСТИ ИСХОДНЫХ ТЕКСТОВ ПРОГРАММ С ТОЧКИ ЗРЕНИЯ СТРУКТУРЫ

02.04.03 Математическое обеспечение и администрирование информационных систем
Магистерская программа «Разработка, администрирование и защита вычислительных систем»

Выполнил работу
студент 2 курса
очной формы обучения — Долгушин Андрей Сергеевич

Научный руководитель
доцент — Павлова Елена Александровна

Рецензент
веб-разработчик
Общество с ограниченной ответственностью
«Геонавигационные технологии» — Пиджаков Святослав Игоревич

Тюмень
2020

**ОГЛАВЛЕНИЕ**

Введение ..... 3

Глава 1. Анализ алгоритмов поиска плагиата ..... 5

§1.1. Проблема плагиата ..... 5

§1.2. Описание методов выявления плагиата ..... 9

§1.3. Анализ существующих решений ..... 16

§1.4. Требования к решению ..... 18

Глава 2. Построение модели ..... 21

§2.1. Абстрактное синтаксическое дерево ..... 21

§2.2. Идея решения ..... 23

§2.3. Векторное представление ..... 25

§2.4. Долговременная краткосрочная память ..... 30

§2.5. Процесс классификации ..... 34

Глава 3. Численные эксперименты ..... 37

§3.1. Программная реализация ..... 37

§3.2. Исходные данные ..... 40

§3.3. Оценка модели ..... 41

§3.4. Сравнение с аналогами ..... 45

Заключение ..... 49

Список литературы ..... 50

Приложение ..... 52

# ВВЕДЕНИЕ

На протяжении веков была весьма актуальна проблема заимствования различных авторских произведений искусства, достижений науки, технических решений или изобретений.

Студенческий плагиат является одной из серьезных проблем образовательных учреждений [22]. Заимствованные фрагменты или даже полностью работы могут встречаться как в лабораторных работах, курсовых проектах, так и дипломных работах.

Однако, как быть, если результат работы представлен в виде программ на языках программирования высокого уровня? Ведь, исходные коды обладают собственной спецификой, так как изменение имен переменных или внесение дополнительных комментариев в код программы не влияют на выполнение программы, но при этом изменяют его текстовую составляющую. Следовательно, методики выявления заимствований в текстах на естественных языках неприменимы напрямую к текстам исходных кодов программ и необходимо расценивать исходный код с точки зрения синтаксического анализа.

Применение разрабатываемой системы (проверки идентичности исходных текстов программ) довольно многогранно. Одним из способов применения такой системы является обнаружение плагиата в исходных кодах программ. Данная задача является актуальной не только в коммерческой сфере, где идет серьезная борьба за интеллектуальную собственность, но и в сфере образования, где плагиат в студенческих работах является одной из основных проблем образовательных учреждений.

Целью данной работы является разработка системы проверки идентичности исходных текстов программ, которая могла бы выявлять случаи "списывания" программ, автоматически сравнивать исходные тексты и позволяла находить похожие или идентичные фрагменты в исходных кодах

программ, оценивать степень обнаруженного сходства и наглядно представлять результат сравнения.

Для достижения указанной цели был поставлен ряд основных задач:

- Изучить существующие методики выявления заимствований, аналогичные решения.
- Построение синтаксического дерева для предложенного решения.
- Создание словаря, построение контекстных пар.
- Обучение модели, способной определить являются ли два текста программы схожими.
- Провести оценку модели и сравнительный анализ со существующими решениями.

# ГЛАВА 1. АНАЛИЗ АЛГОРИТМОВ ПОИСКА ПЛАГИАТА

Плагиат в исходном коде определяют, как "воспроизведение (копирование) исходного кода без внесения каких-либо изменений или с внесением, но незначительных изменений" [22].

Проблема плагиата и других способов обмана стоит очень остро в последнее время. На сегодняшний день при написании курсовых работ или дипломов, во многих учебных заведениях страны, обязательна проверка на антиплагиат.

Проблема плагиата не ограничивается учебными заведениями и научными кругами. Он может быть найден в различных областях: в академической среде, публицистике, патентах, в коммерческой сфере.

С другой стороны, есть исследования [3], показывающие снижение плагиата в академической сфере в последние годы. Именно они показывают важность систем антиплагиата. Исследования сообщают о снижении уровня плагиата после использования такой системы в процессе защиты работ.

## §1.1. Проблема плагиата

В общем представлении плагиат в исходном коде мало чем отличается от плагиата текста работы. Причины его возникновения одинаковы в обоих случаях. Сегодня мы живем в то время, когда в интернете можно получить свободный доступ к любой информации. Большое количество различных работ, публикаций, книг, исходных текстов программных продуктов находятся в свободном доступе на различных ресурсах. Это простота побуждает студентов копировать материал в своих работах.

Проблема выявления плагиата в исходном коде заключается, в частности, в том, что, в отличие от текстовых документов, в настоящее время нет специализированного сервиса, который позволил бы выявлять плагиат в глобальном масштабе.

В отличии от текстовой работы, когда отличия очевидны, определение плагиата в случае исходного кода является затруднительным процессом. Конечно, бывают случаи, когда плагиат очевиден. Одним из таких случаев является полное соответствие, когда два студента сдают одну и ту же работу.

Творчество студентов в области маскирования плагиата довольно велико. Часть таких изменений можно объединить в группу по лексическому признаку. Они представляют собой поверхностные изменения исходного кода программы. Для внесения таких изменений не требуются глубокие знания о языке программирования, его синтаксисе и структурах данных, а достаточно лишь понимание основ языка.

Среди них можно выделить:

1) Изменение разметки (форматирования) исходного текста программы: вставка или удаление пустых строк, разбиение одной строки на две и более строк, путём переноса части строки на следующую строку, вставка или удаление лишних пробельных символов или символов табуляции.

2) Изменение комментариев, то есть вставку новых или удаление уже существующих.

3) Переименование идентификаторов: наиболее часто применяемый способ, при использовании которого изменяются имена переменных, констант, классов, методов классов, полей классов.

4) Объединение или разделение строк объявления переменных: если несколько переменных представляют один и тот же тип данных, то они могут быть объявлены в одной строке кода. Такое изменение предполагает изменение расположения объявлений переменных.

Еще одну группу изменений можно выделить на основе структурного признака. Структурные изменения представляют собой различные изменения, затрагивающие процесс выполнения программы и используемые в ней

структуры и типы данных. Для проведения некоторых типов структурных атак требуется более высокий уровень знаний о языке программирования, на котором написан исходный код программы, а также хорошее понимание структуры изменяемой программы.

Среди такого рода изменений можно выделить следующие типы:

1) Изменение порядка следования переменных в выражениях: изменение должно быть таким, чтобы это не отразилось на конечном результате операции.

2) Изменение порядка следования выражений в блоке кода: изменение должно быть таким, чтобы это не повлияло на результат выполнения программы.

3) Изменение порядка следования блоков кода: изменение порядка следования блоков исходного кода, таким образом, чтобы это не повлияло на результат выполнения программы. Под блоком кода понимается последовательность строк кода, заключённая между специальными символами разделителями (например, для C# – это фигурные скобки).

4) Вставка избыточных выражений, операторов или переменных: данный вид изменений предусматривает произведение вставок избыточных участков кода, которые не влияют на ход выполнения программы и результаты вывода. Это может быть вставка дополнительных локальных переменных, вставка строк кода, которые заведомо никогда не будут выполняться.

5) Встраивание метода: вызовы методов заменяются кодом, содержащимся в теле этих методов.

6) Преобразование блока кода в метод: преобразование произвольного блока кода в новый метод класса является простой и безопасной модификацией кода, которая позволяет визуально изменить исходный код, не нарушая работоспособность программы.

7) Изменение используемых типов: изменение используемого типа (переменных, параметров метода) заключается в замене одного типа данных на другой (например, int на long). Однако при недостатке знаний об особенностях, характерных различным типам данных, такие замены могут привести к потере точности вычислений и искажению конечных выводов программы.

Существует несколько причин плагиата. Наиболее распространенные [21]:

- Нехватка времени
- Незнание материала
- Заимствованная работа лучше, чем собственная
- Лень

Хотя вышеупомянутые исследования были посвящены всем видам работ, а не только к написанию программ, эти причины также описывают ситуацию в данной области с точки зрения студента.

Если есть необходимость решить проблему плагиата, надо сосредоточиться на том, как к данному вопросу подходят студенты. Некоторые из причин (такие как нехватка времени или отсутствия знаний) могут быть связаны с неправильным способом обучения. Введение систем антиплагиата в этом случае не сможет улучшить ситуацию. И наоборот, в тех случаях, когда плагиат вызван ленью или убеждением, что так делают все, система антиплагиата является отличным инструментом для студентов, чтобы заставить их работать независимо друг от друга.

## §1.2. Описание методов выявления плагиата

За все время исследования данной проблемы возникло не малое количество подходов. Среди наиболее распространённых можно выделить следующие:

1) Текстовые (String-based)

Представляют программу в виде строки над алфавитом, символы которого представляют оператор или группу операторов языка программирования. При этом происходит поиск точных строковых совпадений. Данный подход является очень чувствительным к любому изменению имен идентификаторов.

Из современных текстовых алгоритмов и методов можно выделить следующие:

- алгоритм на основе выравнивания строк — раздвигает символы строк так, чтобы выявить совпадающие участки.
- алгоритм на основе строкового замощения — эвристический алгоритм, выдающий наибольшее множество непересекающихся совпадающих подстрок.
- метод просеивания — находит общие подстроки определенного размера.
- метод отпечатков — сравнивает выборочные участки программ.

Одним из таких методов является подход, предложенный Baker [1]. В своей работе он преобразует и нормализует исходный код, удаляя комментарии, пробелы. Затем хэширует каждую строку целевых файлов и сравнивает их с помощью суффиксного дерева. Данный подход способен уловить только неизмененные участки кода или с небольшим числом изменений, за исключением систематических замен параметров, таких как идентификаторы и константы.

Большая работа была проделана в области предотвращения плагиата исходного кода Haider и др [6]. Их статья посвящена работе плагиату исходного кода в колледжах и университетах. В своем исследовании они предлагают новый метод, основанный на алгоритме жадного строкового замещения. В заключении они также отмечают, что их метод эффективен в случае изменения имен переменных и функций, устойчив к перемещению фрагментов кода.

2) Представление в виде токенов (Token-based)

Подходы, основанные на токенах, также называются лексическими подходами. При данном подходе, весь исходный код транслируется в последовательность токенов лексическим анализатором, после чего, все токены формируются в виде набора последовательностей. И в конце концов, полученная последовательность используется для идентификации дублирующегося кода.

Одним из самых первых инструментов, реализующих данный метод, является CCFinder, предложенный Kamiya и др. Прежде всего, лексер разделяет каждую строку текста на токены, а затем образует единую последовательность токенов. Затем, используется алгоритм сопоставления суффиксного дерева для поиска схожих подпоследовательностей последовательности токенов.

В ходе дальнейшего развития для преодоления проблемы CCFinder был представлен CP-Miner, который для обнаружения клонов использует частотный поиск подпоследовательностей.

Еще одна работа, которая заслуживает внимания, является работа Hiroaki Murakami [7]. В своей диссертации автор предложил два метода обнаружения клонов, которые улучшают существующие слабые места данной разновидности подхода. Первый метод — это метод, основанный на токенах,

который складывает повторные инструкции для уменьшения «неинтересных» кандидатов, а затем применяет методы обнаружения на основе токенов. Этот инструмент называется FRISC. Второй метод использует метод локального выравнивания последовательностей под названием - алгоритм Смита — Ватермана. Этот алгоритм является более быстрым и более точным, чем предыдущий алгоритм.

3) На базе деревьев (Tree-based)

Для каждого файла с исходным кодом строится абстрактное синтаксическое дерево, затем полученные деревья сравниваются между собой [12, 13].

Zheng в своей работе [17] описывает используемый им способ для создания системы, которая позволяет пользователям искать репозитории с исходным кодом, используя лишь фрагменты кода в качестве запроса. Он использует абстрактные синтаксические деревья для представления файлов исходного кода, что в сочетании с хэшированием узлов и вычислениями подобия, позволяет пользователям искать фрагменты исходного кода, подозрительным на плагиат. В работе выявляется ряд широко используемых методов, позволяющих избежать выявления плагиата, а также оценивается система на предмет её способности выявлять случаи плагиата в случаях применения методов сокрытия. Результаты оценки весьма многообещающи, 95% всей тестовой выборки были успешно идентифицированы.

4) PDG-based (PDG – Program Dependency Graph)

В дальнейшем, методы, основанные на AST, переросли в методы с использованием графа зависимостей программы (PDG) [15]. Подход PDG использует поток данных и поток управления для обнаружения семантических

и синтаксических клонов, строится граф зависимостей, который отражает зависимость между управляющими конструкциями программы, то есть как переходит управление из одной точки программы в другую, и между потоками данных. Затем полученные графы для каждого файла с исходным кодом сравниваются между собой.

Были разработаны различные методы измерения, и известно, что методы, основанные на использовании графа зависимостей программы, хорошо работают против методов маскировки. Но эти методы обычно направлены на решение NP-трудных задач, которые возникают в ходе проблемы масштабируемости. Kim в статье об измерении сходства исходного кода [10] предлагает генетический алгоритм для измерения сходства путем решения задачи проверки изоморфизма подграфов. В статье описывается новая целевая функция для этой задачи, которая отражает характеристику исходных кодов. Для решения этой задачи используется инкрементный генетический алгоритм. Размер искомого графа постепенно увеличивается в ходе эволюционного процесса. Экспериментальные результаты показали, что данная система успешно работает по выявлению плагиата кода и обнаружению вредоносных программ.

Kamalpriya и др. [8] предложили дальнейшее усовершенствование методов обнаружения на основе PDG для идентификации всех возможных (точных и приближенных) клонов с использованием приближенного подграфа (ASM). Они также представили новую меру расстояния на основе ASM для выявления сходства между клонами программного кода.

5) Метрические (Metrics-based)

Данная разновидность подходов заключается в оценке метрик программы, например, количестве используемых переменных, циклов,

условных операторов и т.д. Далее две программы сравниваются по соответствующим метрикам [9].

Patenaude и его коллеги создали метрический подход для клонов кода. В своей работе они использовали следующие метрики: (1) число вызовов изнутри метода, (2) число аргументов, (3) цикломатическая сложность Маккейба, (4) число глобальных переменных и (5) число локальных переменных. Эти метрики были определены для языка программирования Java.

Kanika и др рассматривается метод, основанный на метрике, который может быть применен в качестве предварительного этапа фильтрации для уменьшения сложности, с которой сталкиваются предыдущие методы. В своем методе они используют байтовый код для вычисления метрик программ, написанных на языке Java, вместо того чтобы использовать любое другое представление. Причина использования байтового кода заключается в том, что он не зависит от платформы и представляет собой унифицированную структуру кода. Таким образом, предложенный подход также в некоторой степени позволяет выявлять сходства по семантическим признакам. Этот подход может быть использован независимо и может быть объединен с другими подходами для обнаружения клонов кода.

6) Машинное обучение

Существуют также и менее распространенные методы, к примеру, основанные на алгоритмах машинного обучения [5].

В последние годы проблеме копирования исходного кода уделяли Ganguly и др. В своей статье [4] они предложили подход на основе классификатора, обученного на признаках, извлеченных из пар исходных кодов, помеченных как плагиат. В качестве признаков они выделили лексические, структурные и стилистические особенности документа. В результате их модель показала неплохую точность и высокую отзывчивость.

В своей статье Zhang [16] предлагают новую нейронную сеть на основе AST под названием ASTNN для представления исходного кода. В отличие от существующих моделей, которые работают на целых AST, ASTNN разбивает каждый большой AST на последовательность небольших деревьев и преобразовывают деревья в векторы, захватывая лексические и синтаксические признаки. Основываясь на этих последовательностях векторов, двунаправленная RNN используется для того, чтобы использовать «естественность» операторов и, наконец, получить векторное представление фрагмента кода. Экспериментальные результаты показывают, что модель превосходит современные подходы.

Sheneamer и др. в своей статье [14] представляют механизм машинного обучения для автоматического обнаружения клонов в программном обеспечении, который способен обнаруживать фрагменты с внесенными в них изменениями и самый сложный вид клонов, когда два или более фрагментов реализуют одинаковую логику, но отличаются синтаксически. Ранее использовавшиеся традиционные методы часто являются слабыми в обнаружении семантических клонов. В качестве новых аспектов своего подхода авторы преподносят извлечение особенностей из абстрактных синтаксических деревьев (AST) и графов программных зависимостей (PDG), представление пар фрагментов кода в виде векторов и использование классификационных алгоритмов. Ключевым преимуществом этого подхода, по словам автора, является то, что их подход может найти как синтаксические, так и семантические клоны чрезвычайно хорошо.

7) Гибридные подходы

Существует также множество гибридных методов обнаружения клонов кода. Они представляют собой совокупность нескольких подходов и могут быть классифицированы на основе вышеперечисленных.

Каждый из перечисленных методов имеет свои достоинства и недостатки. Но главной и наиболее важной, и интересной характеристикой является, то какие типы дублирующихся фрагментов он способен обнаружить.

Методы, основанные на сравнении строк, способны обнаруживать дублирующие фрагменты кода с лишь малым числом изменений лексического типа (как правило не восприимчивы к символам пробела, табуляции, комментариев и только). Алгоритмы, основанные на базе структурного сравнения таких как, токенизация и построении абстрактных синтаксических деревьев, работают практически для всех видов лексических изменений. Методы, основанные на построении графов зависимостей, работают как для структурных, так и лексических изменений.

Cordy и Roy [2] представили гибридную технику обнаружения клонов, используемая для синтаксически схожих клонов. Это комбинация двух методик обнаружения клонов - текстовых и AST. В нём предусмотрены три этапа. На первом - парсер извлекает процедуры и делит их на строки. На втором этапе выполняется нормализация и фильтрация с использованием заданных параметров (правил). На третьем этапе, выполняется сопоставления двух фрагментов с использованием алгоритма LCS (наибольшей общей подпоследовательности). Каждый возможный фрагмент кода сравнивается с каждым другим фрагментом. Таким образом, это очень дорогостоящий инструмент.

В данной статье [11] автор предлагает метод LWH (Light Weight Hybrid) для выявления клонов кода. Метод представляет собой комбинацию использования метрик и текстового анализа, для обнаружения семантических и синтаксических клонов процедурного уровня в проектах на языках C и Java.

## §1.3. Анализ существующих решений

Перечисленные алгоритмы нашли свое программное воплощение. Надо отметить, что среди них достаточно много коммерческих версий, но есть и бесплатные решения.

Данные системы по количеству поддерживаемых языков программирования, можно разделить на следующие группы:

- Универсальные, могут поддерживать любой язык программирования.
- Поддерживающие ограниченный набор языков программирования.
- Узкоспециализированные, которые разработаны специально для определенного языка программирования.

Одними из наиболее доступных и распространенных решений являются:

JPlag – веб-сервис, который находит сходство между несколькими наборами файлов исходного кода [20]. JPlag в настоящее время поддерживает Java, C#, C, C++, Scheme.

MOSS (Measure Of Software Similarity) – веб-сервис обнаружения плагиата в исходных кодах, который был разработан Алексом Айкеном в 1994 году [18]. Сервис является бесплатным, но для его использования нужна регистрация. Поддерживает программы, написанные на C, C++, Java, Pascal, Ada и ряд других языков программирования. Сервис не позиционируется, как полностью автоматический. В качестве результата сервис предоставляет отчет в формате html, поэтому результат требует ручной проверки.

Apollo – утилита, созданная в рамках исследования для решения проблемы дедупликации кода [19].

Codequiry – платный веб-сервис, позиционирующий себя как средство для обнаружения плагиата исходного кода. Имеет поддержку Java, C, C++ и Python.

SIM – программа с открытым исходным кодом. Используется "Амстердамским свободным университетом", для тестирования сходства в исходных кодах программ и обычных текстах;

CloneDR - инструмент, по словам авторов, который должен ускорить отслеживание и удаление дублирующегося кода для снижения затрат на поддержание кода. Не чувствителен к таким изменениям как форматирование кода, переименованию переменных и даже перестановке участков кода.

SourcererCC – это детектор клонов кода на основе токенов для очень больших баз кода и репозиториев интернет-проектов. SourcererCC способен обнаруживать клонов между файлами, методами, выражениями или блоками, на любом языке программирования.

Изначально методы для обнаружения плагиата в исходном коде были созданы вместе с методами текстовых документов или как дальнейшее их развитие.

Основные недостатки, с которыми можно столкнутся при использовании имеющихся в настоящее время систем:

- Устаревание
- Поддержка ограниченного числа языков программирования
- Закрытость
- Невозможность масштабирования

С проблемой устаревания можно встретиться, когда нужно обработать работы, выполненные на более современной версии языка программирования. Как пример, можно упомянуть систему JPlag. Хотя система заявляет о поддержке языка Java, она ограничена 7 версией. Текущей актуальной версией языка программирования Java является 13. Аналогичная ситуация возникает с любым из языков программирования. В новых версия появляется новый

синтаксис, который может повлиять на результат системы или сделать её полностью неработоспособной.

Как правило, задание для выполнения может оставаться одним и тем же, но с развитием технологий меняться язык программирование, в вузах с развитием технологий один набор изучаемых языков сменяется другим. Либо, не всегда по какому-либо курсу есть ограничение на каком языке программирования должен выполнять задание студент.

Закрытость. В качестве аргумента можно привести систему MOSS, распространяющуюся по модели SaaS. Вы можете только загрузить исходные файлы, а она генерирует отчет, основанный на них. В данном случае есть жесткие требования к формату загружаемых файлов, нет возможности гибко формировать итоговый отчет.

Проблема хранения данных. Задания, как правило, могут повторяться из года в год. При использовании готовых средств, данные работы требуется дополнительно размещать наряду с новыми, что вызывает лишние хлопоты.

## §1.4. Требования к решению

Набор токенов является эффективной формой представления исходного кода. Если есть необходимость проводить более сложные анализы, то нужно преобразовать этот набор в подходящую форму. Почти каждый инструмент, который работает непосредственно с набором токенов, использует свою собственную форму, в которой он хранит эти токены. В качестве универсального представления часто используется абстрактное синтаксическое дерево. Его главное преимущество по сравнению с алгоритмом, работающим с текстом или токенами, заключается в том, что он описывает структуру программы без конкретных деталей реализации.

Существуют работы, описывающие различные способы использования абстрактного синтаксического дерева для выявления сходства в исходных

кодах. Эти алгоритмы основаны на простой идее сравнения древовидных структур. Поскольку сравнение древовидных структур является сложной операцией, алгоритмы используют преобразование этих деревьев в линейные формы.

В настоящее время для этого используются два основных подхода:

- Хэширование
- Характеристические векторы

Идея хэширования основана на том, что каждый узел дерева как источник информации можно преобразовать в уникальный набор символов, который присущ только данному узлу. Такая операция позволяет достаточно быстро создать описание для достаточно больших деревьев. Впоследствии, используя специальный алгоритм, эти хэши сравниваются и определяются аналогичные части исходного кода. Эта идея подробно описана, например, в работе усовершенствованного алгоритма обнаружения плагиата на основе абстрактного синтаксического дерева [12, 13]. В ней авторы также описывают возможности дальнейшей оптимизации алгоритмов поиска сходств с использованием АСД.

Другим часто используемым вариантом является представление исходного кода с помощью векторов.

Прежде чем предлагать свой собственный метод для поиска плагиата в исходном коде, сформулируем требования и характеристики предлагаемого метода. Основные требования включают в себя:

- Универсальность
- Создание отчета для конкретного задания

Большинство систем в настоящее время используют для обнаружения плагиата в исходном коде те же методы, что используются в текстовых

документах. В отличии от других подходов, воспользуемся синтаксическими деревьями в качестве основы для предлагаемого способа.

Одним из требований так же было универсальность. Должна быть легкая возможность адаптироваться, то есть должна быть возможность добавления поддержки нового языка программирования.

# ГЛАВА 2. ПОСТРОЕНИЕ МОДЕЛИ

## §2.1. Абстрактное синтаксическое дерево

На начальном этапе необходимо исходный код каждой сравниваемой программы преобразовать в новое представление – абстрактное синтаксическое дерево.

Абстрактное синтаксическое дерево – это представление исходного кода программы в виде ориентированного дерева, в котором внутренние узлы представляют подструктуры, которые содержат вложенные подструктуры (например, вызов метода содержит возвращаемое значение, имя метода, список параметров), а листья представляют подструктуры, которые не содержат вложений (например, имя метода, имя переменной).

АСД является одним из наиболее распространенных способов представления исходного кода в виде древовидной структуры.

Рассмотрим на простом примере. Построим дерево для небольшого фрагмента исходного кода, выглядящим следующим образом:

```
if (a < b)
{
        b = a;
}
```

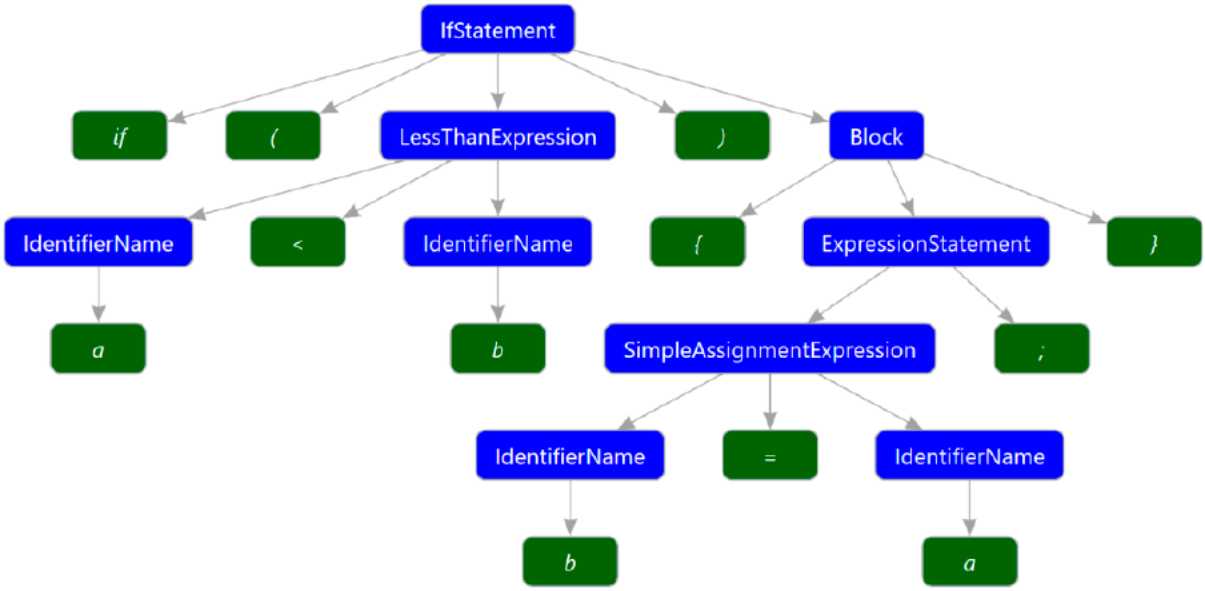


Рисунок 1. Пример АСД

Его синтаксическое дерево можно увидеть на рисунке 1. На этом дереве можно увидеть два основных вида узлов. Первые из них, показанные на иллюстрации синим цветом, представляют некоторое выражение – выражения вызова методов, бинарные, условные выражения, определения переменных и свойств. Они выражают абстрактное представление о структуре исходного кода. Вторая группа называется, показанная зеленым цветом на изображении, представляет отдельные токены (значения). Третья группа узлов, которые не показаны на рисунке, содержит элементы, которые не имеют отношения к смыслу исходного кода: пробелы, пустые строки, комментарии и тому подобное.

Важная особенность в том, что каждый узел, представляющих выражение, имеет свой собственный тип. На картинке можно видеть эти типы узлов (ForStatement, VariableDeclaration и тд). Еще одной особенностью состоит в том, что у каждого синтаксического узла должен быть хотя бы один дочерний элемент. Узлы второй и третьей группы всегда являются листьями этого дерева.

## §2.2. Идея решения

Предлагаемая модель обнаружения плагиата в исходных кодах основана на нейронной сети с использованием сиамской архитектуры, которая довольно популярна в области распознавания лиц, а также используется для многих естественных языков, в таких задачах как сходство предложений. Сиамские сети принимают на вход данные одного и того же типа. например, два изображения или два предложения, что подходит под текущую задачу.

Модель состоит из двух параллельных сетей, которые принимают на вход АСД, а затем преобразуют его в вектор, и классификатора, который выводит оценку схожести между этими двумя закодированными АСД.

В общем случае, модель состоит из следующих компонентов:

- Представление исходного кода в виде дерева
- Преобразование дерева в вектор
- Embedding слой
- LSTM слой
- Классификатор

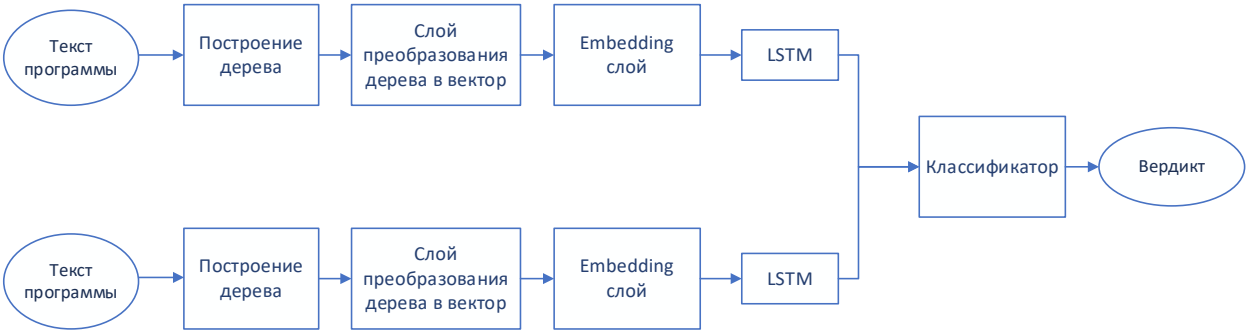


Рисунок 2. Структура сети

Для этапа генерации векторов необходимо создать словарь фиксированного размера для каждого целевого языка программирования. Этот шаг является неконтролируемым и просто требует большого количества кода для каждого целевого языка программирования. В работе рассматривается на

примере языка C#, но это также справедливо и для других языков. Более подробная информация о генерации векторов с использованием алгоритма на основе дерева, приведена ниже.

Слой преобразования АСД - преобразует АСД в вектор, где каждый элемент вектора является индексом АСД узла в словаре. АСД линеаризован путем сортировки его узлов по мере отдаления их от корня.

Embedding слой – сопоставляет каждый индекс с его векторным представлением.

LSTM (Long short-term memory) — преобразует результат предыдущего слоя в вектор в многомерном пространстве.

Классификатор - получаем оценку сходства (вероятность) между 0 и 1.

После того, как векторы сгенерированы, следующий этап — это натренировать нашу модель. Поскольку используется модель обучения с учителем, на этом этапе необходимо иметь полный набор размеченных данных содержащую информацию о том, являются ли данные пары фрагментов кода идентичными (заимствованием). В модели используется векторы, вычисленные на предыдущем шаге, для преобразования каждого узла в дереве в вектор. Построенная модель одновременно учится кодировать все абстрактное синтаксическое дерево в большой вектор, и определять, являются ли два вектора клонами или нет, используя разметку в наборе данных.

Последний шаг - фактическое обнаружение. Для этого система использует словарь с учетом значений. Используя их, система сначала векторизует все фрагменты кода, для которых необходимо выполнить обнаружение клонов, а затем запускает классификатор, обученный ранее, на каждую пару фрагментов для выполнения процесса. Пара фрагментов кода считается клоном, если его оценка сходства превышает 0,5. Чем ближе к 1, тем вероятнее, что это будет клон.

## §2.3. Векторное представление

После того как было получено дерево, необходимо преобразовать его в некоторую последовательность чисел для дальнейшей работы.

Наш алгоритм генерации векторов основан на skip-gram модели, применяемой при обработке текстов, но, в нашем случае, мы воспользуемся структурой АСД для вычисления векторного представления каждой лексемы языка программирования.

Процесс векторизации с использованием древовидной структуры, следующий:

1. Собрать большое количество исходный текстов, написанных на данном языке
2. Представить код в виде дерева
3. Сгенерировать словарь из собранных исходных текстов
4. Генерация контекстных пар с использование АСД
5. Обучение Skip-Gram модели

1) Опишем алгоритм создания словаря.

На вход модель получает АСД, который, очевидно, представляет данные в древовидной структуре. Затем, АСД отображается в вектор целых чисел, где каждое целое число представляет собой индекс токена в АСД в словаре языка программирования, на котором написан исходный код. Пример такого словаря для языка C# приведен в таблице 1.

Таблица 1

Пример словаря

| Id | Токен | Значение |
|---|---|---|
| 6 | IfStatement | |
| 10 | IdentifierName | a |

| | | |
|---|---|---|
| 11 | IdentifierName | b |
| 15 | LessThanExpression | |
| 17 | SimpleAssignmentExpression | b = a |

Более формально, если обозначить набор всех возможных АСД для исходных текстов как $T$, то процесс такого преобразования будет функцией $f_t$ со следующим определением:

$$f_t: T \rightarrow \mathbb{N}^m, \quad (1)$$

где $m$ может варьироваться в зависимости от размера входного дерева и алгоритма, используемого для $f_t$.

Сначала генерируем конечный набор лексем - словарь для целевого языка программирования. У каждой токена есть свой тип, например, IdentidierName или ExpressionStatement и может иметь значение, которое обычно является именем идентификатора и часто является уникальным.

В то время как количество типов токенов ограничено, число значений токенов бесконечно — это может быть любой заданный пользователем идентификатор. Это означает, что мы должны поставить порог на размер словаря, и нужно понимать, что в таком случае, он не будет содержат всех возможных токенов. В большинстве задач NLP (Natural Language Processing, обработки естественного языка), токены, отсутствующие в словаре, заменяются на некоторый общий «неизвестный» токен. Однако, в контексте обработки исходных кодов, вероятность столкнуться с неизвестными значениями гораздо выше, чем в NLP задачах, и поэтому для избегания такой ситуации - использования общего «неизвестного» токена поступим следующим образом: вместо этого сохраним информацию о типе токена, заменим токен на неизвестный с пустым значением, когда ни один токен с таким же типом и значением не будет найден в словаре. Такой способ позволяет сохранить семантическую информацию о токене, чтобы, например,

отличить строковый литерал и идентификатор, в случае, когда их значение не было найдено в словаре.

Если решим проигнорировать все токены, специфичные для конкретного приложения, в итоге получим предсказуемое количество токенов, определенное целевым языком программирования в его спецификации. Такой подход позволит обойти проблему с размером словаря, так как количество лексем всегда будет достаточно маленьким.

Другая проблема использования токенов для конкретного приложения – это случай, когда он встречается после того, как был сгенерирован словарь, но при этом отсутствует в нашем словаре. В таком случае его невозможно сопоставить с индексом в словаре. Стандартным подходом к этой проблеме в задачах обработки естественного языка является присвоение специального индекса, который используется для всех неизвестных токенов. Однако в нашем случае каждый токен может иметь как тип, так и его значение, поэтому мы решили использовать только тип, исключая значение, если данная пара не может быть найдена в словаре.

Это означает, что для каждого типа на целевом языке программирования, у нас будет токен, представляющий тип, не имеющий значения в словаре. При поиске токена в словаре сначала ищем токен с тем же значением и тем же типом. Если он присутствует, используем его индекс, в противном случае, возвращаемся к поиску токена с тем же самым типом и используем уже его индекс. Так как в языке программирования существует ограниченное количество типов токенов, то можем быть уверены, что найдется хотя бы соответствующий тип.

При использовании пар токенов и их значений, которые будут специфичны для приложений, количество токенов будет увеличиваться с количеством файлов, используемых для генерации словаря. По этой причине нам необходимо установить порог для числа токенов, которые мы хотим

использовать. Этот порог устанавливается экспериментально, и если в качестве этого порога задать $n$, то словарь будет состоять из $n$ наиболее распространённых токенов из файлов, которые использовались для генерации словаря.

2) Генерация данных для skip-gram модели

После создания словаря, нам нужно сгенерировать данные для обучения Skip-gram модели. В случае задач, связанных с обработкой естественного языка, входные данные рассматриваются как последовательность, а контекст того или иного слова – слова, стоящие до или после данного слова в предложении.

Мы могли бы применить тот же подход, рассматривая программу как последовательность токенов, но воспользуемся топологической информацией, содержащаяся в АСД. Поэтому нам необходимо определить контекст слова, или, скорее, токена в нашем в случае, иначе, отличным способом от последовательности.

В контексте дерева узел напрямую связан с родителем и дочерними элементами, поэтому мы можем рассматривать родителей и потомков как контекст узла. В некоторых случаях, соседние узлы также могут рассматриваться в качестве кандидата для контекста узла. Размер такого «окна» можно было бы использовать для контроля того, насколько глубоко вверх и\или вниз должен находиться контекст от текущего узла.

Однако несмотря на то, что узел будет иметь только одного родителя, он может иметь любое количество потомков, и поэтому, имея даже небольшое окно размером 3 для предков, генерировало бы только 3 узла в контексте, а если бы каждый потомок узла имел 7 детей, окно того же размера дало бы $7^3 = 343$ узла, что, вероятно, создаст шум при попытке обучения skip-gram модели.

Поэтому используем два различных параметра для управления размером «окна» для предков и размером «окна» для потомков при генерации данных для обучения нашей skip-gram модели. Когда мы включаем соседние узлы в контекст, мы решаем. использовать только прямых братьев и сестер по узлам, хотя это также может быть еще одним параметром алгоритма. Такой выбор узлов исключает случай, когда информация захватывается из середины одного метода и обрывается на середине другого.

3) Обучение skip-gram модели

После того как данные сгенерированы, следующим шагом является обучение skip-gram модели.

Skip-gram модель является разновидностью более общего механизма word2vec.

Основная идея word2vec –слова, находящиеся в похожих контекстах, являются семантически близкими. Для вычисления векторного представления слов используется искусственная нейронная сеть. Во время обучения сети с помощью модели skip-gram формируются оптимальные векторы для каждого слова.

Модель skip-gram позволяет предсказывать близлежащие слова на основании центрального слова. Задача нейронной сети заключается в следующем: для заданного слова вычислить у всех остальных слов в словаре вероятности их появления рядом с этим словом. Под словом «рядом» понимается фиксированный размер окна, к примеру 5 слов слева и 5 слов справа.

Для обучения нейронной сети используются большие наборы текстов. В качестве тренировочных примеров используются пары — «центральное слово – контекстное слово».

В нашем случае данный слой преобразовывает вектор в $\mathbb{N}^n$ в матрицу в $\mathbb{R}^{n*d_w}$, где $d_w$ размерность пространства.

$$f_w: \mathbb{N}^n \rightarrow \mathbb{R}^{n*d_w} \quad (2)$$

На самом деле результат можно представить матрицей $W$, которая имеет размеры $\mathbb{R}^{|V|*d_w}$, где $|V|$ - размер словарного запаса языка программирования, а $d_w$ - размерность пространства представления слов.

Поэтому преобразование индекса $i$ в его векторное представление сводится к выбору его индекса матрицы $W_i$, а функция $f_w$ поэтому может быть выражена как в уравнении (2), и является вычислительно недорогой.

$$f_w(i) = W_i \quad (3)$$

## §2.4. Долговременная краткосрочная память

Рекуррентные нейронные сети (Recurrent Neural Network, RNN) - класс моделей машинного обучения, основанный на использовании предыдущих состояний сети для вычисления текущего. Такие сети удобно применять в тех случаях, когда входные данные задачи представляют собой нефиксированную последовательность значений, как, например, текстовые данные, где текстовый фрагмент представлен нефиксированным количеством предложений, фраз и слов. Каждый символ в тексте, отдельные слова, знаки препинания и даже целые фразы - все это может являться атомарным элементом входной последовательности. На каждом шаге обучения $t$ значение скрытого слоя рекуррентной нейронной сети $h^t \in R^m$ вычисляется следующим образом:

$$h^t = f(Wx^t + Uh^{t-1} + b^h) \quad (4)$$

где $x^t \in R^n$ - входной вектор в момент времени $t$ (например, векторное представление текущего слова в текстовом фрагменте), $h^t$- вектор скрытого состояния в момент времени $t$,

$W \in R^{m*n}, U \in R^{m*m}$ – обучаемые параметры рекуррентной нейронной сети (матрицы весов), $b^h \in R^m$ - вектора смещения, $f$ - функция нелинейного преобразования. Чаще всего в качестве нелинейного преобразования применяют одну из следующих функций: сигмоидальная функция (5), гиперболический тангенс (6), ReLu (7):

$$f(x) = \sigma(x) = \frac{1}{1 + \exp(-x)} \tag{5}$$

$$f(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(\mathrm{x}) + \exp(-\mathrm{x})} \tag{6}$$

$$f(x) = \max(0, x) \tag{7}$$

В простой рекуррентной нейронной сети выходное значение $y^t \in R^l$ на текущем шаге $t$ вычисляется по формуле:

$$y^t = Wh^t + b, \tag{8}$$

где $W \in R^{l*m}$ и $b \in R^l$ – обучаемые параметры.

В 1997 году Зепп Хохрайтер и Юрген Шмидхубер представили новый подход, получивший название LSTM (Long Short-Term Memory - долгая краткосрочная память). Рекуррентные нейронные сети, основанные на этом подходе, имеет более продвинутый (и более сложный) способ вычисления $h^t$.

Данный способ, помимо входных значений и предыдущего состояния сети, использует также фильтры (gates), определяющие, каким образом информация будет использоваться для вычисления как выходных значений на текущем слое $y^t$, так и значений скрытого слоя на следующем шаге $h^{t+1}$. Весь процесс вычисления $h^t$ для простоты упоминается как LSTM-слой.

Рассмотрим подробнее структуру LSTM-слоя. Центральным понятием здесь является запоминающий блок (memory cell), который, наряду с состоянием сети $h$, вычисляется на каждом шаге, используя текущее входное значение $x^t$ и значение блока на предыдущем шаге $c^{t-1}$. Входной фильтр

(input gate) $i^t$ определяет, насколько значение блока памяти на текущем шаге должно влиять на результат. Значения фильтра варьируются от 0 (полностью игнорировать входные значения) до 1, что обеспечивается областью значений сигмоидальной функции:

$$i^t = \sigma\left(W^i x^t + U^i h^{t-1}\right) \quad (9)$$

«Фильтр забывания» (forget gate) позволяет исключить при вычислениях значения памяти предыдущего шага:

$$f^l = \sigma\left(W^f x^t + U^f h^{t-1}\right) \quad (10)$$

На основе всех данных, поступающих в момент времени $t$, вычисляется состояние блока памяти $c^t$ на текущем шаге, используя фильтры. Так получается новое состояние памяти, которое в таком же виде передаётся далее.

$$c^t = \tanh(W x^t + U h^{t-1}) \quad (11)$$

$$c^t = f^l * c^{t-1} + i^l * c^l \quad (12)$$

Выходной фильтр (output gate) аналогичен двум предыдущим и имеет вид:

$$o^t = \sigma(W^o x^t + U^o h^{t-1}) \quad (13)$$

Итоговое значение LSTM-слоя определяется выходным фильтром и нелинейной трансформацией над состоянием блока памяти:

$$h^t = o^t * \tanh(c^t) \quad (14)$$

Безусловно, существует множество вариаций того, какие именно функции активации используются каждым слоем, немного модифицируют сами схемы и прочее, но суть остаётся прежней — сначала забывают часть памяти, затем запоминают часть нового сигнала, а уже потом на основе этих данных вычисляется результат.

Как обсуждалось ранее, на выходе предыдущего слоя у нас будет матрица размерности $\mathbb{R}^{n*d_w}$, где $n$ зависит от размера входного АСД, а $d_w$ — это размерность пространства представления слов, который является гиперпараметром модели.

Целью слоя на основе LSTM является преобразование матрицы в вектор в пространстве $d_e$, где $d_e$ является гиперпараметром модели, который захватывает связь между элементами входной матрицы. Эта задача эквивалентна преобразованию матрицы, представляющей предложение в вектор. Таким образом, этот слой будет функцией следующего вида:

$$f_e: \mathbb{R}^{n*d} \rightarrow \mathbb{R}^{d_e} \quad (15)$$

Интересным свойством этой функции является то, что выходное измерение не зависит от входного, что позволяет легко агрегировать входные данные, которые изначально имели различные размеры, в следующем слое модели.

Для выполнения этой задачи мы используем модель LSTM, и подаем ей результат предыдущего слоя. $d_e$ выбирается экспериментально, и обычно составляет от 50 до 200 в зависимости от других параметров модели.

Есть и другие факторы, которые необходимо принять при построении данного слоя. Сначала нужно решить, будет ли LSTM двунаправленным или нет. Сеть основана на той идее, что выход может зависеть не только от предыдущих элементов в последовательности, но и от будущих. Когда LSTM двунаправлен, на самом деле будет две LSTM: одна для прямого направления вывода данных и LSTM для обратного направления. После вычислений, результаты двух LSTM будут объеденены, чтобы дать конечный результат.

Еще одним решением для энкодера на базе LSTM является вопрос о том, следует ли использовать один слой LSTM, или выстраивать несколько LSTM друг за другом. При суммировании нескольких LSTM, первый LSTM работает точно так же, как и предыдущий, однако его выход не используется напрямую, а подается на другой LSTM, и только результат последнего используется для следующего шага.

## §2.5. Процесс классификации

До сих пор внимание было сосредоточено на том, как преобразовать один АСД в вектор в $\mathbb{R}^{d_e}$. Однако наша модель принимает на вход два параметра и получает один на выходе. Поэтому нам необходимо объединить эти два входа каким-то образом, чтобы получить одно единственное значение, которое даст понять, являются ли два заданных входа схожими или нет. Для этого создадим еще один слой в нашей модели, который будет принимать два входа и возвращать один выход. Таким образом, данный слой определяется функцией $f_m$, как показано в уравнении, где $n$ обычно зависит от $d_e$.

$$f_m: \mathbb{R}^{d_e} \times \mathbb{R}^{d_e} \to \mathbb{R}^n \tag{16}$$

Существует огромный спектр метрик расстояний, которые можно использовать для определения сходства двух фрагментов кода. По сути, все они опираются на возможность представлять их как точки в пространстве, относительная близость которых определяет их сходство.

Рассуждая об измерении расстояний между двумя точками, мы обычно имеем в виду соединяющую их прямую линию, или евклидово расстояние как самое интуитивное для понимания.

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{17}$$

Другой распространенной метрикой является метрика $L1$. Расстояние $L1$ также известно как расстояние городских кварталов, манхэттенское расстояние, метрика прямоугольного города — оно измеряет дистанцию не по кратчайшей прямой, а по блокам. Расстояние $L1$ измеряет дистанцию между городскими блоками: это расстояние всех прямых линий пути.

$$d(x, y) = \|x - y\|_1 = \sum_{i=1}^{n} |x_i - y_i| \tag{18}$$

Однако с ростом словаря корпуса также растет его размерность – и она редко бывает равномерно распределена. По этой причине перечисленные меры расстояния не всегда оказываются эффективными, потому что предполагают симметричность данных и равенство расстояний во всех измерениях.

По умолчанию в гиперпараметрах модели в качестве метрики сходства часто выбирается евклидово расстояние, но нередко косинусное расстояние позволяет добиться лучших результатов.

В качестве используемой меры как раз и было выбрано косинусное сходство. Математически косинусное сходство измеряет косинус угла между двумя векторами, спроецированными в многомерном пространстве. При построении в многомерном пространстве косинусное сходство фиксирует ориентацию (угол) каждого вектора, а не величину.

$$cos\theta = \frac{\vec{a} * \vec{b}}{\|\vec{a}\|\|\vec{b}\|} = \frac{\sum_{i=1}^{n} a_i \times b_i}{\sqrt{\sum_{i=1}^{n}(a_i)^2} \times \sqrt{\sum_{i=1}^{n}(b_i)^2}} \quad (19)$$

Сходство по косинусу выгодно, потому что даже если два одинаковых документа находятся далеко друг от друга на евклидовом расстоянии из-за размера, они все равно могут иметь меньший угол между ними. Чем меньше угол, тем выше сходство.

Как только мы получаем один вектор, содержащий информацию о двух входных фрагментах кода, мы используем перцептрон, чтобы предсказать, были ли входные данные схожими или нет. Выходной слой нашей нейронной сети использует сигмоидную функцию и поэтому выводит действительное число между 0 и 1, которое можно интерпретировать как вероятность. Таким образом, нейронная сеть обратной связи может быть представлена функцией $f_p$, определенной следующим образом.

$$f_p: \mathbb{R}^{d_o} \rightarrow [0,1] \quad (20)$$

Пару фрагментов кода будем считать схожими, если вероятность превышает 0,5, иначе – нет. Чем ближе к 1, тем вероятнее, что это будет клон.

# ГЛАВА 3. ЧИСЛЕННЫЕ ЭКСПЕРИМЕНТЫ

В этой главе приведены проведенные эксперименты и представлены полученные результаты. В параграфе 3.1 представлены технические детали решения, а в параграфе 3.3 описаны проведенные эксперименты по обнаружению заимствований, а также, какие данные использовались для обучения модели и как модель работает с различными параметрами.

## §3.1. Программная реализация

Реализация выполнена в виде web приложения.

Приложение написано на языке Python с использованием TensorFlow и Flask. Код, отвечающий за построение модели, приведен в приложении 1. Часть, отвечающая за генерацию дерева выполнена на C#. Такой выбор был сделан в угоду того, что средства, работающие на том же самом языке программирования, что и анализируемые нами данные лучше пригодны для этого. Есть различные кроссплатформенные средства, например, ANTLR, который поддерживает множество языков для генерации деревьев. Но данный инструмент ограничен поддерживаемыми им набором лексем и грамматик и требует периодического их обновления для встроенного парсера для поддержания новый возможностей языка. Принятое решение обходит данную проблему и перекладывает ответственность на разработчиков языка, тем самым позволяет довольно легко масштабировать для применения с другими языками программирования.

Данное приложение позволяет формировать задачи, а рамках которого решения будут проверяться.

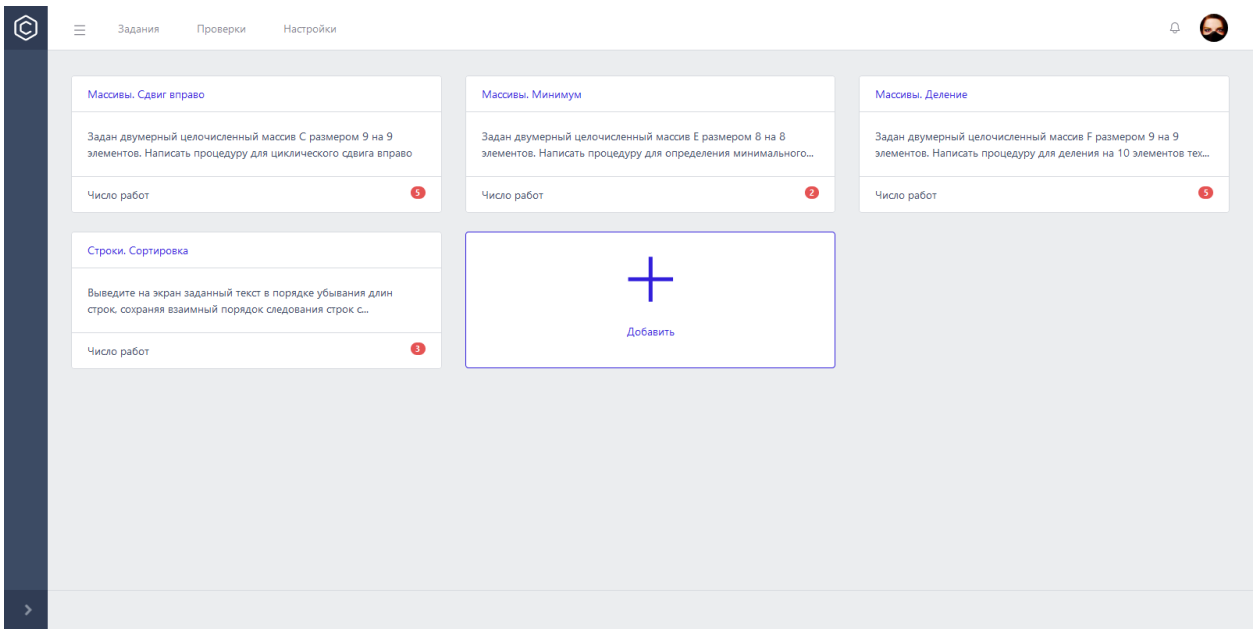


Рисунок 3. Формирование задач

Выполнять загрузку решения.

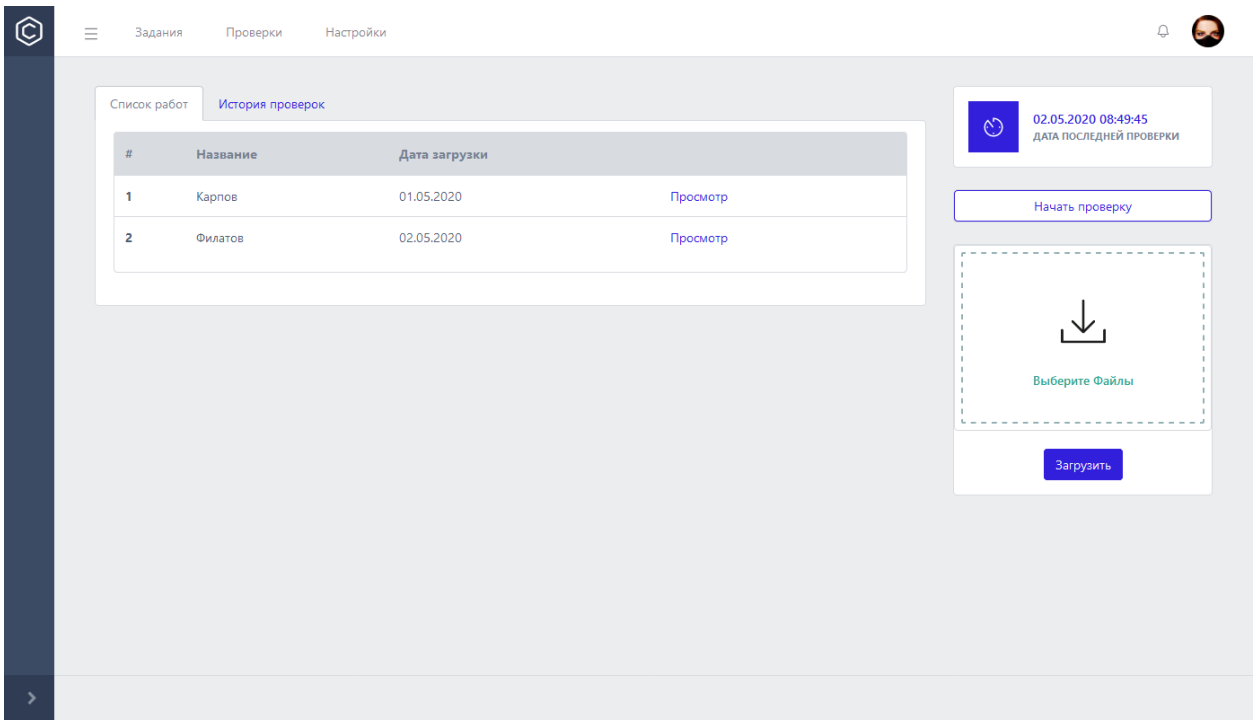


Рисунок 4. Список задач

Одной из главных частей приложения является преобразование входных исходных кодов программ в удобный для сравнения вид. После того, как пользователь указал файл (загружаемое решение), который подлежит сравнению, файл автоматически отправляется на сервис, соответствующий

языку программирования, на котором он написан, для построения дерева с последующим сохранением его в базе данных.

Выполнять и просматривать историю проверок.

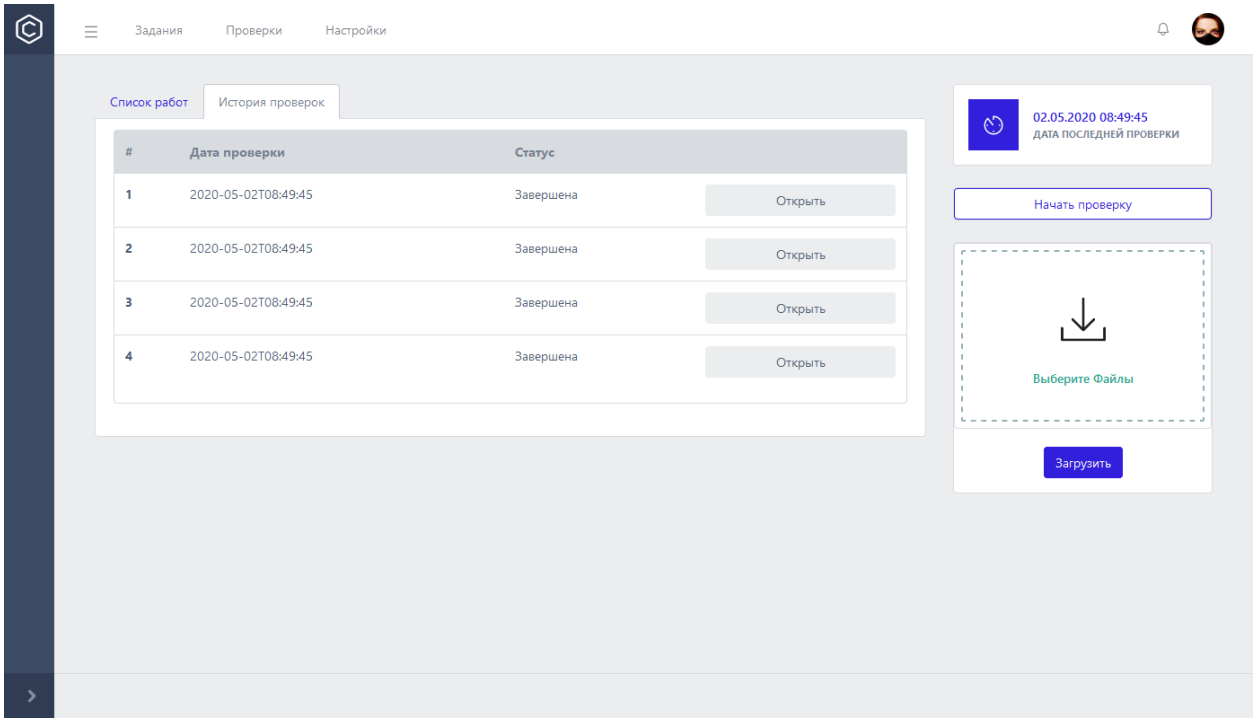


Рисунок 5. История проверок

После того, как началась проверка, данные (файлы на момент загрузки), фиксируются для истории. Выбирается список файлов, присутствующих на данный момент, выбираются ранее сгенерированные деревья и строятся всевозможные пары деревьев, которые в дальнейшем отдаются на распознавание нашей модели. Результат распознавания заносится сохраняется в базу данных.

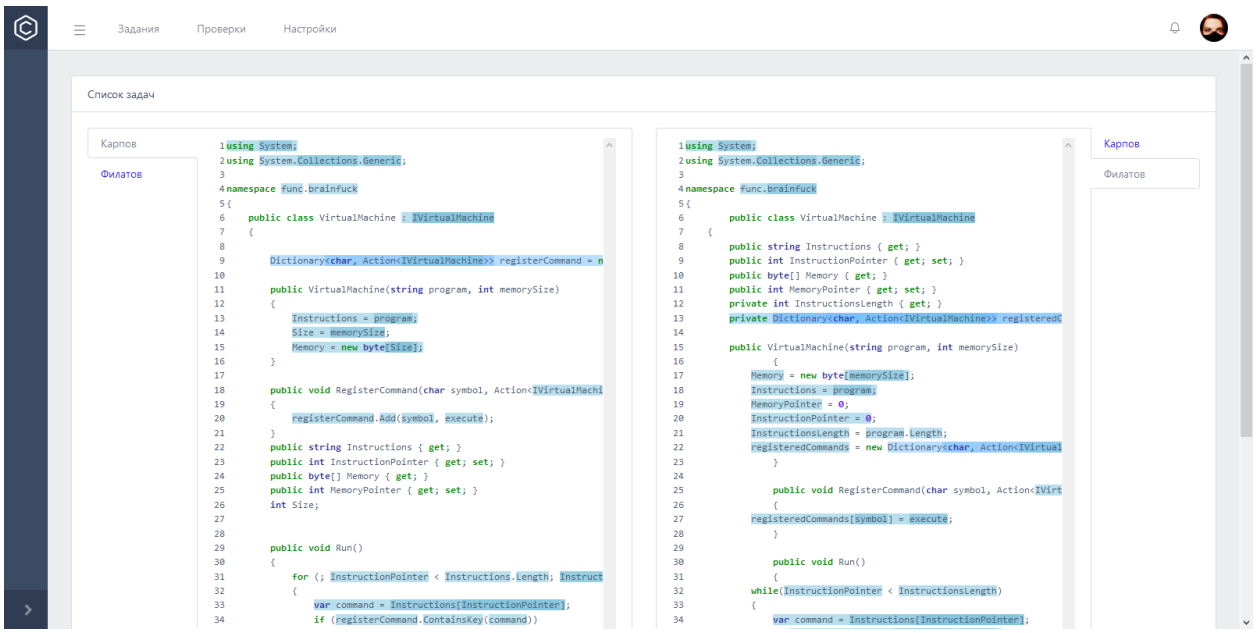


Рисунок 6. Пример результата

После проверки можно увидеть, какой фрагмент из чьей работы и с какой вероятностью был заимствован.

## §3.2. Исходные данные

Так как мы используем обучение с учителем для выявления схожих кодов, для обучения нашей модели нам необходимы данные, которые удовлетворяют следующим свойствам:

1. Несколько фрагментов кода должны реализовывать одну и ту же функциональность
2. Должна присутствовать информация о том, реализуют ли два фрагмента кода одну и ту же функциональность (разметка выборки)

Исходный набор данных представляет собой подборку решений, представленных студентами для выполнения заданий по курсам программирования на языке C#. Решение одной задачи реализуется некоторым количеством людей. Все решения одной задачи должны реализовывать одинаковую функциональность, поэтому мы уверены, что все исходные коды, реализующие решение одной и той же задачи это, по крайней

мере, функционально схожи. Чем проще проблема, тем выше вероятность того, что фрагменты кода, реализующие решение одной и той же проблемы, будут очень похожи друг на друга.

Так как данная реализация в настоящее время поддерживает только C#, то использовались исходные тексты только для этого языка программирования. Для каждого файла были сгенерированы «положительные» и «отрицательные» пары.

Таблица 2

Характеристики набора данных

| Параметр | Мин. | Ср. | Макс. |
|---|---|---|---|
| Кол-во файлов | 1220 | | |
| Число строк | 10 | 100 | 1984 |
| Число токенов | 11 | 515 | 10877 |
| Число узлов | 10 | 514 | 10876 |
| Глубина дерева | 5 | 18 | 66 |

В таблице можете видеть некоторые характеристики исходного набора данных.

## §3.3. Оценка модели

В этом разделе содержится описание проведенных экспериментов, произведенных для оценки модели, а также содержатся результаты и будет дана интерпретация полученных результатов.

Сначала мы подготовили набор данных для обучения нашей модели, разделив данные, которые мы описали в параграфе. на тренировочный набор,

содержащий 70% данных и тестовый, содержащие 30% данных, используемый для окончательной оценки нашей модели. Как для обучения, так и тестирования, мы обрабатываем файлы, реализующие решение той же задачи, и случайно выбираем $n$ штук из файлов, реализующих решение к другой задаче, чтобы использовать в качестве отрицательной пары для модели.

Чтобы оценить точность модели, полученные вероятности необходимо преобразовать в значения целевой переменной.

Сравнив прогнозы модели с фактическими данными, мы получим 4 группы прогнозов:

1. Истинно положительные (TP).
2. Истинно отрицательные (TN).
3. Ложноположительные (FP).
4. Ложноотрицательные (FN).

Одной из наиболее распространенных и простых для понимания мер является accuracy. Accuracy, как показано в уравнении, является долей образцов по которым классификатор принял правильное решение, деленное на общее количество прогнозируемых образцов.

$$accuracy = \frac{TP + TN}{TP + FN + TN + FP}, \quad (21)$$

Accuracy имеет диапазон $[0; 1]$, чем выше значение, тем лучше. Она широко используется для оценки моделей, но имеет недостатки в зависимости от набора данных. Общая проблема с измерение точности связана с несбалансированными наборами данных - набором данных с гораздо большим количеством отрицательных выборок, чем положительных, или наоборот.

Чтобы иметь возможность получить лучшее понимание производительности модели, особенно когда набор данных не сбалансирован, часто используются точность и полноту. Точность модели, учитывая по

уравнению — это соотношение правильно спрогнозированных образцов к общему количеству образцов, которое были оценены как положительные.

$$precision = P = \frac{TP}{TP + FP} \quad (22)$$

Точность также имеет диапазон $[0;\ 1]$, при этом 1 является наилучшей достижимой точностью, то есть все образцы, предсказанные как положительные, на самом деле были положительными. Мера точности, однако, не дает представление о том, все ли правильные ответы вернул классификатор. Поэтому для этого существует и часто используется так называемая мера полноты (R, recall, представляющая собой отношение правильно спрогнозированных образцов к общему количеству положительных образцов.

$$recall = R = \frac{TP}{TP + FN} \quad (23)$$

Precision и Recall дают довольно исчерпывающую характеристику классификатора, причем «с разных углов». Обычно при построении подобного рода систем приходится все время балансировать между двумя этими метриками. Если вы пытаетесь повысить Recall, делая классификатор более «оптимистичным», это приводит к падению Precision из-за увеличения числа ложноположительных ответов. Если же вы настраиваете свой классификатор, делая его более «пессимистичным», например, строже фильтруя результаты, то при росте Precision это вызовет одновременное падение Recall из-за отбраковки какого-то числа правильных ответов. Поэтому удобно для характеристики классификатора использовать одну величину, так называемую метрику $F_1$ — это среднее гармоническое значение таких мер как точность и полнота, как показано в уравнении:

$$F_1 = \frac{2}{\frac{1}{P} + \frac{1}{R}} = 2\frac{P * R}{P + R} \quad (24)$$

Так как $F_1$ — это среднее гармоническое значение двух мер с диапазоном $[0;\ 1]$, оно также имеет тот же диапазон, а что касается точности и полноты, то более высокое $F_1$ означает, что модель работает лучше.

Преимуществом этой меры является то, что данная формула придает одинаковый вес точности и полноте, поэтому $F_1$ будет падать одинаково как при уменьшении точности, так и полноты.

В таблице приведены некоторые параметры модели, при которых проводились испытания.

Таблица 3

Параметры модели

| № | $\|V\|$ | $d_w$ | $d_e$ |
|---|---|---|---|
| 1 | 157 | 100 | 32 |
| 2 | 157 | 100 | 64 |
| 3 | 157 | 100 | 64, 32 |
| 4 | 157 | 100 | 128, 64 |

где $|V|$ – размер словаря, $d_w$ – размерность пространства векторного представления, $d_e$ – число и размерность LSTM слоев.

В таблице, показаны различные метрики для каждой модели при тестировании.

Таблица 4

Результаты испытаний

| № | Accuracy | Precision | Recall | $F_1$ |
|---|---|---|---|---|
| 1 | 0,72 | 0,78 | 0,61 | 0,69 |
| 2 | 0,72 | 0,75 | 0,66 | 0,70 |
| 3 | 0,72 | 0,80 | 0,61 | 0,69 |
| 4 | 0,72 | 0,76 | 0,65 | 0,70 |

В целом модель показывает достаточно высокую отзывчивость и точность. Если же применять данную модель для работы с текстами на разных языках программирования, то предпочтительно сохранять высокую отзывчивость и отдавать ложноположительные срабатывания эксперту. Это связано с тем, что некоторые исходные тексты, которые по факту не имеют ничего общего, могут выглядеть очень похоже и быть сложными для классификации. Например, код, реализующий нахождение чисел Фибоначчи и расчет факториала, содержат цикл, который выполняет какую-то арифметическую операцию, и в конце возвращает число. Несмотря на то, что коды выполняют разный функционал, они содержат достаточно сходств, чтобы они были определены как похожие, поэтому оценка модели может быть заниженной.

Еще один вывод, который можно сделать на основе полученных результатов, заключается в том, что увеличение размерности LSTM слоя улучшает характеристики модели. Увеличение же их количества не приводит к существенному улучшению. Производительность модели, скорее всего, зависит от того, как преобразовываем АСД в вектор, а не от классификации векторов.

## §3.4. Сравнение с аналогами

Для оценки результатов сравним применяемый подход с другими программными решениями. Необходимо чтобы программное средство было свободно доступно, была заявлена поддержка языка и имелась возможность экспорта результата. В качестве кандидатов для сравнения было рассмотрено несколько вариантов.

От использования JPlag пришлось отказаться по причине отсутствия поддержки актуальной версии языка C#, на которой выполнены работы. Последняя поддерживаемая версия языка – 1.2 (2003г.).

От MOSS отказался по той причине, что оно является клиент-серверным приложением с рядом ограничений как на общее число проверяемых работ, так и число возможных проверок в день.

CloneDr имеет ограничение в ознакомительной версии, формирует отчет только о десяти найденных совпадениях.

В качестве кандидата для сравнения был выбран SourcererCC как удовлетворяющий всем требованиям.

SourcererCC – это детектор клонов кода на основе токенов для очень больших баз кода и репозиториев проектов. Он способен обнаруживать клонов между файлами, методами, выражениями или блоками, на любом языке программирования.

SourcererCC работает в два основных этапа: создание индекса, обнаружение клонов.

На первом этапе он анализирует блоки кода (последовательность операторов языка в фигурных скобках) из исходных файлов с помощью парсера, который знает о токенах и семантике блоков данного языка программирования. Из блоков кода он строит индекс, сопоставляющий токены с блоками, которые их содержат.

На этапе обнаружения клонов SourcererCC проходит через все блоки кода, извлекая из индекса блоки-кандидаты. Он не создает индекс всех токенов в блоках кода, вместо этого он использует эвристический метод для фильтрации. После того как кандидаты найдены, SourcererCC использует другой эвристический фильтр, который учитывает порядок расположения токенов в коде для оценки сходства между ними. Блоки - кандидаты, чья верхняя граница сходства оказывается ниже, чем установленный порог, отбрасываются без дальнейшей обработки. Это повторяется до тех пор, пока не будут найдены клоны для каждого блока кода.

Для проведения эксперимента используем тот же тренировочный набор файлов, на котором тестировалась модель.

В таблице приведены используемые параметры для тестирования.

Таблица 5

Параметры SourcererCC

| Параметр | Значение |
|---|---|
| MIN_TOKENS | 65 |
| MAX_TOKENS | 250000 |
| SHARD_MAX_NUM_TOKENS | 65,100,300,500000 |
| BTSQ_THREADS | 4 |
| BTIIQ_THREADS | 4 |
| BTFIQ_THREADS | 4 |
| QLQ_THREADS | 4 |
| QBQ_THREADS | 4 |
| QCQ_THREADS | 4 |
| VCQ_THREADS | 16 |
| RCQ_THREADS | 4 |

Пример нескольких итоговых результатов приведен в таблице 6.

Таблица 6

Результаты испытаний

| Пороговое значение | Precision | Recall | $F_1$ |
|---|---|---|---|
| 0.6 | 0,61 | 0,82 | 0,7 |
| 0.7 | 0,71 | 0,8 | 0,75 |
| 0.8 | 0,77 | 0,75 | 0,76 |

Можно видеть, что предложенный алгоритм не сильно отстает и показывает схожую точность.

SourcererCC использует пороговое значение для определения являются ли два фрагмента кода клонами или нет. Если величина этого порогового значения составляет 0.5, то хотя бы 50% токенов между двумя фрагментами будут совпадать и следует их рассматривать как клоны. Чем выше порог, тем больше фрагменты кода должны быть схожи между собой. По этой причине, с повышением порога повышается Precision, а Recall падает.

Так же причиной разницы между решениями может заключаться в том, что два фрагмента кода считаем, как схожие только в том случае, если они к тому же реализуют точно такую же функциональность. Тем не менее, некоторые пары могут иметь очень близкую структуру, которая могла бы делать их хорошими кандидатами, но они не реализуют ту же функциональность. Например, когда оба кода делают нечто весьма похожее, но не подходят под наше определение. Но при этом фрагменты кода достаточно похожи, чтобы их можно было сопоставить.

# ЗАКЛЮЧЕНИЕ

Таким образом, в процессе выполнения работы были выполнены основные цели и задачи.

В ходе выполнения данной работы было разработано программное обеспечение, которое позволит выявлять случаи "списывания" программ, автоматически сравнивать исходные тексты и позволяла находить похожие или идентичные фрагменты в исходных кодах программ, оценивать степень обнаруженного сходства и наглядно представлять результат сравнения.

Для достижения указанно цели были:

- исследованы существующие методики выявления заимствований исходного кода программ.
- рассмотрены различные типы изменений, вносимых в заимствованный исходный код, с целью сокрытия факта заимствования.
- построены представления исходного кода в виде абстрактного синтаксического дерева.
- выполнена проверка эффективности данного подхода по определению сходства. Проверка проводилась на наборе реальных студенческих проектов, представляющих решения практических заданий.

## СПИСОК ЛИТЕРАТУРЫ


1. Baker B. S. Finding clones with dup: Analysis of an experiment //IEEE Transactions on Software Engineering. – 2007. – Т. 33. – №. 9. – С. 608-621.
2. Cordy J. R., Roy C. K. The NiCad clone detector //2011 IEEE 19th International Conference on Program Comprehension. – IEEE, 2011. – С. 219-220.
3. Curtis, G. "Is plagiarism changing over time? A 10-year time-lag study with three points of measurement." / G. Curtis, L. Vardanega – Higher Education Research & Development, 2016 – 1167-1179 p.
4. Ganguly D. et al. Retrieving and classifying instances of source code plagiarism //Information Retrieval Journal. – 2018. – Т. 21. – №. 1. – С. 1-23.
5. Ghofrani, J. A Conceptual Framework for Clone Detection using Machine Learning / J. Ghofrani, M. Mohseni, A. Bozorgmehr – 2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI), 2017 – 810-817 p.
6. Haider K. Z. et al. Efficient Source Code Plagiarism Identification Based on Greedy String Tilling //IJCSNS. – 2010. – Т. 10. – №. 12. – С. 204.
7. Hiroaki Murakami. Fast and Precise Token-Based Code Clone Detection. – 2016.
8. Kamalpriya C. M., Singh P. Enhancing program dependency graph based clone detection using approximate subgraph matching //2017 IEEE 11th International Workshop on Software Clones (IWSC). – IEEE, 2017. – С. 1-7.
9. Katta, JYB. Machine Learning for Source-code Plagiarism Detection / JYB. Katta – International Institute of Information Technology, 2018 – 71 p.
10. Kim J. et al. Measuring source code similarity by finding similar subgraph with an incremental genetic algorithm //Proceedings of the Genetic and Evolutionary Computation Conference 2016. – 2016. – С. 925-932.
11. Kodhai E., Kanmani S. Method-level code clone detection through LWH (Light Weight Hybrid) approach // Journal of Software Engineering Research and Development. – 2014. – Т. 2. – №. 1. – С. 12.

12. Lazar, F.M. Clone detection algorithm based on the Abstract Syntax Tree approach / F.M. Lazar, O. Banias – 2014 IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI), 2014 – 73-78 p.
13. Ryota A. Code Clone Detection Method Based on the Combination of Tree-Based and Token-Based Methods / A. Ryota, H. Hirohide – Journal of Software Engineering and Applications 10, 2017 – 891-906 p.
14. Sheneamer A., Kalita J. Semantic clone detection using machine learning //2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA). – IEEE, 2016. – С. 1024-1028.
15. Tajima, R. Detecting functionally similar code within the same project / R. Tajima; M. Nagura; S. Takada – 2018 IEEE 12th International Workshop on Software Clones (IWSC), 2018 – 51-57 p.
16. Zhang J. et al. A novel neural source code representation based on abstract syntax tree //2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). – IEEE, 2019. – С. 783-794.
17. Zheng M., Pan X., Lillis D. CodEX: Source Code Plagiarism Detection Based on Abstract Syntax Tree //AICS. – 2018. – С. 362-373.
18. Программа обнаружения плагиата в программных текстах MOSS [Офиц. сайт]. URL: http://theory.stanford.edu/~aiken/moss/ (дата обращения: 23.12.2018)
19. Программа поиска дублирующих фрагментов кода Apollo [Офиц. сайт]. URL: https://github.com/src-d/apollo (дата обращения: 23.04.2019)
20. Программа поиска дублирующих фрагментов кода JPlag [Офиц. сайт]. URL: https://jplag.ipd.kit.edu/ (дата обращения: 23.04.2019)
21. Узлов, Н. Д. Студенческий плагиат: состояние проблемы, последствия и возможные пути решения / Н.Д. Узлов // Аллея Науки: научно-практич. электронный журнал. – 2019. – №2 (29). – С.867-876
22. Чиркин Е.С. Использование систем антиплагиата в образовании / Е.С. Чиркин // Вестник Тамбовского университета, 2013 – 3380-3387 с.

Приложение

# ПРОГРАММНЫЙ КОД ПОСТРОЕНИЯ СЕТИ

```
class TFModel(object):
  def __init__(self):
    self.EPOCHS = 3
    self.BATCH_SIZE = 64
    self.CHECKPOINT_PATH = "./checkpoints/"
    self.WEIGHT_PATH = "./weights/"
    self.VOCAB_PATH = "c:/AST_DATA/pav/pav.tsv"
    self.VOCAB_OFFSET = 1
    self.EMB_PATH = "c:/AST_DATA/pav/emb-100.npy"
    self.EMB_DIM = 100


  def get_embeddings(self, index):
    vocab = Vocabulary.from_file(self.VOCAB_PATH)
    embedding_input_size = len(vocab) + self.VOCAB_OFFSET
    kwargs = dict(
      name="embedding_{0}_{1}".format("csharp", index),
      mask_zero=True
    )
    weights = np.load(self.EMB_PATH, allow_pickle=False)
    padding = np.zeros((self.VOCAB_OFFSET, self.EMB_DIM))
    kwargs["weights"] = [np.vstack([padding, weights])]


    return Embedding(embedding_input_size, self.EMB_DIM, trainable=False, **kwargs)


  def get_lstm(self, out_dim: int, index: int, position: int, return_sequences: bool,
use_bidirectional: bool = False):
    lstm = LSTM(out_dim, return_sequences=return_sequences,
name="lstm_{0}_{1}_{2}".format('csharp', index, position))
    return lstm


  def get_encoder(self, index: int, use_bidirectional: bool = False):
```

```
    ast_input = Input(shape=(None,), dtype=tf.dtypes.int32,
name="input_{0}_{1}".format('csharp', index))

    x = self.get_embeddings(index)(ast_input)

    lstm_dim = [32]
    for i, n in enumerate(lstm_dim[:-1]):
      x = self.get_lstm(n, index, i, True, use_bidirectional)

    x = self.get_lstm(lstm_dim[-1], index, len(lstm_dim), False, use_bidirectional)(x)

    output_dimension = 64

    hash_dim = [20]
    for i, dim in enumerate(hash_dim):
      is_last = i == len(hash_dim) - 1
      activation = None if is_last else "relu"
      x = Dense(dim, use_bias=not is_last, activation=activation,
          name="dense_{0}_{1}_{2}".format("csharp", index, i))(x)
      if is_last:
        output_dimension = dim

    encoder = Model(inputs=ast_input, outputs=x,
          name="encoder_{0}_{1}".format('csharp', index))

    encoder.output_dimension = output_dimension

    return ast_input, encoder


  def get_merge_model(self, input1, input2):
    x = Dot(axes=1, normalize=True)([input1, input2])
    for i, layer_size in enumerate([64, 32]):
```

```
    name = "distance_dense_{0}".format(i)
    x = Dense(layer_size, activation="relu", name=name)(x)
  x = Dense(1, activation="sigmoid", name="output")(x)

  model = Model(inputs=[input1, input2], outputs=x, name="merge_model")

  return model

def get_merge_input(self, input_dimension: int, index: int):
  return Input(shape=(input_dimension,),
         name="encoded_{0}_{1}".format("csharp", index))

def get_model(self):
  with tf.device('/device:GPU:0'):
    input1, encoder1 = self.get_encoder(1, False)
    input2, encoder2 = self.get_encoder(2, False)

    merge_input1 = self.get_merge_input(encoder1.output_dimension, 1)
    merge_input2 = self.get_merge_input(encoder2.output_dimension, 2)
    merge_model = self.get_merge_model(merge_input1, merge_input2)

    output1 = encoder1(input1)
    output2 = encoder2(input2)

    result = merge_model([output1, output2])

    model = Model(inputs=[input1, input2], outputs=result)
    model.compile(loss='binary_crossentropy', optimizer='adam')

    return model

def get_metrics(self, labels, predictions):
  return {
```

```
    "samples_count": len(labels),
    "positive_samples_count": len([labels for t in labels if t == 1]),
    "accuracy": float(metrics.accuracy_score(labels, predictions)),
    "precision": float(metrics.precision_score(labels, predictions)),
    "recall": float(metrics.recall_score(labels, predictions)),
    "avg_precision": float(metrics.average_precision_score(labels, predictions)),
    "f1": float(metrics.f1_score(labels, predictions)),
   }

 def train(self, train_value, test_value=None):
   model = self.get_model()

   cp_callback = tf.keras.callbacks.ModelCheckpoint(self.CHECKPOINT_PATH + "cp-
{epoch:04d}/", save_weights_only=True, verbose=1)

   model.fit_generator(train_value, self.BATCH_SIZE,
           callbacks=[cp_callback],
           epochs=self.EPOCHS, verbose=1)

   model.save_weights(self.CHECKPOINT_PATH + 'last/')
   model.save(self.CHECKPOINT_PATH + 'last/model.h5')

   if (test_value is not None):
    y_pred = model.predict(test_value)
    del model
    return y_pred
   del model
   return None
 def predict(self, value):
   model = keras.models.load_model(self.CHECKPOINT_PATH + 'last/model.h5')
   y_pred = model.predict(value)
   del model
   return y_pred
```