Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский университет «Высшая школа экономики»

Факультет Санкт-Петербургская школа физико-математических и компьютерных наук
Департамент информатики

Основная профессиональная образовательная программа
«Прикладная математика и информатика»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

на тему

Определение авторства разработчиков на основании стиля написания кода

Выполнил студент группы БПМ151, 4 курса,

Богомолов Егор Олегович

Руководитель:

Кандидат физико-математических наук, доцент департамента информатики,

Булычев Дмитрий Юрьевич

Консультант:

Кандидат технических наук, доцент департамента информатики,

Брыксин Тимофей Александрович

Санкт-Петербург

2019

Оглавление

Введение

Актуальность и релевантные работы

Задача определения авторства возникает в разных областях на протяжении многих лет. Работы неизвестных или пожелавших остаться анонимными авторов существуют в литературе, живописи, музыке, и программирование не является исключением. Необходимость определить разработчика, стоящего за кодом, возникает при поиске авторов вредоносного программного обеспечения, решении вопросов интеллектуальной собственности.

Подходы к решению задачи определения авторства основываются на предположении, что каждый автор обладает уникальный стилем, точно так же как уникальным почерком или отпечатком пальца. Идея определения набора характеристик или метрик кода, которые отражают стиль его автора, интересует исследователей не менее трех десятилетий [1, 2]. Статистический анализ стилистики кода, текстов или живописи называется стилометрией. Он основывается на подсчете метрик или факторов по исследуемому материалу и дальнейшем анализе полученных численных представлений.

Предыдущие исследования в области определения авторства кода были мотивированы проверкой авторства [3, 4], поиском плагиата [5, 6], определением авторов вредоносных программ [7, 8]. Для использования автоматических методов во всех перечисленных задачах требуется хорошее тестирование. Для этого необходим набор данных, на котором оно будет производиться. В существующих работах в качестве датасетов применялись студенческие домашние задания [9, 3, 10], примеры кода из книг [2, 11], задачи соревнований по программированию [12, 8, 13, 14] и проекты с открытым исходным, написанные одним автором [15, 4, 16, 17]. У всех перечисленных типов датасетов есть свои недостатки: они отличаются от промышленного кода и у них есть ограничение на количество примеров, доступных для каждого автора.

В предыдущих работах не изучалось масштабирование решений на

случай, когда для каждого автора доступны не десятки или сотни примеров, а десятки тысяч. Это связано с ограничениями имеющихся датасетов: такие объемы доступны только в крупных промышленных проектах, а в них одновременно работает большое количество программистов и сложно разделить код между ними, чтобы собрать примеры, для которых автор известен. Помимо этого, лучшие модели на данный момент для определения авторства по коду на языках C++ и Java используют факторы, специфичные для конкретных языков, что усложняет использование предложенных методов для других языков.

Цель и задачи

Данная работа ставит целью создание универсального относительно языка решения, которое улучшает или повторяет существующие результаты в определении авторства для C++, Java и Python, проверка его масштабируемости на разные объемы данных, доступных для каждого автора, сравнение с имеющимися исследованиями. Для этого ставятся следующие задачи:

- Научиться собирать данные для обучения и тестирования моделей, определяющих авторство кода, из проектов с произвольным количеством авторов.
- Создать набор датасетов для анализа применимости моделей в разных условиях: при разном количестве данных, несбалансированном количестве примеров, доступном для каждого автора, разделяя примеры в обучающей и тестовой выборке по времени.
- Создать модель, позволяющую работать одинаковым образом с кодом на различных языках и адаптировать её для работы с произвольным объемом доступных данных.
- Протестировать модель на собранных датасетах и сравнить с существующими решениями, используя данные, на которых они тестировались.

Достигнутые результаты

В рамках данной работы был реализован инструмент для сбора данных из проектов с произвольным числом автором, которые разрабатывались с использованием системы контроля версия (VCS, version control system) git. Для этого история проекта разбивается на отдельные изменения, для каждого из которых известен автор. Эти изменения в свою очередь разбиваются на изменения методов или классов, которые и составляют итоговый датасет. Такой подход снимает ограничения, которые были у предыдущих наборов данных: есть возможность использовать промышленный код, нет ограничения в выборе проектов и количестве доступных примеров.

При помощи инструмента были собраны данные обо всех изменениях методов в IntelliJ IDEA[1], втором по размеру проекте на GitHub с открытым исходным кодом на языке Java. Это 700 тысяч созданных методов и 2 миллиона изменений, принадлежащих 500 разработчикам. Из них было составлено 5 датасетов, предназначенных для тестирования моделей в разных условиях (подробнее в главе 3).

Было реализовано две модели: для работы в условиях небольшого количества примеров (от нескольких до сотен), доступных для каждого разработчика и в условиях большого (десятки и сотни тысяч). Обе модели основываются на технике представления кода при помощи контекстов в абстрактном синтаксическом дереве (AST, abstract syntax tree) [18]. Использование AST для определения авторства по коду уже позволяло улучшать результаты в нескольких недавних работах [8, 13].

Полученные модели были протестированы на наборах данных для C++, Java и Python, результаты тестирования на которых приводили исследователи в недавних работах. Полученная точность в случае Python и Java оказалась лучше, чем у предыдущих решений и повторила предыдущие результаты в случае C++ (подробнее в главе 6). Для датасетов, собранных на основе IntelliJ IDEA, результаты не сравниваются с предыдущими решениями из-за отсутствия их реализации в

[1]https://github.com/jetbrains/intellij-community

открытом доступе или того, что имеющиеся реализации не поддерживают Java.

Структура работы

В главе 1 представлен обзор предыдущих исследований в области, история развития подходов к определению авторства с точки зрения данных, факторов и методов машинного обучения.

В главе 2 представлена предложенная техника сбора данных из проектов с произвольным числом авторов, реализованный инструмент для сбора данных и собранные датасеты.

В главе 3 приводятся технические детали и архитектура созданных моделей.

В главе 4 представлены численные результаты проделанной работы, результаты проведенных экспериментов и приведено исследование поведения моделей в разных условиях.

В последней главе анализируется проделанная работа и приводятся возможные перспективы дальнейшей деятельности.

1. Обзор литературы

1.1. История развития области

Первоначально задача установления авторства появилась в литературе не позднее, чем в XIX веке. Первые исследования, связанные с авторством программного кода, появились только в 1970-е [5] и основывались на теории программного обеспечения, разработанной Холстедом [1, 19]. Теория утверждала, что всего четырёх метрик и их комбинаций достаточно, чтобы передать внутренние свойства программы, в том числе, различия в коде, написанном разными авторами. Метрики, названные метриками Холстеда, приведены в таблице 1.1.

В дальнейшем было показано, что предсказания теории расходятся с эмпирическими данными [20, 21], но, несмотря на это, она дала начало исследованиям по поиску сходств в программах. Первоначально они были направлены на автоматическое определение списывания в домашних заданиях.

В дальнейшем теорию активно развивали: был предложен алгоритм для быстрого вычисления метрик по коду [22], показана связь между размером проекта, временем, потраченным на его разработку и метриками Холстеда [23]. Фитцсиммонс [24] продемонстрировала, что при достаточно большом количестве программ теория программного обеспечения даёт точные оценки на число ошибок, допущенных в коде. Несмотря на поддержку теории частью учёных, подвергались критике её расхождения с эмпирическими данными [20, 21].

Таблица 1.1: Метрики Холстеда [1].

Метрика	**Определение**
n1	Число различных операторов
n2	Число различных операндов
N1	Суммарное число операторов
N1	Суммарное число операндов

Задача определения авторства исходного кода впервые привлекла

внимание исследователей в 1989 году. Оман и Кук [2] показали, что авторов программ можно различить с хорошей точностью, основываясь на статистическом анализе метрик форматирования. К ним относится расстановка переносов строк, отступов, длины названий переменных и функций, типы комментариев. Авторы не учитывали, что часть из этих метрик легко изменить при помощи текстовых редакторов или сред разработки.

Затем, Спэффорд и др. [25] впервые заговорили о применении автоматического определения авторства кода для решения вопросов интеллектуальной собственности. Авторы утверждали, что у каждого программиста есть свой уникальный стиль, по которому его можно определить, точно так же, как это делается по почерку в судебно-медицинской экспертизе. Несмотря на отсутствие эмпирической проверки этих утверждений, в работе было приведено большое количество метрик или факторов, по которым можно определять стиль программиста: использование библиотек, возможностей языка, допускаемые ошибки, стиль форматирования и комментариев, названия переменных и ошибки в них. В дальнейшем они использовались во многих работах по установлению авторства [26, 27, 28, 4, 8, 16].

Определение авторства в литературе и программировании являются похожими по формулировке задачами, но между областями есть существенные различия. В работе [10] авторы пришли к выводу, что определить автора фрагмента кода сложнее, чем литературного текста. Это вызвано тем, что разработчики переиспользуют код, работают в командах, используют среды разработки, автоматически форматирующие код, используют сторонние библиотеки и инструменты. В работе [29] было показано, что несмотря на вышеописанные различия, методы из области обработки естественного языка (ОЕЯ или NLP, Natural Language Processing) могут давать хорошие результаты, если будут использованы вместе с метриками программного обеспечения.

Последующие работы [10, 30, 26] сфокусировались на расширении множества используемых факторов, в том числе добавляя факторы, применимые для конкретного языка программирования, например, чис-

ло директив препроцессора в C++ [26]. В 2004 году Динг и др. [27] применили отбор факторов для определения авторства кода на Java. Они использовали набор метрик из предыдущих работ и отфильтровали его при помощи пошагового дискриминантного анализа и дисперсионного анализа. Оба способа показали улучшение точности с 62% до 85% по сравнению с использованием всех метрик.

В то время как вышеописанные исследования работали с метриками, которые понятны человеку и на основе которых люди могут считать код похожим, в работе Франческу и др. [11, 9] были использованы N-граммы, полученные по коду. N-граммы — это последовательности из N идущих подряд символов, например, 4-граммы строки "СТРОКА" это "СТРО", "ТРОК", "РОКА". Авторы построили профиль для каждого автора, состоящий из нескольких тысяч наиболее часто встречающихся N-грамм в его документах. Определение автора нового документа состояло из построения по нему набора N-грамм такого же размера и нахождения самого близкого профиля по размеру пересечения с набором. Такой подход показал высокую точность при различении как студенческих заданий на Java, так и проектов с открытым исходным кодом, при этом будучи простым с точки зрения вычислений и независимым от языка программирования.

Из-за простоты и эффективности анализа N-грамм он использовался во многих последующих работах [31, 32, 7]. Бёрроуз и др. [31] исследовали применение N-грамм к изменённой версии кода, в которой оставлены только операторы и ключевые слова. Кришна и др. [32] использовали подход для поиска компьютерных вирусов. Лейтон и др. [7] определяли с его помощью авторов фишинговых программ.

Общей проблемой анализа N-грамм для определения авторства является выбор N. В разных работах, в зависимости от данных и метрики оценки, оптимальные значения варьируются от 4 [4] до 20 [31]. Второй проблемой оказалось падение точности при увеличении числа различаемых программистов [31]. Последующие работы переключились с анализа синтаксического сходства кода на анализ его семантики [33, 8, 34].

К 2007 году сформировался общий подход к решению задачи опре-

деления авторства, подробно описанный в работе Котари [4]. Он имеет следующую структуру: сперва собирается набор фрагментов кода, для которых известны авторы. Фрагментами могут быть как целые файлы, так и отдельные классы или методы. Собранный набор данных разделяется на обучающую или тренировочную и тестовую выборки. По каждому фрагменту в обучающей выборке считается заданный набор метрик (например, частоты N-грамм [11] или метрики, специфичные для языка [26]), называемый численным представлением фрагмента. Затем выбирается модель для решения задачи и обучается на тренировочных представлениях. Она может агрегировать информацию для отдельных авторов в профили или же пользоваться ей в явном виде. Качество обученной модели проверяется на тестовой выборке.

1.2. Выбор данных

Получаемые результаты зависят от того, какие данные были собраны. Чтобы можно было утверждать о применимости модели на практике, она должна быть протестирована на данных, близких к реальным. В исследованиях 90-х годов [10, 35] в качестве датасетов использовались домашние задания студентов. Это вызывало проблемы с распространением данных, поскольку их требовалось анонимизировать, удалять комментарии.

Помимо домашних заданий в работах использовались примеры кода из книг по программированию и код, представленный в качестве примера авторами компиляторов [26, 2]. В случае таких фрагментов считалось, что весь код в одной книге или одном компиляторе написан одним человеком, что не обязательно является правдой.

С ростом популярности программ с открытым исходным кодом и сервисов для их загрузки исследователи стали использовать их в качестве датасетов. Так, использовались фрагменты кода, скачанные на freshmeat[2] [11], SourceForge[3] [28, 15], planet-source-code[4] [36]. Такие дан-

[2]http://freshmeat.sourceforge.net/

[3]http://sourceforge.net/

[4]https://www.planet-source-code.com/

ные ближе к промышленному коду, чем домашние задания или код из книг, но при этом фрагменты разных авторов могут относиться к проектам разной тематики. Это вызывает у исследователей опасение, что итоговое решение учится различать тематику проекта, а не его автора [8].

Розенблум и др. [12] стали первыми, кто в качестве датасета использовались решения участников Google Code Jam (GCJ). GCJ — ежегодное соревнование по программированию, проводимое компанией Google с 2008 года. Соревнование состоит из нескольких раундов, в каждом из которых участникам предлагается набор из нескольких алгоритмических задач, которые требуется решить за ограниченное время. Хотя решения принимается на большинстве популярных языков программирования, чаще всего используются C++, C и Python. Все корректные решения, отправленные во время соревнования, доступны для скачивания и активно используются исследователями, занимающимися задачей определения авторства [12, 8, 13, 14]. Это вызвано тем, что все участники решают одни и те же задачи, а значит их код должен выполнять одну функцию, различаясь только индивидуальными особенностями авторов. Тем не менее, использование данных GCJ подвергается критике [37] за свою искусственность: код в решениях алгоритмических задач может сильно отличаться от промышленного кода.

Помимо нового подхода к выбору данных, в работах Розенблума и др. [38, 12, 39] был предложен новый способ определения автора по бинарным файлам и возможность восстановить набор инструментов, которым бинарный файл был получен: компилятор, его версию и уровень оптимизации. В этой серии работ была предложена методика получению факторов по бинарному коду. Для этого сперва из бинарного файла извлекается код отдельных функций или логических блоков и граф потока управления. Затем в коде ищутся идиомы — последовательности из 1-3 инструкций, содержащие низкоуровневую информацию о логике кода — наличие которых является фактором. В графе потока управления ищутся графлеты — подграфы из 3 вершин, которые отражают локальную структуру программы. Количества найденных граф-

летов каждого типа является фактором. В итоге авторы получили точность 51% при определении автора бинарного файла, выбирая из 191 автора.

1.3. Выбор методов

Помимо развития в выборе данных и метрик происходило развитие в выборе методов. Для определения авторства исходного кода применялись разные методы машинного обучения.

Часть работ использовали подходы, основывающиеся на поиске похожих фрагментов: метод ближайших соседей [40, 28, 11], ранжирование [31, 15]. Суть этих подходов заключается в подсчёте расстояния между численными представлениями кода. Автором фрагмента кода считается автор большинства близких к нему фрагментов обучающей выборке. Расстоянием может выступать расстояние в векторном пространстве [15], пересечение гистограмм [28] или специально подобранная метрика, например, пересечение наборов N-грамм [11, 9]. В работе Бёрроуза и др. [31], исследовавшей определение авторства на основе ранжирования, сравнивалась эффективность четырёх расстояний: косинусной меры, языковой модели со сглаживанием Дирихле, BM25 и предложенной авторами метрики. Лучшие результаты показало использование BM25, что согласуется с результатами в области информационного поиска [41].

Также исследователи применяли вероятностные методы: байесовский классификатор [4] и логистическую регрессию [36]. Суть этих методов заключается в оценке вероятности того, что фрагмент кода принадлежит автору, при условии того, что в нём встречаются определённые N-граммы символов или токенов. Несмотря на хорошие результаты обеих работ относительно других исследований своего времени, в случае вероятностных подходов сложнее расширять множество факторов, поэтому их используют реже в пользу других методов.

В недавних работах популярностью пользуются методы, работающие с набором факторов как с векторами. К таким методам отно-

сится дискриминантный анализ [27, 42, 43] и метод опорных векторов [12, 44, 45]. Сперва по фрагментам кода подсчитывается набор из n метрик, образуя n-мерный вектор численного представления фрагмента. Примеры кода разных авторов образуют множества из векторов в n-мерном пространстве, и, если подобран качественный набор метрик, множества можно различить. Дальнейшую задачу дискриминантный анализ и метод опорных векторов решают по-разному.

Дискриминантный анализ трактует векторы фрагментов кода как примеры, полученные из разных случайных распределений. Задача определения авторства в таком случае сводится к подбору параметров распределений, которые наилучшим образом объясняют примеры из обучающей выборки, а для примеров из тестовой выборки в качестве предсказания берется автор, к распределению которого вектор кода относится с наибольшей вероятностью.

Метод опорных векторов основан на поиске поверхности, которая наилучшим образом отделяет в пространстве векторы одного класса от всех остальных. В самом простом случае поверхностью является плоскость, в более сложных применяется метод опорных векторов с ядром, что позволяет искать более сложные поверхности. Для применения метода к новым примерам вычисляется их положение относительно разделяющих поверхностей.

Ещё одним методом, использовавшимся для решения задачи определения авторства, является решающее дерево. Оно применялось в работе Эленбогена и др. [3]. Для построения дерева авторы применяли алгоритм C4.5 [46]. Это жадный алгоритм, который строит дерево от корня к листьям, в каждой вершине выбирая фактор и его значение такое, что оно наилучшим образом разбивает часть выборки, дошедшую до вершины, на две части. Затем, алгоритм рекурсивно применяется в двух новых вершинах. Когда дерево построено, чтобы уменьшить переобучение, в нём обрезается часть веток.

Идею решающего дерева развивает случайный лес. Вместо построения одного решающего дерева, используя все доступные данные и факторы, строится от нескольких до сотен или даже тысяч деревьев. Каж-

дое из них строится по случайной части обучающей выборки, используя случайный поднабор факторов. При определении ответа каждое полученное дерево принимает решение и решение большинства принимается за ответ. Случайный лес использовали в нескольких недавних работах по установлению авторства [8] и имитации стиля [14].

В исследовании по определению авторства Калискан и др. [8] продолжили работу с данными GCJ. В то время как большинство предыдущих исследований работали с кодом как с обычным текстом, отличающимся только форматированием и наличием специальных слов, они использовали факторы, полученные по абстрактному синтаксическому дереву (AST, Abstract Syntax Tree), соответствующему коду. Такие факторы называют синтаксическими, к ним в данной работе относились частоты встречаемости и средние глубины отдельных вершин разных типов и их пар, соединённых ребром. Использование AST позволяет передать внутреннюю структуру кода, которая в свою очередь отвечает за его функциональность. Помимо синтаксических также в работе применялся большой набор метрик форматирования и лексических, что породило около 100 тысяч факторов. Их количество затем было сокращено до 1000 наиболее полезных, при этом критерием отбора была совместная информация между значением фактора и автором. Описанный набор метрик затем использовался другими исследователями [16].

По полученному после подсчёта и фильтрации факторов численному представлению Калискан и др. обучили случайный лес из 300 деревьев. В результате они получили решение, которое хорошо масштабировалось на данных GCJ: 98% точности при различении кода 250 авторов на C++, 92% при различении 1600. Стоит отметить, что работ со сравнением такого количества программистов ранее не проводилось и результат остаётся лучшим для данных GCJ на языке C++. Препятствием к дальнейшему увеличению масштаба стали ограничения датасета: только у 1600 участников было 9 решений по одним и тем же задачам.

С развитием методов, использующих нейронные сети в работе с

изображениями [47], обработке естественного языка [48] и распознавании речи [49], сети стали применяться и в определении авторства. Нейронная сеть или нейросеть это математическая модель, построенная по принципу организации нейронов в мозгу человека. Нейросеть состоит из элементов, соединенных направленными связями, по которым данные передаются от входных нейронов к следующим элементам, вплоть до выходных нейронов. Элементы применяют к полученным на вход данным различные математические операции: линейную комбинацию, применение нелинейной функции, свертку, сдвиг и так далее. Набор элементов и связей между ними описывается архитектурой нейросети. В области обработки естественного языка для решения задач активно применяются рекуррентные нейронные сети [50] и, в частности, LSTM [51].

Идея работы с AST при помощи нейронных сетей была продолжена Алсулами и др. [13]. Вместо дальнейшего расширения набора факторов, они применили модификацию LSTM для работы с деревьями [52]. Она работает следующим образом: каждому типу вершины в AST и токену сопоставляется вектор, называющийся эмбеддинг, изначально случайный, но изменяющийся в процессе обучения сети. Дерево обходится в глубину, в листьях сеть получает на вход эмбеддинг токена, относящегося к листу, в остальных вершинах — векторы, полученные в детях, и эмбеддинг типа текущей вершины. В результате в корне дерева сеть генерирует вектор, являющийся представлением всего дерева. По нему ещё одним линейным слоем с функцией активации softmax делается предсказание. Сеть обучается при помощи стохастического градиентного спуска и алгоритма обратного распространения ошибки.

Тестирование проводилось для языков Python и C++. Для Python были использованы решения по 10 задачам 70 участников GCJ. Для C++ использовались проекты 10 авторов, размещённые на GitHub. Модель достигла точности 88.8% при распознавании кода на Python 70 авторов, что является лучшим из известных результатов на данный момент.

В работах по установлению авторства кода использовали не только

разные архитектуры сетей, но и методы оптимизации для их обучения. В 2017 году Янг и др. [16] применили набор факторов из работы Калискан и др. [8], объединив некоторые из них в более общие, и обучили трёхслойную нейросеть для определения автора по ним. Они сравнили два метода обучения: стохастический градиентный спуск (SGD, stochastic gradient decent) и метод роя частиц (PSO, particle swarm optimisation) [53]. Сравнение производилось на датасете из 40 проектов на Java, каждый из которых был написан одним автором. Результаты при использовании PSO оказались значительно лучше, чем при стохастическом градиентном спуске, который обычно используется при обучении нейросетей: 91% против 76%. Результат является наилучшим на данный момент для этого набора данных.

Изменение методов машинного обучения, использовавшихся для решения задачи определения авторства, происходило одновременно с эволюцией методов для решения других задач в области машинного обучения на исходном коде. К ним относится генерация кода, определение тематики, генерация описания по фрагменту кода, определение имени метода.

Недавний прогресс в определении имени метода и генерации описания кода связан с техникой векторного представления кода, предложенной Алоном и др. в 2018 году [18, 54]. Её можно рассматривать как обобщение использования биграмм типов вершин в AST и токенов, предложенных в работе Калискан и др. [8]. По фрагменту кода строится AST и из него извлекаются тройки из пути между парой листьев и токенов, соответствующим им. Такие тройки называются контекстами. Фрагмент представляется набором контекстов, ему соответствующим. Контексту сопоставляется вектор, состоящий из конкатенации эмбеддингов токенов и пути. Затем векторы контекстов агрегируются при помощи механизма внимания [55]. Это механизм, позволивший улучшить результаты в задачах обработки естественного языка [48]. Вектором для фрагмента является взвешенная сумма векторов контекстов, где весами являются значения внимания.

1.4. Выводы

- Для трёх разных языков, Java, C++ и Python, наилучшие результаты показывают разные работы. Решения для Java [16] и C++ [8] используют факторы, специфичные для конкретного языка, что затрудняет их адаптацию к другим. Решение для Python [13] использует только AST, поэтому является универсальным, при этом показывая очень высокую точность. Это позволяет надеяться на то, что используя AST можно получить универсальное решение, показывающее хорошие результаты сразу для нескольких языков.

- В исследованиях не рассматривалась работа с большим количеством данных для каждого автора. Это актуально в задачах проверки авторства, кластеризации и профилировании разработчиков. Ограничением для подобных исследований является отсутствие датасета соответствующего размера.

- Датасеты в недавних работах состоят из задач олимпиадного программирования [12, 8, 13, 14] или из проектов с одним автором [17, 13, 44, 36, 16]. Решения задач на соревнованиях значительно отличаются от промышленного кода: участникам требуется написать решение максимально быстро, что может быть сделано в ущерб читаемости и архитектуре. Чаще всего решение состоит из одного файла, где большая часть кода находится в одной функции. В дополнение к отличиям от промышленного кода, количество доступных данных значительно отличается от обычных проектов, поскольку программисты гораздо реже участвуют в соревнованиях, чем пишут код на работе. В датасетах из проектов, написанных разными людьми, присутствует разница в предметной области между кодом разных авторов. Решение, которое показывает высокую точность на подобных датасетах, может показывать другие результаты применительно к коду разных авторов в одной предметной области, что важно для поиска авторов вредоносных программ. Проблемы обоих видов датасетов можно решить, ес-

ли собирать данные из крупных проектов с большим количеством авторов.

2. Данные

Чтобы обучить модель машинного обучения и протестировать её качество, нужен датасет. У имеющихся датасетов есть слабые стороны, которые указывались в работах других исследователей. В этой главе предлагается новый подход к сбору данных для оценки качества решений задачи определения авторства, описывается реализованный инструмент для создания датасетов, использующий этот подход и датасеты, собранные с его помощью.

2.1. Ограничения существующих датасетов

Прошлые исследования в области определения авторства исходного кода работали с наборами данных нескольких видов: домашние задания студентов [9, 3, 10], примеры кода в книгах по программированию [2, 11], решения задач соревнований по программированию [12, 8, 13, 14], проекты с открытым исходным кодом, имеющие одного автора [15, 4, 16, 17]. Первые три типа данных значительно отличаются от промышленного кода: студенты имеют меньше опыта, чем профессиональные разработчики, примеры в книгах – это короткие и неполные фрагменты, на соревнованиях участникам важно написать программу быстро, не задумываясь о читаемости и качестве написанного. Отличия не позволяют однозначно утверждать, что решение, демонстрирующее хорошую точность на таких датасетах, будет так же хорошо работать для промышленного кода, что важно при его применении для решения вопросов интеллектуальной собственности.

Существующие исследования работают не с произвольными проектами с открытым исходным кодом, а только с имеющими одного автора или такими, где хотя бы 90% кода принадлежит одному разработчику. В датасетах, состоящих из таких проектов, код разных авторов соответствует разным проектам, а значит может иметь разную тематику и функциональность. Из-за этого нельзя быть уверенными в том, что полученное решение будет хорошо работать, если будет применяться для различения кода разработчиков, работающих в одной области или

реализующих схожую функциональность, хотя это важно при поиске авторов вредоносного ПО, поиске плагиата, проверке авторства.

Также, у всех четырёх типов данных есть ограничение на количество фрагментов кода, доступных для каждого автора: ограниченный набор задач на соревнованиях и в домашних работах, требуемое количество поясняющих примеров в книгах. Проекты с одним автором также редко состоят из более чем нескольких сотен файлов, поскольку при дальнейшем росте проекта у него появляются пользователи, которые начинают самостоятельно предлагать изменения и вносить свой вклад в его развитие. В то же время, профессиональные программисты пишут гораздо большие объемы кода. Например, в проекте IntelliJ IDEA есть 40 разработчиков, каждый из которых добавил не менее 10000 методов.

2.2. Методика сбора данных

В этой работе предлагается метод сбора данных из проектов с произвольным числом разработчиков, разрабатываемых с использованием системы контроля версий (VCS, version control system). Необходимость в VCS не является сильным ограничением, поскольку в настоящее время это общепринятая практика. Самой популярной платформой для загрузки программ с открытым кодом является GitHub[5], поддерживающая VCS git. По состоянию на конец 2018 года, платформой пользуется более 30 миллионов разработчиков, создавших 96 миллионов репозиториев [56]. Датасеты, состоящие из проектов на GitHub, уже использовались в работах по определению авторства [44, 16, 13, 17].

В проектах, использующих VCS, изменения добавляются группами, которые называются коммитами. Коммит чаще всего принадлежит одному автору и состоит из набора изменений, которые могут затрагивать произвольное число файлов. В таблице 2.1 приведены 10 репозиториев на GitHub с самым большим числом коммитов. Каждый коммит можно разбить на изменения, затрагивающие части кода определенного уровня: изменения методов, классов, файлов. Из таких изменений в свою

[5]https://github.com/

очередь можно составить датасет, поскольку для каждого из них автор известен благодаря VCS. В этой работе рассматриваются изменения отдельных методов.

Таблица 2.1: Репозитории на GitHub с самым большим числом коммитов

Репозиторий	**Коммиты (тыс.)**	**Основной язык**
torvalds/linux	782	C
LibreOffice/core	428	C++
liferay/liferay-portal	283	Java
jsonn/pkgsrc	266	Makefile
freebsd/freebsd	254	C
JetBrains/intellij-community	230	Java
cms-sw/cmssw	194	C++
openbsd/src	192	C
NixOS/nixpkgs	154	Nix
Wikia/app	152	PHP

Использование такого подхода затруднено тем, что git, являющийся самой популярной VCS, не хранит подробную информацию об изменениях. Это означает, что чтобы получить информацию об изменении метода, нужно найти его в старой и новой версии файла, что может быть проблематичным в случае переименования или изменения сигнатуры. Для решения этой проблемы была использована библиотека GumTree [57], позволяющая строить AST по коду на языке Java и искать по версиям файла до и после коммита изменившиеся методы и соответствующие им поддеревья в обеих версиях.

Для сбора данных была написана библиотека на двух языках, Python и Kotlin. Часть, написанная на Python, обрабатывает историю проекта, отслеживая в ней изменения отдельных файлов с кодом на Java, и сохраняет информацию о них в бинарном виде. Часть, написанная на Kotlin, проходит по истории изменения Java-файлов и для каждого файла строит AST и определяет изменившиеся методы при помощи библиотеки GumTree [57]. Информация об AST изменившихся методов сохраняется, образуя данные для построения датасетов.

2.3. Собранные датасеты

С помощью инструмента из истории проекта IntelliJ IDEA были извлечены все изменения отдельных методов. Это 2 миллиона изменений, принадлежащих 500 разработчикам. Из них 98% сделаны 100 самыми активными разработчиками, а 90% — 50 самыми активными. 700 тысяч изменений являются добавлениями метода и не требуют дополнительной адаптации алгоритмов определения авторства, умеющих работать с фрагментами кода.

Из изменений затем было создано 7 датасетов. Во всех датасетах выборка заранее разделена на тренировочную и обучающую в отношении 70%/30%, при этом в тренировочной и тестовой выборке изменения методов относятся к разным файлам. Датасеты 1-4 моделируют условия разного количества разработчиков, количества доступных примеров и разной степени несбалансированности между классами. Степень сбалансированности выражается отношением максимального числа фрагментов кода среди программистов, попавших в датасет, к минимальному. Подробная информация о датасетах 1-4 приведена в таблице 2.2.

В датасетах 2-4 требуется предсказать автора не для одного метода, а для группы из нескольких, выбранных случайным образом из примеров, входящих в тренировочную или тестовую выборку. Такая постановка задачи заставляет решение поддерживать возможность работы с несколькими фрагментами кода, что может встречаться при практическом применении к задачам проверки авторства, поиске авторов вредоносных программ и анализе стиля кода.

Таблица 2.2: Датасеты с разной сбалансированностью и числом разработчиков, построенные по проекту IntelliJ IDEA

	IDEA1	**IDEA2**	**IDEA3**	**IDEA4**
Число авторов	10	16	50	20
Число примеров (тыс.)	912	648	1623	1228
Макс. / Мин. примеров	5	3	32	8.5
Примеров в группе	1	16	16	16
Деление по пакетам	0	0	0	0

Основным отличием наборов данных получаемых из одного проекта по сравнению с наборами, состоящими из проектов написанных разными авторами, является меньшая разница в тематике и функциональности кода разных разработчиков, которую мы называем рабочим контекстом. Хотя внутри одного проекта также существует разделение по областям ответственности между разработчиками, оно менее значительно. Чтобы сократить его ещё больше, были созданы датасеты 4, 5 и 6, в которых тренировочная и тестовая выборки разделены по пакетам разного уровня. Нулевой уровень соответствует тому, что методы в тренировочной и обучающей выборке относятся к разным файлам, это верно для всех собранных датасетов. Первый уровень запрещает примерам в выборках находится в одной директории, а второй уровень — лежать в одной над-директории, при этом проверка осуществляется по файлу, над-директория которого лежит менее глубоко. Таким образом, тренировочный и тестовый код каждого разработчика относятся к разным частям проекта, что позволяет определить, влияет ли рабочий контекст на результаты тестирования.

В датасете 7 тренировочная и тестовая выборки разбиты по времени. Эксперименты с разделением данных по времени ставились в работе Калискан и др. [8]. В их работе рассматривались 25 программистов, участвовавших в GCJ в 2012 и 2014 годах. Их решения 2012 года были тренировочной выборкой, а решения 2014 — тестовой. В обе выборки входило по 9 решений для каждого человека, ограничение вызвано количеством задач в соревновании. Метод, предложенный в этой работе, позволяет проводить исследования изменения стиля кода со временем в гораздо большем масштабе: для попавших в датасет программистов доступны десятки тысяч примеров. Более подробная информация о датасетах 5-7 приведена в таблице 2.3.

2.4. Выводы

В этой работе предлагается метод сбора данных для тестирования решений задачи определения авторства кода из проектов с произволь-

Таблица 2.3: Датасеты с разделением по времени и по пакетам разного уровня, построенные по проекту IntelliJ IDEA

	IDEA5	**IDEA6**	**IDEA7**
Число авторов	20	20	10
Число примеров (тыс.)	1228	1228	912
Макс. / Мин. примеров	8.5	8.5	5
Примеров в группе	16	16	16
Деление по пакетам	1	2	0
Деление по времени	-	-	+

ным числом авторов. Он позволяет составлять датасеты для тестирования моделей в различных условиях: при разделении выборки по времени, по рабочему контексту, для разного количества примеров, доступных для каждого разработчика. Предложенный метод реализован для работы с языком Java и протестирован на втором по величине Java-проекте на GitHub, IntelliJ IDEA. В результате получен набор из 7 датасетов, позволяющих проверку моделей для определения авторства в разных условиях.

3. Модель

Существующие решения, показывающие наилучшие результаты на опубликованных датасетах для Java [16] и C++ [8], используют факторы, специфичные для конкретных языков. В этой работе предлагается два решения, основывающиеся на представлении кода при помощи контекстов в AST [18], работающие с произвольным языком программирования. Модели предназначены для работы с разным количества примеров, доступных для каждого автора. Первая модель применима при числе примеров порядка тысячи и более, вторая — при числе примеров в пределах нескольких сотен.

3.1. Модель на основе code2vec

Схема работы метода векторизации кода на основе контекстов в AST представлена на Рис. 3.2a. В дереве выделяются листья и соответствующие им токены. Они нумеруются в порядке появления в коде. Затем из дерева выделяются тройки из пары токенов и пути между ними, называемые контекстами. Токены называются начальным и конечным в порядке их появления в коде. Путь представляет из себя последовательность типов вершин в порядке от начального листа до конечного и пометок о том, вверх или вниз совершен переход. На Рис. 3.1b приведен пример AST, построенного по фрагменту кода. Цветом выделен путь между двумя листьями. Этому пути соответствует следующая последовательность, где стрелками обозначается направление перехода между вершинами, а типы сокращаются по первым буквам:

$$SN \uparrow MD \downarrow B \downarrow RS \downarrow IE \downarrow SN$$

Контекст, образованный из пути и токенов на его концах выглядит следующим образом:

$$(square, SN \uparrow MD \downarrow B \downarrow RS \downarrow IE \downarrow SN, x)$$

Фрагмент кода представляется мультимножеством контекстов, со-

держащихся в его AST. Если фрагмент содержит N токенов, то размер мультимножества будет равен количеству пар токенов и асимптотически равен $O(N^2)$. Чтобы уменьшить их количество, вводится ограничение на разницу в индексах между начальным и конечным токеном, называющуюся шириной, и количество вершин в пути, называющееся длиной. Если ширина ограничена значением w, то контекстов останется $O(Nw)$.

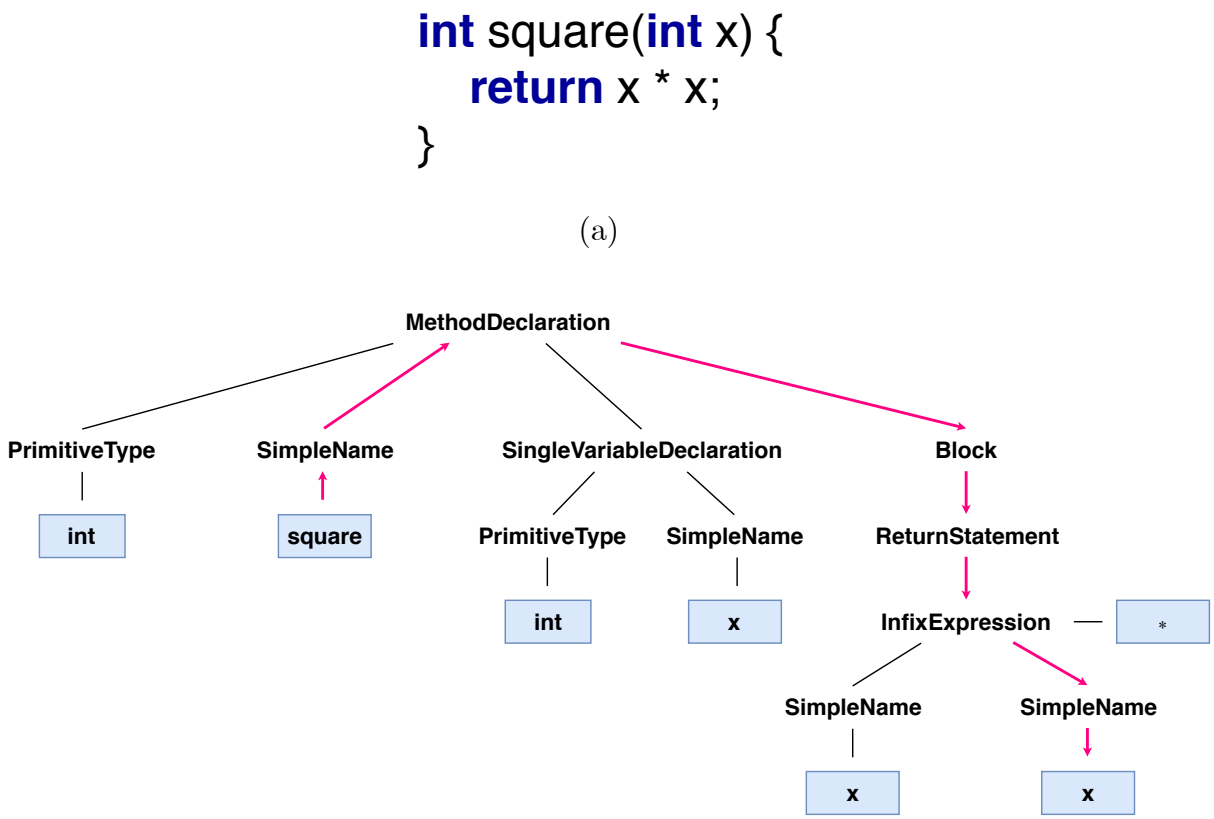


Рис. 3.1: (a) Пример фрагмента кода. (b) AST, построенное по фрагменту кода при помощи библиотеки GumTree [57]. Цветом выделен путь между парой листьев.

Каждому пути и токену сопоставляется вектор эмбеддинга. Эмбеддинг — это обучаемое отображение из конечного множества в векторное пространство. Изначально оно выбирается случайным, а в процессе обучения модели модифицируется. Контексту сопоставляется конкатенация из трёх векторов, соответствующих токенам и пути. Затем, чтобы получить вектор для фрагмента кода, векторизации его контекстов

складываются с весами, которые им назначает механизм внимания. Он реализуется при помощи скалярного произведения вектора контекста на вектор внимания, обучаемый вместе с моделью.

В данной работе модель была дополнена путём добавления возможности работать с группами из нескольких фрагментов кода. Ими могут быть методы, относящиеся к одному коммиту или находящиеся в одном файле, случайная выборка из всего множества методов, методы, написанные в определенный промежуток времени. Для этого в неё был добавлен ещё один слой с механизмом внимания, агрегирующий векторизации нескольких фрагментов в одну. Помимо увеличения точности модели он позволяет выделить фрагменты, имеющие большее значение при решении задачи, в нашем случае — при определении авторства группы из нескольких методов. Модифицированная архитектура представлена на Рис. 3.2c.

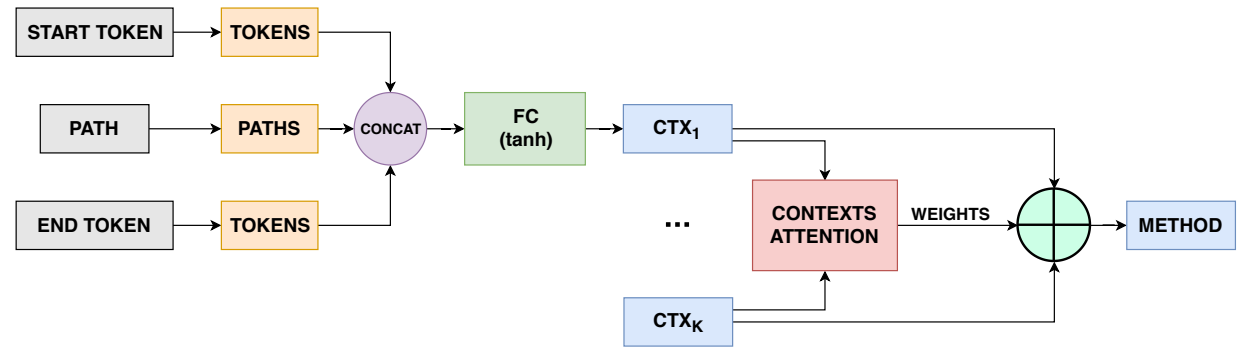
Количество параметров в предложенной модели зависит от числа различных путей и токенов и от размерности используемых векторных представлений. Обозначим число параметров за $ModelSize$ и запишем его асимптотику:

$$ModelSize = O(Td + Pd),$$

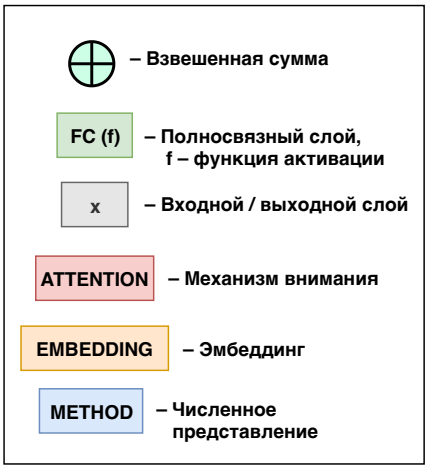
где T — число токенов, P — число путей, d — размерность векторизаций.

Число различных путей и токенов для датасетов, использованных в предыдущих работах, а также количество примеров, доступных для обучения, приведено в таблице 3.1. Количество параметров модели даже при выборе значения размерности равного 1 остается на порядок выше, чем число примеров, поэтому для наборов данных с высоким соотношением числа различных токенов и путей к размеру датасета предлагается вторая модель.

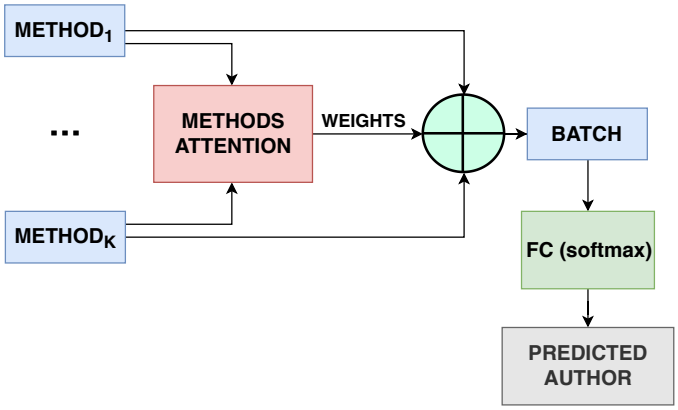
Модель может применяться для любого языка программирования, по коду которого можно построить AST. Таковыми является большинство популярных языков программирования, так как в них AST используется компиляторами и интерпретаторами для промежуточного представления кода. Для построения AST и дальнейшего получения



(a)



(b)



(c)

Рис. 3.2: (a) Архитектура модели code2vec [54]. (b) Условные обозначения для описания нейросетевой архитектуры. (c) Расширение архитектуры code2vec для работы с группами из нескольких методов.

контекстов используется библиотека, более подробно описанная в разделе 2.

3.2. Модель на основе случайного леса

Вторая модель представляет из себя случайный лес. В качестве факторов используются те же данные, которые образовывали контексты, описанные выше: частоты встречаемости путей между листьями в AST и частоты встречаемости токенов на их концах. Таким образом, каждый фрагмент кода представляется разреженным вектором размера $P+2T$. Для работы модели, как и в предыдущем случае, требуется построение AST, что возможно для большинства современных языков программи-

Таблица 3.1: Размеры датасетов, использовавшихся в предыдущих работах и число различных токенов и путей, встречающихся в них. Приведено число различных путей с шириной не более 3 и длиной не более 10.

	C++	**Java**	**Python**
Число авторов	1600	40	70
Размер датасета (файлов)	14400	3021	700
Различные токены (тыс.)	30.2	36.7	4.3
Различные пути (тыс.)	169.9	7.9	46.3
Токены и пути / Размер	13.9	14.8	72.3

рования.

Для улучшения качества работы модели применяется фильтрация множества факторов при помощи подсчета совместной информации между значением фактора и автором фрагмента. Совместная информация двух случайных величин определяется следующим образом:

$$I(A; F) = H(A) + H(F) - H(A, F),$$

где A — множество авторов, F — множество факторов, $H(X)$, $H(Y)$, $H(X, Y)$ — энтропия и условная энтропия случайных величин.

В результате фильтрации остаются только факторы с высоким значением совместной информации с целевой функцией. Доля факторов, которая остается после фильтрации, определяется эмпирически. Полученные эмпирические значения, результаты применения модели на различных датасетах, сравнение моделей между собой и с существующими решениями приведено в главе 4.

У модели есть два гиперпараметра: число деревьев в лесу и количество факторов, остающихся после фильтрации. Для их подбора применяется кросс-валидация и поиск по сетке. Качество работы модели оценивается путем кросс-валидации, перебирая разные значения гиперпараметров, и эмпирически устанавливаются оптимальные значения. Они варьируются в зависимости от набора данных: оптимальная доля остающихся после фильтрации факторов составляет от 5% до 10%, а оптимальное число деревьев — от 300 до 500.

3.3. Выводы

Эта работа предлагает две модели для определения авторства в условиях разного количества данных. В обоих случаях по фрагменту кода строится AST, а затем он представляется в виде набора контекстов, состоящих из путей между парами листьев в AST и токенами на их концах. Первая из моделей — нейронная сеть, использующая механизм внимания для получения векторного представления фрагмента кода из эмбеддингов путей и токенов. Ее целесообразно использовать, когда количество различных путей и токенов сопоставимо или меньше числа доступных для обучения примеров, на практике это тысячи или десятки тысяч примеров для каждого автора. Вторая модель — случайный лес из 500 деревьев, строящийся по частотам встречаемости токенов и путей, встречающихся в фрагменте кода. Для фильтрации множества факторов применяется метод фильтрации на основании совместной информации значения фактора и автора кода. Случайный лес целесообразно использовать, когда размер датасета меньше, чем число различных путей и токенов.

4. Тестирование

Для тестирования моделей, предложенных в главе 3, применялись датасеты для трех языков, использовавшиеся в недавних работах других исследователей: решения задач GCJ для Python [13] и C++ [8], набор из 40 проектов с открытым исходным кодом для Java [16]. Эти три датасета представляют собой наиболее актуальные результаты для каждого из языков среди работ, использовавших данные находящиеся в открытом доступе. Результаты сравнения приведены в таблице 4.1. Помимо тестирования на указанных датасетах были проведены эксперименты с данными проекта IntelliJ IDEA, более подробно описанными в главе 2.

4.1. Сравнение с предыдущими работами

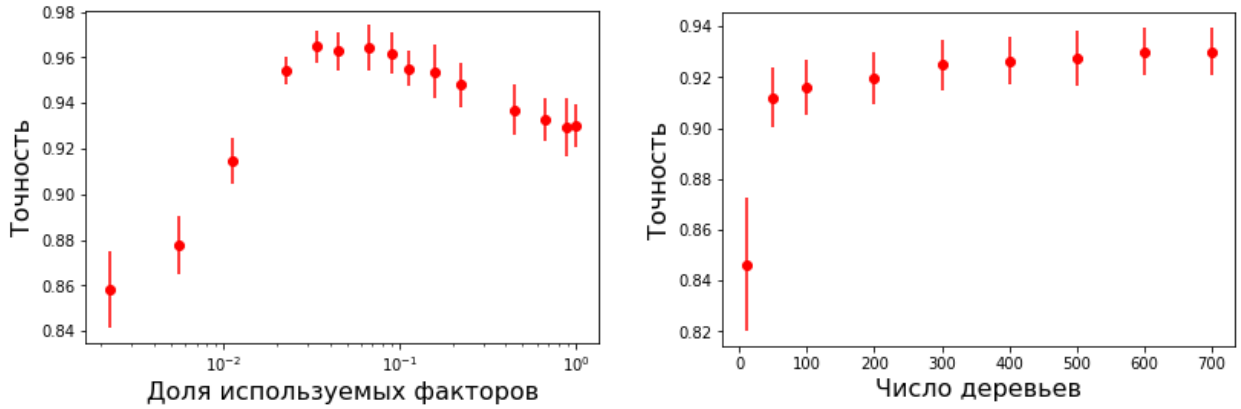
4.1.1. Java

Датасет для Java состоит из 40 проектов, написанных разными авторами. Он включает в себя 3021 файл, число файлов в проектах варьируется от 11 до 712 с медианой 54. Из-за значительного различия в числе доступных примеров между разработчиками датасет является несбалансированным. Поскольку в среднем число примеров не превышает сотни, из двух предложенных в этой работе моделей лучшие результаты показывает случайный лес, специально созданный для работы в условиях небольшого количества данных.

Сравнение производилось с лучшим на данный момент решением, использовавшим этот набор данных [16]. Для получения сравнимых результатов применяется такое же разбиение данных, как и в вышеуказанной работе. Все файлы разбиваются на 10 частей, или фолдов, в каждую из которых попадает одинаковое количество примеров от каждого из авторов. Например, от разработчика с 31 файлом в одной части окажется четыре файла, а в остальных по три. Затем проводится 10 экспериментов, в каждом из которых одна часть представляет тестовую выборку, а остальные — тренировочную. По выборке, состо-

ящей из результатов экспериментов, вычисляется выборочное среднее и дисперсия, которые затем сравниваются с результатами предыдущей работы.

Для применения случайного леса требуется подобрать оптимальные гиперпараметры, к которым относится число деревьев и количество используемых факторов. Для подбора оптимальных гиперпараметров была проведена серия экспериментов с разным количеством деревьев и разным количеством факторов. Средняя точность и отклонение, полученные в результате кроссвалидации, в зависимости от варьируемых параметров приведены на Рис. 4.1b для числа деревьев и на Рис. 4.1a для числа факторов. Точность при оптимальном выборе параметров значительно превосходит результаты предыдущей работы [16]: 97% против 91.1%.



(a) (b)

Рис. 4.1: Зависмость точности и стандартного отклонения для датасета на языке Java от (a) доли используемых факторов; (b) числа деревьев.

Использование нейросетевой модели показало точность 86%, что хуже чем результаты случайного леса и нейросети, обученной при помощи PSO [16]. При этом результаты оказались выше, чем у той же нейросети, обученной при помощи SGD. Это говорит о том, что даже в условиях маленького количества данных предложенная в данной работе нейросетевая модель может показывать хорошие результаты.

4.1.2. C++

Датасет для C++ состоит из решений задач GCJ. Он включает в себя решения по одним и тем же 9 задачам от 1600 участников. Датасет является сбалансированным, от каждого автора доступно очень мало данных, из-за чего вновь лучшие результаты показывает применение случайного леса. Для оценки качества модели используется кросс-валидация с разбиением на 9 частей, фолды представляют из себя решения по конкретной задаче. Для подбора гиперпараметров случайного леса использовался поиск по сетке.

Достигнутое значение средней точности по итогам кросс-валидации немного меньше, чем в предыдущей работе: 92.7% против 92.8%. Тем не менее, стандартное отклонение результатов при кросс-валидации составляет 0.8% и статистическая значимость различия между полученными результатами не ясна, так как авторы прошлой работы не приводят стандартного отклонения или значения точности для отдельных фолдов. Если допустить, что точность в предыдущей работе составляла 92.8% для всех разбиений, то разница в результатах оказывается статистически незначительной.

Нейросетевая модель достигла 41.5% точности, что значительно проигрывает случайному лесу и объясняется малым количеством доступных примеров.

4.1.3. Python

Датасет для Python, как и для C++, состоит из решений GCJ, но отличается размером. В нем использованы решения 10 задач от 70 участников. Для оценки точности модели применялась кросс-валидация, части разбиения состояли из пар задач. Датасет отличается от двух других еще более значительной разницей между числом доступных файлов и число различных путей и токенов (см. таблицу 3.1). Из-за этого для него не удалось получить содержательной оценки работы нейросетевой модели: при точности приближающейся к 100% на тренировочной выборке, на тестовой результаты не отличались от случайных.

Случайный лес с подобранными оптимальными параметрами показал улучшение точности по сравнению с предыдущими работами: 94% против 88.9%.

Таблица 4.1: Средняя точность определения авторства по итогам кросс-валидации на датасетах, использовавшихся в предыдущих работах.

	C++ 1600	**Python 70**	**Java 40**
Калискан и др. [8]	**92.8%**	72.9%	-
Алсулами и др. [13]	-	88.9%	-
Янг и др., SGD [16]	-	-	76%
Янг и др., PSO [16]	-	-	91.1%
Данная работа, нейросеть	41.5%	-	86%
Данная работа, случайный лес	**92.7%**	**94%**	**97%**

4.2. Тестирование на данных IntelliJ IDEA

Для датасетов, описанных в главе 2, производилось тестирование нейросетевой модели, результаты тестирования приведены в таблице 4.2. Модель на основе случайного леса была создана для работы с меньшими объемами данных и при запуске на числе примеров близком к миллиону она столкнулась с проблемами из-за большого потребления памяти и времени, требуемого для обучения.

На каждом из датасетов нейросетевая модель обучалась при помощи оптимизатора Adam [58] с шагом 0.01 на протяжении 10 эпох. Размерность векторизации после тестовых запусков была выбрана равной 8 для достижения баланса между пропускной способностью, сложностью модели и скоростью обучения. При этом пропускная способность была равна 1500 примерам в секунду при обучении и 2200 при тестировании.

Все 7 собранных датасетов в разной степени несбалансированные, из-за чего использование одной лишь точности для оценки качества оказывается недостаточным. Например, в датасете 3 95% изменений методов принадлежат 40 из 50 разработчиков. В таком случае модель,

которая полностью игнорирует 10 наименее активных разработчиков, может достигнуть точности 95%. Чтобы учесть влияние несбалансированности, в дополнение к точности использовалась метрика MAP (mean average precision, средняя по классам точность):

$$MAP = \frac{1}{|A|} \sum_{a \in A} precision_a,$$

где A — множество авторов, $precision_a$ — точность определения примеров разработчика a.

Значение MAP в описанном выше случае опустится с 95% до 80%, что лучше отражает действительность. Помимо точности и MAP использовалась Acc_k (accuracy at k, точность для k) — доля примеров, для которых правильный ответ находится среди первых k предсказанных. Например, Acc_1 это обычная точность.

Таблица 4.2: Результаты тестирования нейросетевой модели на датасетах IntelliJ IDEA.

	Точность	**Acc$_2$**	**Acc$_5$**	**MAP**
IDEA1	74.4%	86.3%	95.7%	74.3%
IDEA2	99.7%	100%	100%	99.7%
IDEA3	94.2%	97.2%	99%	91.9%
IDEA4	97.8%	99.3%	99.8%	97.4%
IDEA5	92%	96.5%	99.1%	93%
IDEA6	87.3%	93.2%	97.7%	85.9%
IDEA7	89.6%	95.6%	98.9%	88.5%

Тестирование показало, что рабочий контекст оказывает значительное влияние на определение авторства: с увеличением разделения тренировочной и тестовой выборок в датасетах 4-6 значения всех метрик ухудшаются. При разделении выборок по времени точность также падает. Это может быть вызвано как изменением стиля кода авторов, так и сменой рабочего контекста, в котором находится разработчик, поскольку со временем он может менять область ответственности внутри проекта.

4.3. Выводы

Модель на основе случайного леса повторила лучшую на данный момент точность определения авторства на датасете для C++ и улучшила результаты для Java и Python. Нейросетевая модель из-за небольших размеров этих датасетов показала более низкие результаты и была протестирована на описанных в главе 2 данных IntelliJ IDEA. Из результатов тестирования можно заключить, что проверка моделей в условиях разбиения выборок по рабочему контексту и по времени должна быть изучена более подробно, например, создав обучающую выборку из данных в одном или нескольких проектах, а тестовую — из кода тех же авторов в других проектах.

Заключение

Главным результатом данной работы стала модель, улучшающая точность определения авторства по коду на языках программирования C++, Java и Python по сравнению с имеющимися исследованиями в области. В отличие от предыдущих исследований, использовавших свойства конкретных языков и требовавших модификации решений для применения к другим языкам программирования, предложенное решение не нуждается в дополнительной модификации. Также в рамках данной работы были достигнуты следующие результаты:

- Предложен метод для сбора примеров кода с известным автором из произвольных проектов, использовавших при разработке VCS git. Метод основывается на рассмотрении истории проекта и ее дальнейшем разбиении на изменения небольших фрагментов кода. С помощью предложенного подхода можно строить датасеты для тестирования решений в большем масштабе, чем это было возможно ранее, изучать поведение моделей при разбиении тренировочной и тестовой выборки во времени, изучать влияние контекста на точность определения автора кода.

- Предложенный подход к сбору данных был реализован и протестирован на Java-проекте с открытым исходным кодом IntelliJ IDEA. Из него было получено 2 миллиона изменений методов сделанных 500 разработчиками. Из полученных данных было сформировано 7 датасетов для тестирования моделей в разных условиях. Датасеты отличаются числом авторов, числом примеров, степенью несбалансированности. Также представлены датасеты, в которых тренировочная и тестовая выборки разбиты во времени и разделены по частям проекта. Подобные наборы данных до этого не существовали для промышленного кода.

- Созданы две модели для определения авторства по коду. Первая из них это нейросеть, использующая механизм внимания и эмбеддинги на основе путей и токенов в AST для векторизации одного

или нескольких фрагментов кода. Предсказание автора делается по полученному вектору. Модель нацелена на использовании в случае большого количества данных, тысяч или десятков тысяч примеров от каждого из известных разработчиков. Такой масштаб достигается в крупных промышленных проектах. Вторая модель представляет собой случайный лес, обучаемый на частотах встречаемости путей и токенов в AST. Для уменьшения числа факторов и увеличения точности применяется филтрация на основе совместной информации значения фактора и автора. Модель применяется в случае небольшого количества данных, от десятка до нескольких сотен примеров от каждого автора. Обе модели не используют факторов, специфичных для конкретного языка и для применения требуют только построения AST.

- Улучшена точность определения авторства для датасетов на языках Python и Java, повторен результат для C++. При этом использовалась модель на основе случайного леса, поскольку датасеты в предыдущих исследованиях имели небольшой размер. Нейросетевая модель была протестирована на датасетах, собранных из проекта IntelliJ IDEA, представив базовый результат для будущих исследований.

Дальнейшая работа возможна в нескольких направлениях:

- Применить предложенные модели к коду на других языках программирования. Помимо Java, C++ и Python в современных исследованиях было рассмотрено только определение авторства для Javascript [44]. Поскольку предложенные модели позволяют работать с произвольным языком, с их помощью можно расширить множество исследуемых языков.

- Собрать датасеты для определения авторства, моделирующие разные условия применения полученных решений. Например, поместить в тренировочную и тестовую выборки код одних и тех же разработчиков, но в разных проектах, собрать больше данных с

выборками, разделенными по времени, сделать это для разных языков программирования.

- Научиться решать задачу профилирования разработчиков. Нейросетевая модель в процессе работы строит векторы для фрагментов кода. Эксперименты показывают, что по ним можно определять автора фрагмента. В задаче профилирования вместо этого требуется предсказать какие-то свойства автора. Ими могут выступать опыт работы, уровень экспертизы в отдельных областях, знание других языков. Решение задачи профилирования интересно для определения того, как вышеописанные свойства программиста влияют на код, который он пишет.

Список литературы

[1] Halstead M. H. Natural Laws Controlling Algorithm Structure? // SIGPLAN Not. — 1972. — February. — Vol. 7, no. 2. — P. 19--26. — URL: http://doi.acm.org/10.1145/953363.953366.

[2] Oman P. W., Cook C. R. Programming Style Authorship Analysis // Proceedings of the 17th Conference on ACM Annual Computer Science Conference. — CSC '89. — New York, NY, USA : ACM, 1989. — P. 320--326. — URL: http://doi.acm.org/10.1145/75427.75469.

[3] Elenbogen B. S., Seliya N. Detecting Outsourced Student Programming Assignments // J. Comput. Sci. Coll. — 2008. — January. — Vol. 23, no. 3. — P. 50--57. — URL: http://dl.acm.org/citation.cfm?id=1295109.1295123.

[4] Kothari J., Shevertalov M., Stehle E., Mancoridis S. A Probabilistic Approach to Source Code Authorship Identification // Fourth International Conference on Information Technology (ITNG'07). — 2007. — April. — P. 243--248.

[5] Ottenstein K. J. An Algorithmic Approach to the Detection and Prevention of Plagiarism // SIGCSE Bull. — 1976. — December. — Vol. 8, no. 4. — P. 30--41. — URL: http://doi.acm.org/10.1145/382222.382462.

[6] Liu C., Chen C., Han J., Yu P. S. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis // Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. — KDD '06. — New York, NY, USA : ACM, 2006. — P. 872--881. — URL: http://doi.acm.org/10.1145/1150402.1150522.

[7] Layton R., Watters P., Dazeley R. Automatically determining phishing campaigns using the USCAP methodology // 2010 eCrime Researchers Summit. — 2010. — October. — P. 1--8.

[8] Caliskan-Islam A., Harang R., Liu A. et al. De-anonymizing Programmers via Code Stylometry // 24th USENIX Security Symposium (USENIX Security 15). — Washington, D.C. : USENIX Association, 2015. — P. 255--270. — URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/caliskan-islam.

[9] Frantzeskou G., Stamatatos E., Gritzalis S. et al. Identifying Authorship by Byte-Level N-Grams: The Source Code Author Profile (SCAP) Method. // IJDE. — 2007. — January. — Vol. 6.

[10] Krsul I., Spafford E. Authorship analysis: Identifying the author of a program // Computers and Security. — 1997. — December. — Vol. 16. — P. 233--257.

[11] Frantzeskou G., Stamatatos E., Gritzalis S., Katsikas S. Source Code Author Identification Based on N-gram Author Profiles // Artificial Intelligence Applications and Innovations / Ed. by I. Maglogiannis, K. Karpouzis, M. Bramer. — Boston, MA : Springer US, 2006. — P. 508--515.

[12] Rosenblum N., Zhu X., Miller B. P. Who Wrote This Code? Identifying the Authors of Program Binaries // Proceedings of the 16th European Conference on Research in Computer Security. — ESORICS'11. — Berlin, Heidelberg : Springer-Verlag, 2011. — P. 172--189. — URL: http://dl.acm.org/citation.cfm?id=2041225.2041239.

[13] Alsulami B., Dauber E., Harang R. et al. Source Code Authorship Attribution Using Long Short-Term Memory Based Networks // Computer Security -- ESORICS 2017 / Ed. by S. N. Foley, D. Gollmann, E. Snekkenes. — Cham : Springer International Publishing, 2017. — P. 65--82.

[14] Simko L., Zettlemoyer L., Kohno T. Recognizing and Imitating Programmer Style: Adversaries in Program Authorship Attribution // Proceedings on Privacy Enhancing Technologies. — 2018. — Vol. 2018,

no. 1. — P. 127--144. — URL: https://content.sciendo.com/view/journals/popets/2018/1/article-p127.xml.

[15] Shevertalov M., Kothari J., Stehle E., Mancoridis S. On the Use of Discretized Source Code Metrics for Author Identification // 2009 1st International Symposium on Search Based Software Engineering. — 2009. — May. — P. 69--78.

[16] Yang X., Xu G., Li Q. et al. Authorship attribution of source code by using back propagation neural network based on particle swarm optimization // PLOS ONE. — 2017. — November. — Vol. 12, no. 11. — P. 1--18. — URL: https://doi.org/10.1371/journal.pone.0187204.

[17] Zhang C., Wang S., Wu J., Niu Z. Authorship Identification of Source Codes // APWeb/WAIM. — 2017.

[18] Alon U., Zilberstein M., Levy O., Yahav E. A General Path-based Representation for Predicting Program Properties // SIGPLAN Not. — 2018. — June. — Vol. 53, no. 4. — P. 404--419. — URL: http://doi.acm.org/10.1145/3296979.3192412.

[19] Halstead M. H. Toward a Theoretical Basis for Estimating Programming Effort // Proceedings of the 1975 Annual Conference. — ACM '75. — New York, NY, USA : ACM, 1975. — P. 222--224. — URL: http://doi.acm.org/10.1145/800181.810326.

[20] Hamer P. G., Frewin G. D. M.H. Halstead's Software Science - a Critical Examination // Proceedings of the 6th International Conference on Software Engineering. — ICSE '82. — Los Alamitos, CA, USA : IEEE Computer Society Press, 1982. — P. 197--206. — URL: http://dl.acm.org/citation.cfm?id=800254.807762.

[21] Shen V. Y., Conte S. D., Dunsmore H. E. Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support // IEEE Transactions on Software Engineering. — 1983. — March. — Vol. SE-9, no. 2. — P. 155--165.

[22] Bulut N., Halstead M. H., Bayer R. Experimental Validation of a Structural Property of Fortran Algorithms // Proceedings of the 1974 Annual Conference - Volume 1. — ACM '74. — New York, NY, USA : ACM, 1974. — P. 207--211. — URL: http://doi.acm.org/10.1145/800182.810404.

[23] Albrecht A. J., Gaffney J. E. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation // IEEE Transactions on Software Engineering. — 1983. — November. — Vol. SE-9, no. 6. — P. 639--648.

[24] Fitzsimmons A., Love T. A Review and Evaluation of Software Science // ACM Comput. Surv. — 1978. — March. — Vol. 10, no. 1. — P. 3--18. — URL: http://doi.acm.org/10.1145/356715.356717.

[25] Spafford E. H., Weeber S. A. Software Forensics: Can We Track Code to Its Authors? // Comput. Secur. — 1993. — October. — Vol. 12, no. 6. — P. 585--595. — URL: http://dx.doi.org/10.1016/0167-4048(93)90055-A.

[26] Macdonell S. G., Gray A. R., MacLennan G., Sallis P. J. Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis // ICONIP'99. ANZIIS'99 ANNES'99 ACNN'99. 6th International Conference on Neural Information Processing. Proceedings (Cat. No.99EX378). — Vol. 1. — 1999. — November. — P. 66--71 vol.1.

[27] Ding H., Samadzadeh M. Extraction of Java program fingerprints for software authorship identification // Journal of Systems and Software. — 2004. — June. — Vol. 72. — P. 49--57.

[28] Lange R. C., Mancoridis S. Using Code Metric Histograms and Genetic Algorithms to Perform Author Identification for Software Forensics // Proceedings of the 9th Annual Conference on Genetic and Evolutionary

Computation. — GECCO '07. — New York, NY, USA : ACM, 2007. — P. 2082--2089. — URL: http://doi.acm.org/10.1145/1276958.1277364.

[29] Sallis P., Aakjaer A., MacDonell S. Software forensics: old methods for a new science // Proceedings 1996 International Conference Software Engineering: Education and Practice. — 1996. — January. — P. 481--485.

[30] Gray A., Sallis P., MacDonell S. Software Forensics: Extending Authorship Analysis Techniques to Computer Programs. — 1998. — March.

[31] Burrows S., Tahaghoghi S.M.M. Source code authorship attribution using n-grams // ADCS 2007 - Proceedings of the Twelfth Australasian Document Computing Symposium. — 2007. — January.

[32] Krishna S. R. D., Pujari A. K. N-gram analysis for computer virus detection // Journal in Computer Virology. — 2006. — December. — Vol. 2. — P. 231--239.

[33] Vinod B. P. Software Piracy Forensics: A Proposal for Incorporating Dead Codes and Other Programming Blunders as Important Evidence in AFC Test. — 2012. — July. — P. 206--212.

[34] Chen R., Hong L., Lu C., Deng W. Author Identification of Software Source Code with Program Dependence Graphs // 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops. — 2010. — July. — P. 281--286.

[35] Leach R. J. Using Metrics to Evaluate Student Programs // SIGCSE Bull. — 1995. — June. — Vol. 27, no. 2. — P. 41--43. — URL: http://doi.acm.org/10.1145/201998.202010.

[36] Bandara U., Wijayarathna G. Source Code Author Identification with Unsupervised Feature Learning // Pattern Recogn. Lett. — 2013. — February. — Vol. 34, no. 3. — P. 330--334. — URL: http://dx.doi.org/10.1016/j.patrec.2012.10.027.

[37] Dauber E., Caliskan-Islam A., Harang R., Greenstadt R. Git Blame Who?: Stylistic Authorship Attribution of Small, Incomplete Source Code Fragments. — 2017. — January.

[38] Rosenblum N. E., Miller B. P., Zhu X. Extracting Compiler Provenance from Program Binaries // Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. — PASTE '10. — New York, NY, USA : ACM, 2010. — P. 21--28. — URL: http://doi.acm.org/10.1145/1806672.1806678.

[39] Rosenblum N., Miller B. P., Zhu X. Recovering the Toolchain Provenance of Binary Code // Proceedings of the 2011 International Symposium on Software Testing and Analysis. — ISSTA '11. — New York, NY, USA : ACM, 2011. — P. 100--110. — URL: http://doi.acm.org/10.1145/2001420.2001433.

[40] Keselj V., Peng F., Cercone N., Thomas C. N-Gram-Based Author Profiles For Authorship Attribution // Proceedings of the Conference Pacific Association for Computational Linguistics PACLING'03: 2003. — 2003. — September.

[41] Robertson S., Zaragoza H. The Probabilistic Relevance Framework: BM25 and Beyond // Found. Trends Inf. Retr. — 2009. — April. — Vol. 3, no. 4. — P. 333--389. — URL: http://dx.doi.org/10.1561/1500000019.

[42] Hayes J. H. Authorship attribution: A principal component and linear discriminant analysis of the consistent programmer hypothesis // International Journal on Computers and Their Applications. — 2008. — Vol. 15, no. 2. — P. 79--99.

[43] Hayes J. H., Offutt J. Recognizing authors: an examination of the consistent programmer hypothesis // Software Testing, Verification and Reliability. — 2010. — Vol. 20, no. 4. — P. 329--356. — https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.412.

[44] Wisse W., Veenman C. Scripting DNA: Identifying the JavaScript programmer // Digital Investigation. — 2015. — October. — Vol. 15.

[45] Meng X. Fine-grained Binary Code Authorship Identification // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. — FSE 2016. — New York, NY, USA : ACM, 2016. — P. 1097--1099. — URL: http://doi.acm.org/10.1145/2950290.2983962.

[46] Quinlan J. R. C4.5: Programs for Machine Learning. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1993. — ISBN: 1558602402.

[47] He K., Zhang X., Ren S., Sun J. Deep Residual Learning for Image Recognition // 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). — 2016. — June. — P. 770--778.

[48] Young T., Hazarika D., Poria S., Cambria E. Recent Trends in Deep Learning Based Natural Language Processing [Review Article] // IEEE Computational Intelligence Magazine. — 2018. — August. — Vol. 13. — P. 55--75.

[49] Zhang Z., Geiger J., Pohjalainen J. et al. Deep Learning for Environmentally Robust Speech Recognition: An Overview of Recent Developments // ACM Trans. Intell. Syst. Technol. — 2018. — April. — Vol. 9, no. 5. — P. 49:1--49:28. — URL: http://doi.acm.org/10.1145/3178115.

[50] Jordan M. I. Artificial Neural Networks / Ed. by Joachim Diederich. — Piscataway, NJ, USA : IEEE Press, 1990. — P. 112--127. — URL: http://dl.acm.org/citation.cfm?id=104134.104148.

[51] Hochreiter S., Schmidhuber J. Long Short-term Memory // Neural computation. — 1997. — December. — Vol. 9. — P. 1735--80.

[52] Tai K. S., Socher R., Manning C. D. Improved Semantic Representations From Tree-Structured Long Short-Term Memory

Networks // Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). — Beijing, China : Association for Computational Linguistics, 2015. — July. — P. 1556--1566. — URL: https://www.aclweb.org/anthology/P15-1150.

[53] Kennedy J., Eberhart R. Particle swarm optimization // Proceedings of ICNN'95 - International Conference on Neural Networks. — Vol. 4. — 1995. — November. — P. 1942--1948 vol.4.

[54] Alon U., Zilberstein M., Levy O., Yahav E. Code2Vec: Learning Distributed Representations of Code // Proc. ACM Program. Lang. — 2019. — January. — Vol. 3, no. POPL. — P. 40:1--40:29. — URL: http://doi.acm.org/10.1145/3290353.

[55] Bahdanau D., Cho K., Bengio Y. Neural Machine Translation by Jointly Learning to Align and Translate // 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings. — 2015. — URL: http://arxiv.org/abs/1409.0473.

[56] The State of the Octoverse. — 2018. — URL: https://octoverse.github.com/.

[57] Falleri J., Morandat F., Blanc X. et al. Fine-grained and accurate source code differencing // ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. — 2014. — P. 313--324. — URL: http://doi.acm.org/10.1145/2642937.2642982.

[58] Kingma D., Ba J. Adam: A Method for Stochastic Optimization // International Conference on Learning Representations. — 2014. — December.