ОГЛАВЛЕНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

API — набор готовых классов, процедур, функций, структур и констант, предоставляемых сервисом для использования во внешних программных продуктах.

IDE (Integrated development environment, интеграированная среда разработки) — комплекс программных средств, используемый программистами для разработки ПО. Как правило, среда разработки включает в себя: текстовый редактор, компилятор и/или интерпретатор, средства автоматизации сборки, отладчик.

IntelliJ платформа — платформа, разработанная для создания инструментов анализа кода на различных языках программирования в рамках IDE.

JVM (Java Virtual Machine) — виртуальная машина ЯП Java — основная часть исполняющей системы Java, так называемой Java Runtime Environment (JRE). Виртуальная машина Java исполняет байт-код Java, предварительно созданный из исходного текста Java-программы компилятором Java.

Байт-код — стандартное промежуточное представление, в которое может быть переведена компьютерная программа автоматическими средствами. По сравнению с исходным кодом, удобным для создания и чтения человеком, байт-код — это компактное представление программы, уже прошедшей синтаксический и семантический анализ.

Дерево разбора PSI (Program Structure Interface) — конкретное синтаксическое дерево определенного формата, используемое в IntelliJ платформе.

Кодогенератор — специальная часть компилятора, которая конвертирует синтаксически корректную программу в последовательность инструкций, которые могут выполняться на машине.

Оптимизации компилятора — методы получения более оптимального программного кода при сохранении его функциональных возможностей.

ПО — программное обеспечение.

Профилировщик (профайлер) — инструмент, предназначенный для сбора характеристик программы, таких как время выполнения отдельных ее фрагментов (функций, строк) с целью оценки производительности программы и проведения её дальнейшей оптимизации.

Целевая программа — программа, получающаяся в результате работы компилятора.

Целевой язык — язык, на котором составлена целевая программа.

ЯП — язык программирования.

ВВЕДЕНИЕ

В области разработки программного обеспечения часто поднимается вопрос производительности разрабатываемых программ. Требование к производительности — одно из важнейших нефункциональных требований для большинства продуктов.

На пути от написания кода программы до исполнения соответствующего ей машинного кода на процессоре есть множество факторов, которые так или иначе могут влиять на производительность разрабатываемой программы.

Не углубляясь в конкретные факторы, выделим стадии с этими факторами от написания кода до исполнения машинного кода программы на процессоре:

1) написание кода программистом;
2) компиляция кода (некоторые стадии могут либо отсутствовать, либо быть совмещены):
    а) лексический анализ,
    б) синтаксический анализ,
    в) семантический анализ,
    г) оптимизации на абстрактном синтаксическом дереве,
    д) трансляция в промежуточное представление,
    е) машинно-независимые оптимизации,
    ж) трансляция в конечное представление (машинный код),
    з) машинно-зависимые оптимизации;
3) интерпретация или компиляция «на лету» промежуточного кода виртуальной машиной (при трансляции кода компилятором не в машинный код, а в код некоторой виртуальной машины);
4) планирование исполнения машинного кода ядром операционной системы;

5) исполнение кода программы на процессоре;
6) исполнение кода функций библиотек поддержки времени исполнения, используемых в программе;
7) исполнение кода системных вызовов, используемых в программе.

На каждой из перечисленных стадий существует множество факторов, способных в конечном счете повлиять на производительность программы. Под контролем программиста непосредственно исходной программы находится лишь первая группа факторов. За остальные группы факторов ответственны разработчики соответствующих инструментов и вспомогательных программ или их частей: виртуальных машин, библиотек поддержки времени исполнения, ядер операционных систем и непосредственно самих процессоров и других физических компонентов, способных влиять на производительность исполняемой программы.

В данной работе предлагается провести исследование, связанное с анализом стадии компиляции кода. Предлагается осуществлять поиск кодовых аномалий, которые потенциально могут способствовать нахождению проблем производительности в компиляторе. Под кодовой аномалией понимается нетипичный по меркам анализируемого набора данных код. На таком коде работа компилятора либо не изучена разработчиками языка, либо малоизучена, из-за чего на подобных примерах и могут быть обнаружены проблемы в производительности. Соответственно, предполагается, что результаты такого исследования будут использованы разработчиками языка.

В качестве языка программирования, программы на котором и компилятор которого будут исследоваться, был выбран Kotlin[1], разработанный компанией JetBrains, как один из наиболее молодых, быстро развивающихся и ещё недостаточно исследованных языков. Kotlin — это высокоуровневый язык программирования общего назначения, работающий

[1] https://kotlinlang.org/

поверх JVM (также есть компиляция в Javascript и в код других платформ с использованием инфраструктуры LLVM). В мае 2017 года язык Kotlin был включен в Android Studio 3.0 [24], как официальный язык для разработки приложений под ОС Android, что стало причиной для увеличения его аудитории.

Таким образом, целью данной магистерской диссертации является разработка набора инструментов для анализа исходного кода программ на Kotlin и проведение соответствующего исследования на некотором наборе данных с целью поиска аномалий и возможного дальнейшего выявления проблем производительности в компиляторе разработчиками Kotlin на основе полученных аномалий. Анализ предлагается проводить в доступных представлениях кода: в виде дерева разбора PSI, а также сгенерированного по нему JVM байт-кода.

Практическая значимость работы заключается в получении по результатам исследования и разработки списка файлов с исходным кодом и байт-кодом из достаточно большого набора данных, являющихся с точки зрения тех или иных алгоритмов аномальным.

ГЛАВА 1. АНАЛИЗ ПРДЕМЕТНОЙ ОБЛАСТИ И ПОСТАНОВКА ЗАДАЧИ

1.1 Анализ предметной области

Предметной областью является анализ исходного кода на языке программирования Kotlin (и других представлений этого исходного кода) с целью поиска аномалий, которые впоследствии, по мнению разработчиков Kotlin, могут способствовать обнаружению проблем производительности.

Компилятор Kotlin транслирует исходный код в промежуточный код (называемый также байт-кодом) — набор инструкций для виртуальной машины ЯП Java — JVM. Исследование предлагается проводить в рамках представлений исходного кода, ответственность за которые лежит непосредственно на стороне компилятора Kotlin (дальнейшая интерпретация или компиляция байт-кода JVM «на лету» в машинный код и его исполнение рассматриваться не будут, т. к. данные стадии лежат за рамками ответственности компилятора Kotlin).

В компиляторе Kotlin производятся следующие преобразования исходного кода:

1. Преобразование исходного кода в набор лексем, и, далее — в набор токенов (токенизация или лексический анализ);
2. Построение дерева разбора на основе набора лексем (в компиляторе Kotlin данное дерево называется PSI — Program Structure Interface — по общепринятой терминологии его можно назвать *конкретным синтаксическим деревом* или просто *деревом разбора*);
3. Генерация байт-кода JVM на основе дерева разбора PSI.

Стоит отметить, что построение дерева разбора PSI возможно на любом исходном коде. Для синтаксически некорректных конструкций (тех, которые не вписываются в грамматику Kotlin) в дереве разбора будет создаваться узел специального типа. В то время, как генерация байт-кода JVM по исходному

коду с синтаксически и/или семантически некорректными конструкциями невозможна.

На данный момент в компиляторе Kotlin нет представления исходного кода в виде абстрактного синтаксического дерева при компиляции в JVM байт-код. Генерация JVM байт-кода производится напрямую по дереву разбора PSI.

Дерево разбора PSI, как уже было сказано, представляет из себя конкретное синтаксическое дерево. В отличие от абстрактного синтаксического дерева в нём в том числе содержатся узлы, несущие исключительно синтаксический характер (не несущие семантический характер с точки зрения целевого языка — например, отступы, пробелы, комментарии и т. д.). Листья данного дерева, как правило, соответствуют конкретным токенам (таким образом, список листьев данного дерева является набором токенов, которые, в свою очередь, являются результатом работы лексического анализатора). Корень дерева разбора PSI является узлом с типом FILE, который представляет весь исходный код анализируемого файла. Таким образом, по данному дереву разбора может быть однозначно восстановлен исходный код программы.

1.2 Уточнение постановки задачи

Предполагается, что разработчикам Kotlin будут интересны следующие результаты:

1. полный набор найденных файлов-аномалий с исходным кодом и байт-кодом: предполагается, что такие файлы могут способствовать переосмыслению некоторых дизайнерских решений в конструкциях языка, либо разработке новых конструкций — вероятно, код таких файлов будет трактоваться как нерациональное использование языка, также, такие аномалии могут быть включены в тесты на производительность компилятора, как примеры сложного, нетипичного и мало исследованного кода;

2. набор файлов с исходным кодом, сгенерированный JVM байт-код по которым считается аномальным, а дерево разбора PSI не превышает некоторые заданные ограничения (на глубину, на количество узлов, либо по других критериям), такой случай может соответствовать одному из двух вариантов:
    2.1. по заданной конструкции было либо сгенерировано больше JVM байт-кода, чем ожидалось, либо был сгенерирован нетипичный (по меркам анализируемого набора данных) JVM байт-код, что может свидетельствовать о наличии каких-либо проблем в кодогенераторе или оптимизациях компилятора Kotlin;
    2.2. по заданной конструкции было сгенерировано большое количество JVM байт-кода вследствие удачной реализации данной конструкции: код, который предполагает выполнение большого числа действий, удалось записать достаточно коротко (данный случай, хоть и имеет позитивный характер, также должен быть интересен разработчикам компилятора Kotlin — можно перенять опыт реализации данных конструкций для других конструкций);
3. набор файлов с исходным кодом, дерево разбора PSI которых считается аномальным, а размер JVM байт-кода не превышает некоторые заданные ограничения: гипотетически такой случай также должен нести позитивный характер: по аномальному дереву разбора PSI был сгенерирован достаточно небольшой JVM байт-код, что безусловно должно позитивно сказаться на производительности программы (но такие файлы также могут быть интересны разработчикам компилятора Kotlin: можно также перенять опыт реализации данных конструкций, и, помимо этого, слишком большая разница в размере дерева разбора PSI и JVM байт-кода может

гипотетически соответствовать каким-либо проблемам генерации кода).

Предполагается и следует из вышеизложенного, что анализ исходного кода и JVM байт-кода будет статическим (без фактического запуска программы), т. к. анализ поведения программы во время исполнения выходит за обозначенные рамки: целью исследования является анализ исходного кода в разных представлениях.

Также предполагается отсутствие заранее предоставленных примеров аномалий кода.

Найденные аномалии должны сопровождаться некоторым численным показателем, показывающим «степень аномальности» для дальнейшего ранжирования списка и отсечения не интересующих примеров.

Для анализа поведения программы (анализа во время исполнения) существует множество инструментов. Одни из самых популярных — профилировщики. Они осуществляют сбор характеристик работы программы: времени выполнения отдельных фрагментов, числа верно предсказанных условных переходов, числа кэш-промахов и т. д. Данные характеристики могут быть так же использованы для оценки производительности программы в целом и для осуществления её дальнейшей оптимизации. Но в данном способе поиска потенциальных проблем производительности по сравнению с предложенным есть ряд отличий:

1. необходимость фактического запуска программы в тестовой среде, что, как правило, является более сложно организуемым, чем статический анализ кода;
2. оценка лишь фактического времени выполнения части кода (как правило, замеряется время и другие характеристики процесса исполнения соответствующего JVM байт-кода), без привязки к конкретной стадии преобразования анализируемого кода, которая

могла повлечь найденную проблему (по этой причине динамический анализ кода будет мало полезен для разработчиков языка);

3. более позднее обнаружение проблемы, как правило, может повлечь большие убытки различного характера (стадия написания кода и его компиляции в отличие от стадии исполнения является самой ранней в этапе программирования при разработке ПО).

По вышеизложенным причинам для решения обозначенной задачи и был выбран статический анализ кода.

ГЛАВА 2. ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ

Задача поиска кодовых аномалий уже была поставлена ранее. Рассмотрим различные работы в этой области, предлагаемые их авторами решения и сопоставим их с поставленной задачей. Для исследования проблемы поиска кодовых аномалий в целом рассмотрим в том числе и работы, предполагающие динамический анализ кода.

В рамках работы «Graph-based Mining of Multiple Object Usage Patterns» [2] была разработана система GrouMiner (для программ на Java), которая включает в себя возможность описания шаблонов взаимодействия объектов в объектно-ориентированном программировании (основанную на использовании графов) и механизмы обнаружения аномалий (аномального взаимодействия). Подход предполагает моделирование взаимодействия в виде направленного ациклического графа, узлы которого являются вызовами конструкторов, методов объектов и обращениями к полям, а ребра — фактами использования одними методами других и зависимостями данных между ними. В данной работе для поиска аномалий используется статический анализ кода: исходный код разбирается в абстрактное синтаксическое дерево, далее в нём выделяются узлы для передачи или приеме управления, и на основе них формируется граф использования объектов. Данный подход позволяет обнаруживать аномалии в исходном коде, связанные со взаимодействием объектов. Но такие аномалии — лишь часть из всех возможных. Аномалии могут быть связаны, например, с нетипичным использованием некоторых конструкций языка (не предполагающих передачу или прием управления), которые в свою очередь могут соответствовать проблемам в дизайне языка, в кодогенераторе или оптимизациях его компилятора.

В работе «Detecting Object Usage Anomalies» [3] предлагается введение понятия «типичного использования объекта» и осуществление поиска его аномального использования. Кодовые аномалии могут быть выражены, например, в нетипичной последовательности вызовов методов. Данный

подход предполагает использование статического анализа кода. Происходит сбор информации по использованию методов объекта и формирование шаблонов, далее с помощью классификатора происходит поиск мест использований объектов, отклоняющихся от этих шаблонов (аномалий). В данной работе также можно заметить направленность только на аномальное использования объектов, что также является лишь одной разновидностью кодовых аномалий.

В работе «Tracking down software bugs using automatic anomaly detection» [1] была реализована система DIDUCE для Java-программ, которая предполагает поиск аномалий с использованием динамического анализа кода: система динамически формулирует гипотезы об инвариантах в исполняемой программе, начиная с самых строгих, но впоследствии ослабляя их и уведомляя при этом ослаблении пользователя об аномалиях, которые соответствуют потенциальным ошибкам в программе. Векторное представление программы состоит из набора значений выражений, которые и рассматриваются системой, как инварианты. Данный подход направлен прежде всего на пользователей языка программирования и будет малопригоден для разработчиков языка, поскольку здесь не анализируется непосредственно исходный код и его представление, которые являются основным объектом интереса в поставленной задаче (анализ непосредственно JVM байт-кода предполагается только в связке с анализом PSI деревьев разбора).

В работах [4] и [5] предлагается анализ осуществления системных вызовов для детектирования аномального поведения программ. В данном подходе используется динамический анализ кода (анализ поведения). Стоит отметить, что такой анализ кода будет опять же мало пригоден для разработчиков языка программирования. А аномальные наборы системных вызовов также являются лишь одной из разновидностей аномалий.

Таким образом, были рассмотрены уже существующие работы в области обнаружения кодовых аномалий и отмечены их недостатки и непригодность или лишь частичная пригодность для поставленной задачи. Можно сделать вывод о необходимости разработки инструмента, который бы удовлетворял заявленным требованиям, и проведения соответствующего исследования.

Также, стоит отметить, что параллельно с данной работой в Санкт-Петербургском государственном университете выполнялась работа «Детектор аномалий в программах на языке Kotlin», целью которой также являлся поиск кодовых аномалий. Но в данной работе использовался несколько другой подход, о чем будет подробнее рассказано в главе 3.

ГЛАВА 3. ОБЗОР МЕТОДОВ И СРЕДСТВ РЕШЕНИЯ ЗАДАЧИ

3.1 Определение области решения задачи

Задача обнаружения аномалий уже была поставлена в области машинного обучения. Для её решения существует множество техник. Для поставленной задачи рационально использовать алгоритмы именно машинного обучения, поскольку понятие «нетипичности» (аномальности) является сложно формализуемым и описать в терминах классических алгоритмов его будет сложно, такое описание также создаст границы — мы не сможем обнаруживать объекты, которые являются также аномальными, но по каким-либо другим параметрам, которые не были описаны.

Рассмотрим существующие техники машинного обучения, оценим их пригодность для поставленной задачи и определим условия для их применения.

3.2 Методы решения задачи обнаружения аномалий

В области машинного обучения под аномалией понимают отклонение поведения системы от ожидаемого. Аномалии обычно подразделяют на три вида [18]:

1. точечные аномалии соответствуют случаям, когда отдельный объект данных является аномальным по отношению к остальным данным (см. рисунок 3.2.1: C1 и C2 — не аномальные группы объектов, A1 и A1 — аномалии);
2. контекстуальные аномалии соответствует случаям, когда отдельный объект данных является аномальным лишь в определенном контексте, определяется контекстуальными атрибутами — например, определенным временем наблюдения аномальных объектов (см. рисунок 3.2.2: в точке A наблюдается аномалия в отличие от точек N1-N5 с аналогичным значением функции);

3. коллективные аномалии соответствуют случаям, когда совместное появление некоторого числа объектов является аномальным (см. рисунок 3.2.2: участок A является коллективной аномалией).

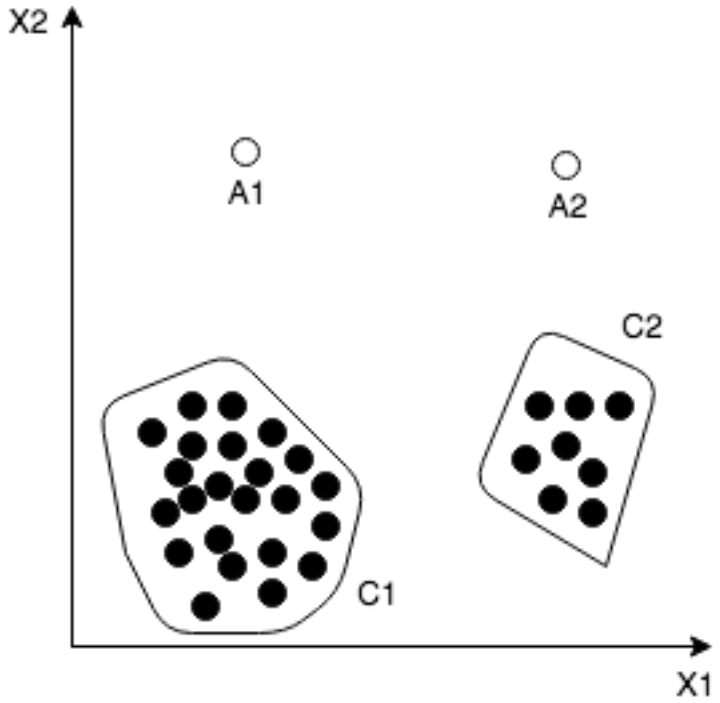

Рисунок 3.2.1. Демонстрация вида точечных аномалий.

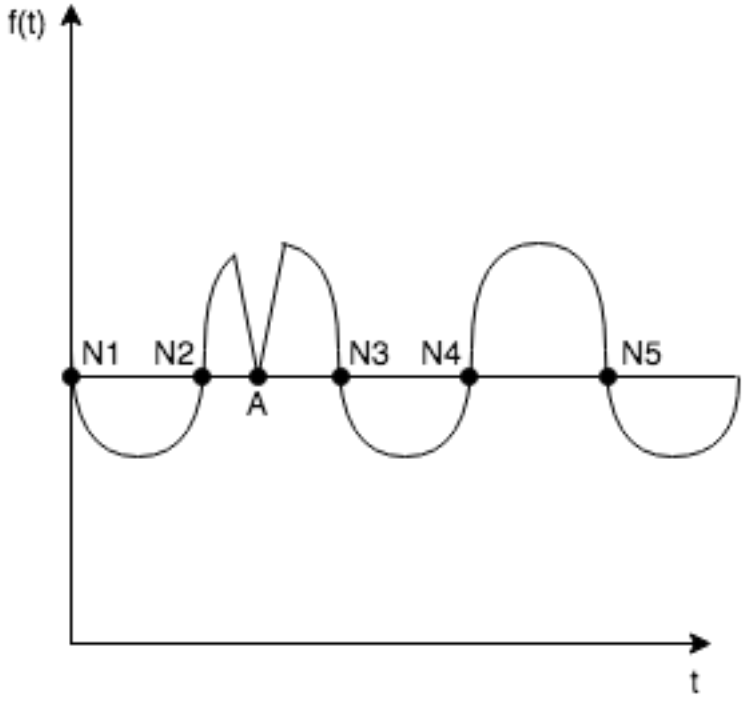

Рисунок 3.2.2. Демонстрация вида контекстуальных аномалий.

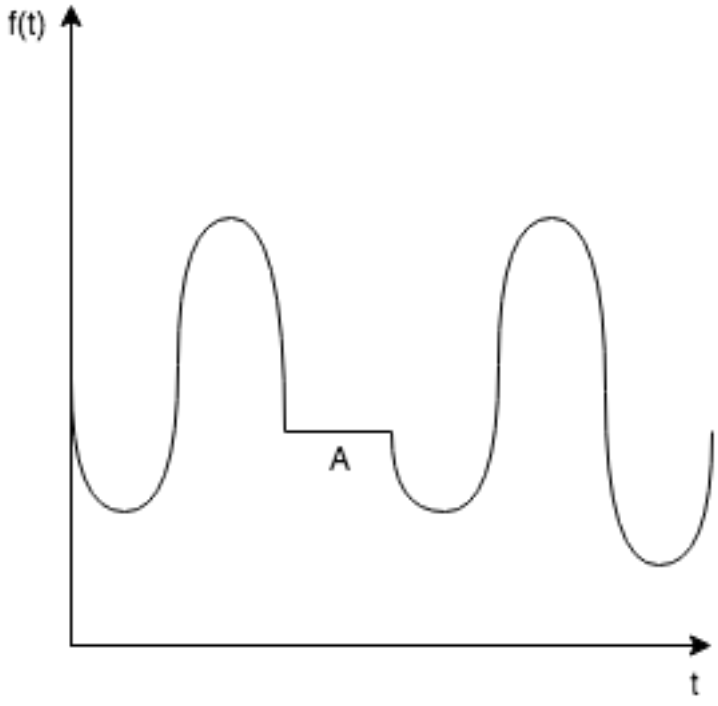

Рисунок 3.2.3. Демонстрация вида коллективных аномалий.

Для поставленной задачи предполагается обнаружение точечных аномалий, т. к. потенциальные объекты-аномалии, деревья разбора PSI и JVM байт-код, рассматриваются изолированно (не коллективные аномалии) и не имеют контекстной привязки к чему-либо (не контекстуальные аномалии).

Контекстуальными аномалиями, например, могут являться части дерева (в контексте других частей дерева), либо отдельные узлы в контексте других узлов. Но, в соответствии с постановкой задачи, нас интересуют лишь аномалии, представляющие из себя всё дерево/список JVM-инструкций, а не отдельные части дерева/списка или узлы.

Существующие методы обнаружения аномалий делят на три больших категории:

1. Обучение с учителем: в обучающей выборке на вход поступают помеченные данные, каждый объект относят либо к нормальной группе, либо к группе аномалий (сюда же относят обучение с частичным привлечением учителя, когда для тренировки используется небольшое количество размеченных данных и большое количество неразмеченных);
2. Обучение без учителя: в обучающей выборке на вход поступают непомеченные данные, обучение происходит без вмешательства экспериментатора.

Нас будет интересовать группа методов обучения без учителя, так как в поставленной задаче предполагается отсутствие примеров объектов-аномалий (предполагается работа с непомеченными данными).

Рассмотрим существующие техники с обучением без учителя, позволяющие осуществлять обнаружение точечных аномалий.

3.2.1 Одноклассовый метод опорных векторов

Данный метод [19] относится к группе методов, основанных на классификации, но в нём предполагается наличие лишь одного класса (при том областью поиска будут объекты, которые нельзя отнести к этому единственному классу). То есть метод позволяет работать с непомеченными данными (обучаться без учителя).

Метод использует перевод исходных векторов в пространство более высокой размерности и поиск разделяющей гиперплоскости с максимальным зазором в этом пространстве.

По результатам работы предполагается получение двух наборов объектов: тех, которые удалось отнести к единственному классу (не аномалии) и тех, которые отнести к этому классу не удалось (аномалии).

3.2.2 Методы, основанные на кластеризации

В методах кластеризации [20] объекты данных, которые похожи, относятся к сходным группам или кластерам, что определяется их расстоянием от локальных центров. Данные методы предлагают решение задачи с помощью обучения без учителя (т. е. подразумеваются непомеченные входные данные).

Применительно к задаче поиска аномалий можно выделять те объекты, которые не удалось отнести ни к одному кластеру (такие объекты помечаются как аномальные).

На выходе, так же, как и в методах, основанных на классификации, предполагается двоичный результат — принадлежность к тому или иному кластеру (а применительно к задаче поиска аномалий — принадлежность к какому-либо известному кластеру, либо неизвестному). Часть методов также требует на входе задания числа кластеров.

Одним из представителей данной группы методов является метод k-средних (k-means). Он подразумевает разбиение множества элементов векторного пространства на заранее заданное число кластеров. Идея состоит в том, что на каждой итерации заново вычисляется центр масс кластера (тем самым минимизируется суммарное квадратичное отклонение точек кластеров от центров этих кластеров). Остановка алгоритма происходит тогда, когда внутрикластерное расстояние перестает меняться.

К методам кластеризации, заранее не требующим задания числа кластеров, относятся, например, алгоритмы семейства FOREL.

3.2.3 Статистические методы

Статистические методы [21] предполагают сопоставление точек с некоторым статическим распределением. Если отклонение от распределения превышает некоторое пороговое значение, то объект считается аномальным. Соответственно, на выходе можно получить не просто двоичный результат (принадлежность к аномалиям), но и численную оценку — степень аномальности, выраженную в величине отклонения.

Для части методов данной группы требуется начальное предположение о распределении данных, для другой части построение модели возможно на самих данных, без априорных сведений.

3.2.4 Методы, основанные на применении репликаторных нейронных сетей

Методы данной группы [22] основаны на сжатии данных с потерями и их последующем восстановлении. Ошибку восстановления применительно к задаче поиска аномалий можно трактовать как численную оценку аномальности.

Данные нейронные сети способны обучаться без учителя, то есть для них не требуются помеченные данные.

Одним из представителей репликаторных нейронных сетей является автоэнкодер (autoencoder). Автоэнкодер использует метод обратного распространения ошибки. Простейшая архитектура автоэнкодера представляет из себя три слоя: входной, промежуточный и выходной. Количество нейронов на входном слое должно совпадать с количеством нейронов на выходном. Количество нейронов на промежуточном слое соответствует требуемой степени сжатия данных (но обязательно меньше количества нейронов на выходном и выходном слоях). Принцип обучения

автоэнкодера заключается в получении на выходном слое отклика, наиболее близкого к входному.

3.2.5 Выбор метода для решения задачи обнаружения аномалий

Сделав обзор существующих методов обнаружения аномалий, можно оценить пригодность каждого из них для поставленной задачи.

Группа статистических методов позволяет давать оценку аномальности, которая выражается в величине отклонения от распределения, что является плюсом данных методов для поставленной задачи. Но с другой стороны для данной группы методов требуется начальное знание о распределении данных. К сожалению, о распределении по собранным данным нам ничего неизвестно.

Методы, основанные на кластеризации, в свою очередь, не требуют априорных знаний о наборе данных, но, с другой стороны, не позволяют дать оценку степени аномальности, а лишь позволяют определить принадлежность к классу аномалий (к примерам, не попавших ни в один кластер).

Одноклассовый метод опорных векторов также не требует априорных сведений о данных, но по результатам своей работы предоставляет для каждого объекта лишь маркер успешности отнесения к единственному классу (аномалиями являются примеры, которые отнести к этому классу не удалось).

Методы, основанные на применении репликаторных нейронных сетей, помимо отсутствия требования к начальным знаниям о данных, по окончании своей работы могут давать оценку аномальности, выраженную в ошибке восстановления. Например, автоэнкодер уже содержит в себе механизмы сокращения размерности (иногда для этого вводят дополнительные слои), а в итоговом наборе данных ожидается, что будет достаточно большое количество признаков.

Таким образом, выбор был сделан в сторону автоэнкодера, который удовлетворяет всем заявленным требованиям. Также, предполагается, что автоэнкодер будет более эффективно работать с данными, так как

предположительно они будут содержать большое количество признаков. Помимо прочего, он по окончании своей работы отдает набор восстановленных векторов, что предоставляет некую гибкость: можно анализировать как вектора разностей входного и выходного векторов каждого объекта, так и расстояния между этими векторами. Анализ векторов разностей, например, может позволить не просто определять принадлежность объекта к группе аномалий с некой оценкой степени аномальности, но и понимать, какие именно компоненты вектора вызвали эту аномальность.

Была составлена простая модель автоэнкодера (продемонстрирована на рис. 3.2.6.1), состоящая из трех слоев: входной, промежуточный и выходной. В качестве функции активации в процессе кодирования (переход от входного слоя к промежуточному) была выбрана rectifier (relu). Для процесса декодирования была выбрана функция активации sigmoid (переход от промежуточного слоя к выходному). Рекомендации по выбору данных функций были предоставлены в статье «Building Autoencoders in Keras» [15]. Количество нейронов на входном и выходном слоях равно количеству измерений (n-грамм). Количество нейронов на промежуточном слое в два раза меньше, чем на входном и выходном (то есть коэффициент сжатия был выбран равным 2 [23]).

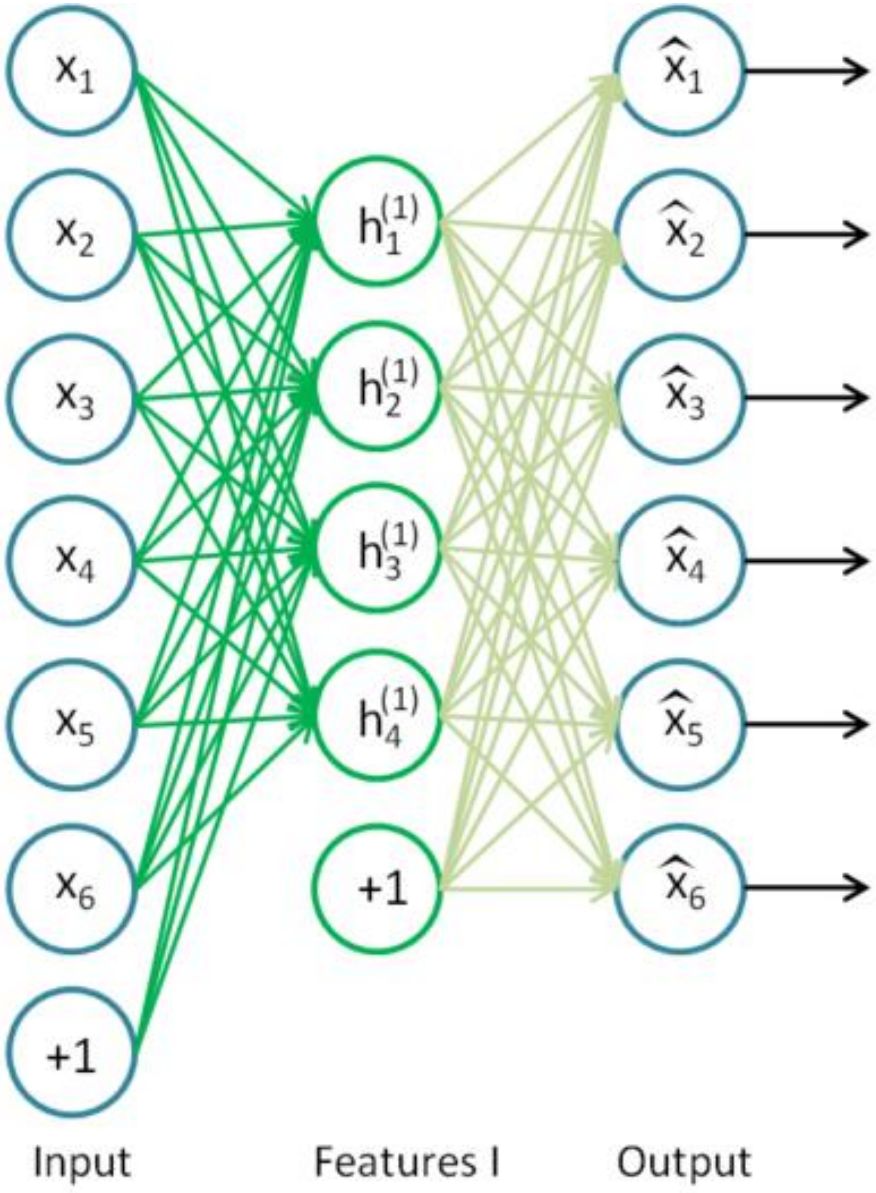

Рисунок 3.2.6.1. Схематичное изображение модели автоэнкодера.

3.3 Задача подготовки данных

Поскольку большинство алгоритмов машинного обучения (в том числе и выбранный для решения задачи обнаружения аномалий автоэнкодер) требует на вход набор данных в виде набора векторов (вектора соответствуют объектам, а компоненты векторов — признакам), встает задача подготовки исходных данных, в т. ч. формирования признаков — формирования векторного представления дерева разбора PSI и JVM байт-кода.

3.3.1 Выбор способа векторного представления дерева разбора PSI и JVM байт-кода

Для формирования векторного представления дерева разбора PSI и JVM байт-кода было выбрано разложение их на n-граммы. В отличие от выделения признаков ручным образом такой способ кодирования дерева и списка позволяет выявлять в том числе и неявные признаки (например, стоящие на определенном расстоянии две конструкции цикла), которые сложно интерпретировать и описать ручным образом (для n-грамм с $n \geq 2$), но которые могут внести существенный вклад в формирование аномалии.

В работе «Детектор аномалий в программах на языке Kotlin», о которой было упомянуто в главе 2, использовался подход с явным извлечением признаков, который также имеет свои плюсы, но не позволяет извлекать сложно формализуемые признаки.

Стоит также отметить, что помимо неявного извлечения признаков путем разложения на n-граммы есть и ряд других. Так, в [8, 9] используется способ кодирования, называемый «Code criterion». Он подразумевает кодирование отдельных узлов через дочерние (то есть каждому узлу дерева сопоставляется вектор).

В работе [10] была реализована система DECKARD. Она также предполагает кодирование отдельных узлов — на основе дочерних узлов, включая их количество и счетчики встречаемости узлов такого же типа по всему дереву.

Был сделан выбор в пользу разложения на n-граммы по причине того, что аномалии, в соответствии с постановкой задачи, должны представлять не отдельные узлы в дереве, а примеры целиком (файлы с исходным кодом и с байт-кодом).

Таким образом, после такой факторизации подразумевается, что будет получен набор векторов. Каждый вектор соответствует файлу с исходным

кодом или файлу с байт-кодом. Компоненты векторов соответствуют количествам встречаемости в рассматриваемом дереве разбора или байт-коде тех или иных n-грамм.

Ещё одно преимущество данного метода формирования векторного представления заключается в попутном сборе легко интерпретируемой человеком информации о программе (в случае с uni-граммами): имея векторные представления деревьев разбора PSI, можно посчитать количество наиболее часто используемых конструкций, либо наоборот — наиболее редко используемых. Данная информация может быть полезна как для дальнейшего машинного анализа, так и для оценки востребованности конструкций, переосмысления их дизайна и учета данной информации при разработке конструкций в будущем.

ГЛАВА 4. ПРЕДЛАГАЕМОЕ РЕШЕНИЕ

4.1 Сбор данных

Поскольку аномалии должны составлять существенно малую часть набора данных, сам набор данных должен быть достаточно большим. Чем больше набор данных, тем больше аномалий удастся обнаружить.

Одним из самых больших хранилищ кода на Kotlin является ресурс GitHub. Так, по данных на ноябрь 2017 года на ресурсе GitHub размещено файлов с исходных кодом, составляющих суммарно более 25 миллионов строк кода [14]. Также ресурс GitHub имеет удобный в использовании API, который позволяет встроить в том числе процесс получения исходных кодов во внешнюю программу. Альтернативой GitHub является ресурс GitLab, который также содержит открытый исходный код на языке Kotlin. Но GitLab содержит существенно меньшее количество кода на Kotlin, а для поиска аномалий нам требуется сформировать набор данных как можно большего размера.

Таким образом, был сделан выбор в пользу GitHub в качестве источника данных.

Эксперимент предлагается проводить на двух наборах данных: на малом, полученным пофайловым сбором кода с GitHub, и на большом, полученным уже сбором кода по репозиториям путем их клонирования и дальнейшей фильтрации.

В большом наборе данных контекст репозитория также может оказаться полезным для смежных исследований (например, на основе конфигурации сборки можно сопровождать аномалии дополнительной информацией, потенциально полезной для разработчиков Kotlin при дальнейшем изучении найденных аномалий). По предварительным подсчетам, всего GitHub содержит 47 тыс. репозиториев, основным языком которых является Kotlin (большинство кода написано на нём). Для пофайлового сбора кода с учетом ограничений в API со стороны GitHub нам доступно примерно 50 тыс. файлов.

JVM байт-код удобно собирать также на GitHub. Он предоставляет функциональность для публикации и хранения готовых сборок проекта. Будем осуществлять выкачивание данных сборок для дальнейшего извлечения из них JVM байт-кода.

Для большого набора данных байт-код будем собирать из готовых сборок соответствующих репозиториев. Для малого — из сборок ограниченного числа репозиториев.

Предполагается, что с результатами на малом наборе данных будет проведена ручная работа (например, классификация полученных аномалий и сбор экспертных оценок), на большем же наборе данных предполагается автоматизированная работа с результатами по сравниванию аномалий по дереву разбора и по байт-коду.

Перед сбором непосредственно самого кода был предоставлен список всех репозиториев, основным языком которых является Kotlin, авторами похожей работы — «Детектор аномалий в программах на языке Kotlin», о которой было рассказано в главе 2.

Далее был разработан инструмент[2], который по файлу со списком репозиториев осуществлял скачивание данных репозиториев, а также готовых сборок, если они имелись у соответствующего репозитория.

Для пофайлового сбора кода (для малого набора данных) также были разработаны инструменты, первый из которых[3] путем обращения к API GitHub получает список файлов с кодом на Kotlin и сохраняет их в указанное место, второй[4] — собирает готовые сборки проектов в соответствии с ограничением GitHub API по репозиториям.

4.2 Сопоставление и фильтрация и собранных файлов

Поскольку в загруженных файлах сборок часто находились не только файлы с релевантным байт-кодом, но и файлы с байт-кодом сторонних библиотек, встает задача сопоставления файлам с исходным кодом и последующей фильтрации.

Будем осуществлять сопоставление только для большого набора данных, т. к. файлы готовых сборок есть лишь у небольшого числа репозиториев (оно становится уместным лишь на большом наборе данных). Сбор исходного кода и байт-кода для малого набора данных происходил изолировано и без дополнительного связывания на уровне репозиториев.

Для решения задачи сопоставления файлам с исходными кодами файлов с байт-кодом был разработан инструмент[5], который выполняет фильтрацию файлов репозитория (оставляя только файлы с исходным кодом Kotlin), а также сопоставление этих файлов с файлами с байт-кодом (и удалением нерелевантных файлов).

Сопоставление файлов выполняется следующим образом: сначала осуществляется разбор файлов с JVM байт-кодом, из мета-информации

[2] https://github.com/PetukhovVictor/github-kotlin-repo-collector
[3] https://github.com/PetukhovVictor/github-kotlin-code-collector
[4] https://github.com/PetukhovVictor/github-kotlin-jar-collector
[5] https://github.com/PetukhovVictor/bytecode-to-source-mapper

которых достается название пакета и название файла, из которого был сгенерировал данный байт-код; далее происходит обход исходных кодов и с помощью регулярного выражения из каждого файла достается название пакета; в результате удается однозначно сопоставить файлы с исходным кодом и файлы с байт-кодом. Файлы, которые сопоставить не удалось, являются файлами сторонних библиотек, и подлежат удалению.

4.3 Разбор собранных файлов

После сбора исходных кодов и файлов с байт-кодом встает задача парсинга исходных кодов (получение дерева разбора PSI) и байт-кода (получения списка JVM-инструкций, сгруппированных по методам). Будем записывать результат в формате JSON для удобства дальнейшей обработки.

Для парсинга исходных кодов на Kotlin был склонирован репозиторий компилятора Kotlin[6], в код которого была добавлена возможность промежуточного вывода дерева разбора PSI. Далее код компилятора был скомпилирован и получен бинарный файл компилятора, который по запуску с указанием пути к директории с исходными кодами на Kotlin записывал деревья разбора в файлы с аналогичным именем, но с добавленным расширением json.

Для файлов с байт-кодом был также разработан инструмент[7], преобразующий файлы с байт-кодом в json-файлы (class-файлы), с JVM-инструкциями, сгруппированными по методам.

4.4 Факторизация деревьев разбора и байт-кода

Для осуществления факторизации деревьев разбора PSI и JVM байт-кода был разработан инструмент[8], который в зависимости об объекта факторизации

[6] https://github.com/PetukhovVictor/kotlin-academic
[7] https://github.com/PetukhovVictor/bytecode-parser
[8] https://github.com/PetukhovVictor/ngram-generator

(дерево разбора или байт-код) осуществляет извлечение n-грамм соответствующим образом.

По окончании работы инструмент записывает файл в формате json, в котором сопоставлены идентификаторы n-грамм их количеству встречаемости.

Написанный инструмент в режиме «факторизация по дереву» осуществляет извлечение униграмм, биграмм и триграмм в соответствии с заданным плавающим окном. Плавающее окно является максимально допустимым расстоянием (т. е. количеством других узлов) между двумя соседними узлами в составляемой n-грамме.

Принцип работы следующий. На протяжении всего обхода дерева алгоритм помнит историю обхода и при достижении очередного узла использует её для составления уникальных путей от этого узла и дальше со следующим свойством: если в таких путях зафиксировать первый и последний элементы, а варьировать лишь промежуточный, то будут получаться уникальные n-граммы. В путях все узлы получаются уникальными (нельзя составить n-грамму, в которой один и тот же узел фигурирует более одного раза).

Процесс факторизации проиллюстрирован на рис. 4.4.1. Зеленым цветом выделены триграммы, оранжевым — биграммы и фиолетовым — униграммы.

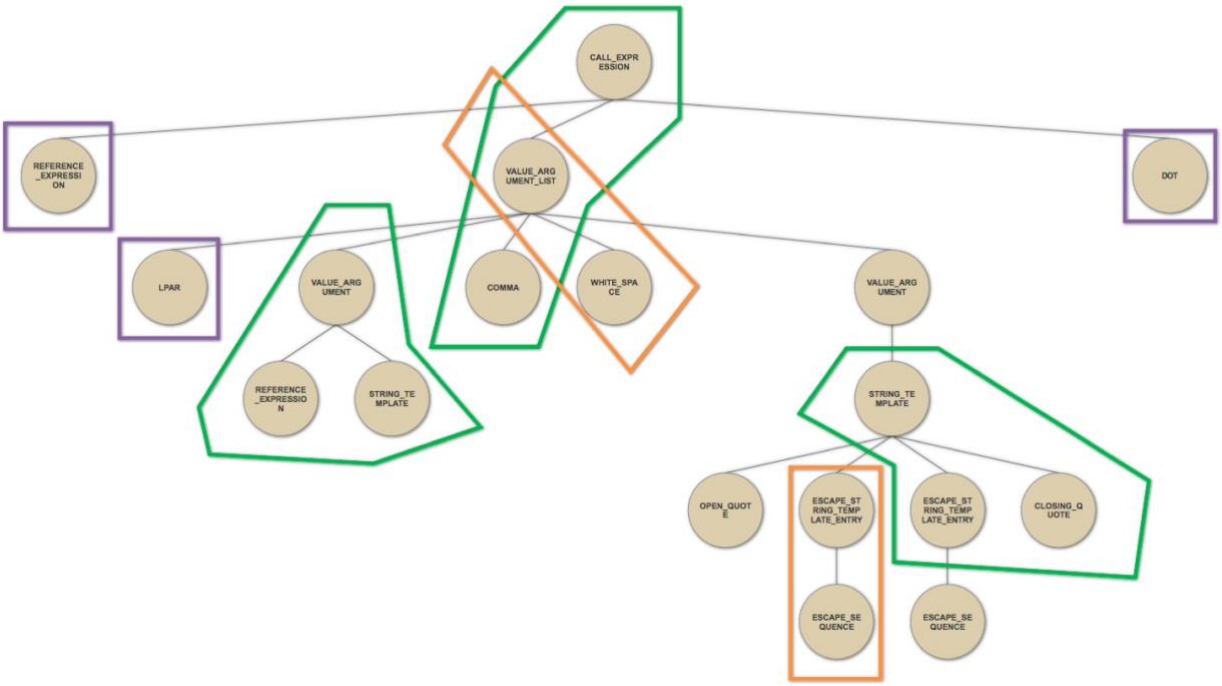

Рисунок 4.4.1. Демонстрация процесса факторизации дерева разбора.

Файлы с набором JVM-инструкций были факторизованы аналогичным образом. Алгоритм состоял в следующем: от начала списка инструкций до конца осуществлялось перемещение области в три узла, в рамках которой всегда генерировалась одна 3-грамма, три bi-граммы и три uni-граммы (данные числа соответствуют количеству сочетаний без повторений в рамках перемещающейся области).

Процесс факторизации проиллюстрирован на рис. 4.4.2 Смысл цветового выделения аналогичен смыслу на рисунке с демонстрацией факторизации дерева разбора.

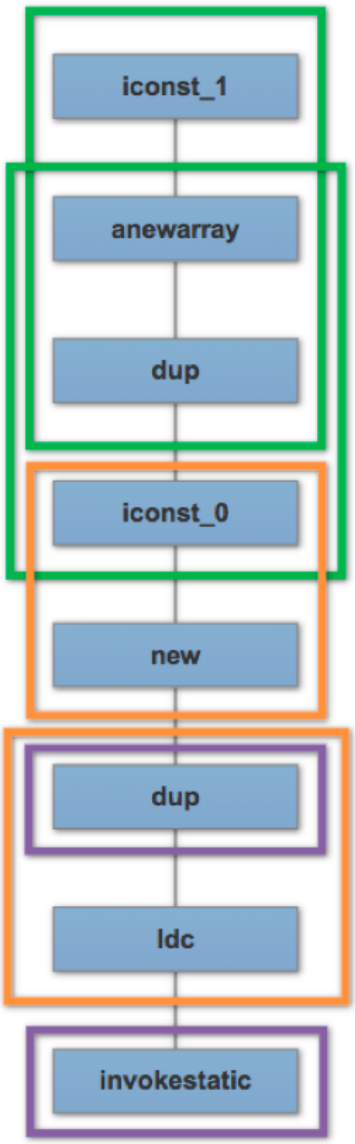

Рисунок 4.4.2. Демонстрация процесса факторизации байт-кода.

4.5 Отбор n-gram

Поскольку после факторизации байт-кода получилось слишком большое количество n-грамм (на которое предположительно не хватит ресурсов при работе автоэнкодера), встает задача отбора n-грамм.

Для отбора n-gram был написан инструмент[9], который позволяет отсекать те n-граммы, которые предположительно несут наименьший информационный вклад: наиболее частотные и наименее частотные.

4.6 Преобразование данных в разреженный вид

Перед запуском автоэнкодера данные с сопоставлением n-грамм их счетчикам необходимо преобразовать в разреженный вид: привести все примеры к одному количеству измерений (добавить n-граммы с нулевым счетчиком встречаемости). Также на данном этапе предполагается составление списка n-грамм, порядок их следования в котором будет совпадать с порядком следования счетчиков встречаемости в файлах, соответствующих примерам.

Для решения данной задачи был разработан инструмент[10], принимающих на вход путь к директории с факторизованными деревьями разбора или байт-кодом и записывающий в другую указанную директорию по аналогичному пути разреженное представление векторов со счетчиками встречаемости. Также инструмент предполагает по мере преобразования данных формирование списка n-грамм с порядком следования, равным порядку следования их счетчиков встречаемости в файлах с векторами, соответствующими примерам. Таким образом, программа записывает в текущую директорию файл all_ngrams.json.

[9] https://github.com/PetukhovVictor/ngram-selector
[10] https://github.com/PetukhovVictor/tree-set2matrix

4.7 Составление конечного набора данных

Наконец, встает задача объединения всех векторов в единый набор данных в формате, с которым сможет работать автоэнкодер. Для решения данной задачи также был разработан инструмент, объединяющий все вектора в единый набор данных в формате CSV.

4.8 Запуск автоэнкодера и обработка результатов

Для запуска автоэнкодера и обработки результатов (формирование списка аномалий на основе отклонения от среднего) был также разработан инструмент. Инструмент опубликован на GitHub[11].

[11] https://github.com/PetukhovVictor/anomaly-detection

ГЛАВА 5. ЭКСПЕРИМЕНТ

5.1 Конфигурация оборудования

Эксперимент проводился на машине, обладающей 16 GB оперативной памяти LPDDR3 и доступными дополнительно 65 GB виртуальной памяти, с процессором Intel Core i7 2800 МГц. Также дополнительно было доступно 2 GB графической памяти GDDR5 (видеокарта AMD Radeon Pro 555).

5.2 Сбор данных

Из списка в 47171 репозиториев каждый был склонирован. С помощью Github API были получены ссылки на последний релиз склонированных репозиториев, файлы сборок были также загружены. После фильтрации всех нерелевантных файлов размер набора данных составил 20GB.

Итого было получено 930 тыс. файлов с исходным кодом на Kotlin. Файлы готовых сборок подвергались фильтрации и сопоставлению файлам с исходным кодом на лету.

При пофайловом сборе (для небольшого набора данных) было собрано 40 тыс. файлов с исходным кодом на Kotlin и около 8 тыс. файлов с байт-кодом.

5.3 Сопоставление и фильтрация файлов

По результатам сопоставления в директории каждого репозитория был сформирован json-файл, в котором были перечислены для всех файлов с байт-кодом соответствующие им файлы с исходным кодом на Kotlin.

5.4 Разбор собранных файлов

После выполнения этапа разбора файлов был получен набор деревьев разбора PSI, json-файлы с JVM инструкциями, сгруппированные по методам, а также файлы с сопоставлением файлов с байт-кодом файлам с исходным кодом на Kotlin.

5.5 Факторизация деревьев разбора и байт-кода

В результате факторизации деревьев разбора на большом наборе данных было извлечено 11 тыс. n-грамм (из них униграмм — 250, биграмм — 1773, триграмм — 9068), на малом наборе данных было извлечено 5 тыс. n-gram.

В результате факторизации байт-кода по большого набору данных было извлечено 111 тыс. n-грамм (из них униграмм — 197, биграмм — 9382, триграмм — 101256), по малому набору данных — 79 тыс. n-грамм.

5.6 Отбор n-грамм

По результатам этапа отбора по большому набору данных с байт-кодом было отобрано 25 тыс. n-грамм, по малому набору данных — 2.8 тыс. n-грамм.

5.7 Составление конечного набора данных

Так как для большого набора данных после загрузки в память (перед запуском автоэнкодера) такой набор данных занимал более 80 GB, что является недопустимым из-за существующих ограничений в использовании памяти, было принято решение записать сформированный список векторов (массив массивов) в бинарный файл с последующим его использованием напрямую с диска, без загрузки в оперативную память.

Таким образом, по окончании данного этапа было получено для каждого набора данных (соответствующих деревьям разбора и байт-коду) два бинарных файла: с тестовой выборкой и обучающей (составленной из случайных примеров тестовой выборки, составляющих 10% от неё).

Файлы для малого набора данных подавались на вход автоэнкодеру в формате CSV и предварительно загружались в оперативную память.

5.8 Запуск автоэнкодера

На полученных наборах данных по деревьям разбора был запущен автоэнкодер в соответствии с составленной ранее моделью (см. раздел 3.2.6) со следующими параметрами:

1. количество эпох (epochs) — 5;
2. размер партии (batch size) — 1024 для большого набора данных, 64 — для малого.

Выбор данных параметров был обусловлен существующими ограничениями на используемую память (про которые было рассказано в разделе 5.1).

Время обучения для большого набора данных составило около 8 часов, время трансформации — 1.5 часа; для малого набора данных — 20 минут на обучение и 7 минут на трансформацию.

На полученных наборах данных по байт-коду автоэнкодер был запущен с таким же количество эпох, но с batch size, равным 128 (как для большого набора данных, так и для малого; был уменьшен пропорционально размеру набора данных). Для большого набора данных время обучения составило около 3 часов, время трансформации — 15 минут; для малого набора данных: 2 часа и 10 минут соответственно.

5.9 Получение результатов

По окончании работы автоэнкодера был произведен расчет евклидова расстояния между входными и выходными векторами автоэнкодера. Данное расстояние будет трактоваться как степень аномальности.

Таким образом, по окончании работы программы, включающей запуск автоэнкодера и расчет расстояний между векторами, был получен файл distances.json с набором расстояний, соответствующим примерам.

На рис. 5.9.1 в виде гистограммы изображены временные затраты на этапы преобразования и анализа данных для большого набора данных.

Также, для консолидации всех этапов от сбора кода до обнаружения аномалий был разработан единый инструмент[22].

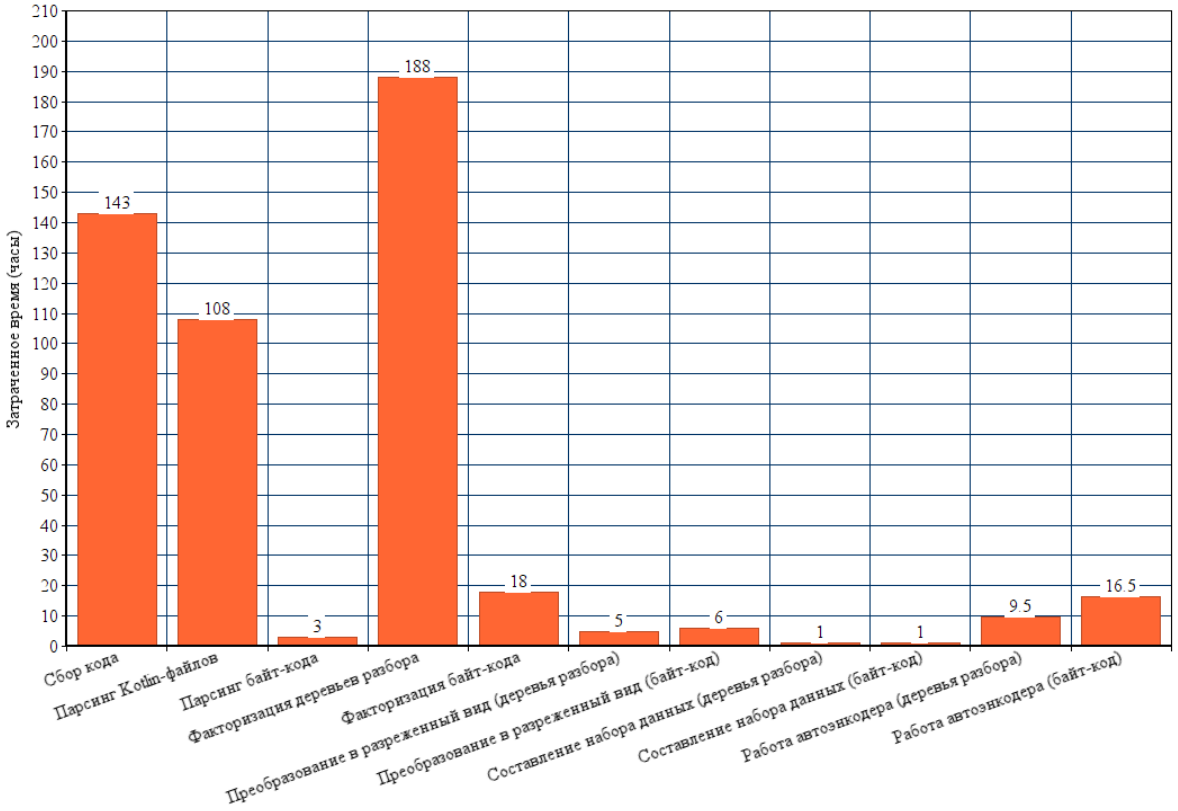

Рисунок 5.9.1. Временные затраты в процессе преобразования и анализа большого набора данных.

[22] https://github.com/PetukhovVictor/code-anomaly-detection

ГЛАВА 6. РАЗБОР И АНАЛИЗ РЕЗУЛЬТАТОВ

6.1 Разбор набора расстояний

Для дальнейшего рассмотрения из полученного набора расстояний была произведена выборка. Были выбраны только те примеры, соответствующие расстояния по которым отклонялись более чем на 3 среднеквадратических отклонения. Данные примеры были условно названы «аномалиями». Для данного отбора был написал соответствующий инструмент[23] с возможностью задания произвольного требуемого минимального отклонения.

Таким образом, для большого набора данных было получено 4924 аномалий по дереву разбора и 456 аномалий по байт-коду, для малого: 362 аномалии по дереву разбора и 151 аномалия по байт-коду. Набор расстояний был сопоставлен исходным файлам с помощью написанного скрипта.

6.2 Классификация аномалий для малого набора данных

Полученный набор файлов-аномалий для малого набора данных с исходных кодом и байт-кодом был вручную классифицирован. Было сформировано 30 классов (по 513 аномалиями). Классы и количество аномалий в них приведены в таблице 6.2.1.

Таблица 6.2.1 — количественная статистика аномалий по классам.

Название класса	**Количество аномалий по дереву разбора**	**Количество аномалий по байт-коду**	**Общее количество аномалий**
Большие и/или сложные перечисления	0	7	7
Большая иерархия в коде	30	6	36

[23] https://github.com/PetukhovVictor/anomaly-detection

Название класса	**Количество аномалий по дереву разбора**	**Количество аномалий по байт-коду**	**Общее количество аномалий**
Большие объекты-компаньоны	0	2	2
Большой набор констант	0	22	22
Большое количество инструкций в методе-инициализаторе в байт-коде	0	2	2
Большие методы	0	3	3
Большие многострочные переменные типа «строка»	27	14	41
Большие статически заданные массивы или отображения	30	3	33
Сложные и/или длинные логические выражения	6	0	6
Длинные цепочки вызовов методов	5	2	7
Длинные перечисления	2	0	2
Большое количество выражений присваивания	57	9	66

Большое количество веток в конструкциях when	34	3	37
Название класса	**Количество аномалий по дереву разбора**	**Количество аномалий по байт-коду**	**Общее количество аномалий**
Много конкатенаций строк	13	0	13
Много сложных арифметических выражений	0	1	1
Много свойств-делегатов	0	2	2
Много аргументов функций	17	8	25
Много параметров generic-классов	6	0	6
Много условных операторов	38	5	43
Много встроенных (inline) функций	0	3	3
Много литеральных строк	0	4	4
Много циклов	8	1	9
Много вложенных структур	0	2	2

Много операторов приведения к not-nullable типу	0	1	1
Название класса	**Количество аномалий по дереву разбора**	**Количество аномалий по байт-коду**	**Общее количество аномалий**
Много корневых функций	0	4	4
Много безопасных вызовов	0	1	1
Много похожих вызовов методов или функций	86	42	128
Много аннотаций, заданных квадратными скобками	0	3	3
Много параметров generic-классов с ключевым словом reified	0	1	1
Много вложенных вызовов	3	0	3

Из 30 в 25 классах присутствовали аномалии по байт-коду, в 15 классах присутствовали аномалии по дереву разбора, в 10 классах присутствовали аномалии как по дереву разбора, так и по байт-коду. Большинство файлов-аномалий (384 из 513 — 75%) распределились по семи классам:

- большая иерархия в коде,
- много похожих вызовов методов или функций,

- большие многострочные переменные типа «строка»,
- большие статически заданные массивы или отображения,
- большое количество выражений присваивания,
- большое количество веток в конструкциях when,
- много условных операторов.

Данная статистика показывает, какие проблемы чаще всего бывают в исходном коде программ на Kotlin.

Предполагается, что примеры-аномалии по дереву разбора являются представителями случаев неправильного использования языка, и будут проанализированы разработчиками языка Kotlin на предмет потенциальных улучшений дизайна конструкций языка или разработки новых конструкций (на основе анализа задач пользователей, которые они хотели решить с помощью кода, который оказался аномальным). Также на данных примерах предполагается тестирование компилятора на производительность. Некоторые из примеров являются, своего рода, экстремальными для компилятора, и будет резонно замерять производительность именно на таких примерах.

Для просмотра файлов-аномалий по типам и классам был разработан сайт[24] и инструмент для публикации новых аномалий и новых классов[25].

6.3 Сбор экспертных оценок по аномалиям и их классам для малого набора данных

Для сбора экспертных оценок предварительно на разработанном сайте были выделены лишь наиболее показательные аномалии-представители классов (из группы похожих аномалий было отобрано лишь по одному экземпляру; были исключены аномалии, в которых особенность класса была

[24] http://victor.am/code-anomaly-detection/
[25] https://github.com/PetukhovVictor/kotlin-code-anomalies-publisher

выражена в меньшей степени). Таким образом, было отобрано 103 аномалии. Из них 48 — по дереву разбора PSI, 55 — по байт-коду.

Была разработана функциональность на сайте для сбора оценок как по классам, так и по конкретным аномалиям по пятибалльной шкале.

Оценки выставлялись разработчиками компилятора Kotlin, наиболее близко знакомыми со всем многообразием конструкций языка, устройством кодогенерации по ним и её возможным влиянием на конечную производительность.

Таким образом, по всем классам и аномалиям были собраны оценки. Средняя оценка по классам получилась равной 2.462, по аномалиям — 2.786 (по аномалиям на дереве разбора —3.083, по аномалиям на байт-коде — 2.527). В таблице 6.3.1 представлены экспертные оценки по классам, на рис. 6.3.2 — по аномалиям (также, на данной гистограмме показано, какую долю составляют аномалии по дереву разбора, а какую — по байт-коду).

Таблица 6.3.1 — перечень экспертных оценок классов аномалий.

Оценка	**Название класса**
5	Many case in when
	Many delegate properties
	Many generic parameters
4	Big and complex enums
	Big constants set
	Long calls chain
	Long enumerations
	Many if statements
	Many root function definitions

	Many similar call expressions
	Many square bracket annotations
	Nested calls
3	Big code hierarchy
	Big methods
	Big multiline strings
	Many assignment statements
	Many consecutive arithmetic expressions
	Many function arguments
2	Big static arrays or map
	Complex or long logical expressions
	Many concatenations
	Many inline functions
	Many loops
	Many not null assertion operators
1	Big companion object
	Big init method in bytecode
	Many literal strings
	Many nested structures
	Many safe calls

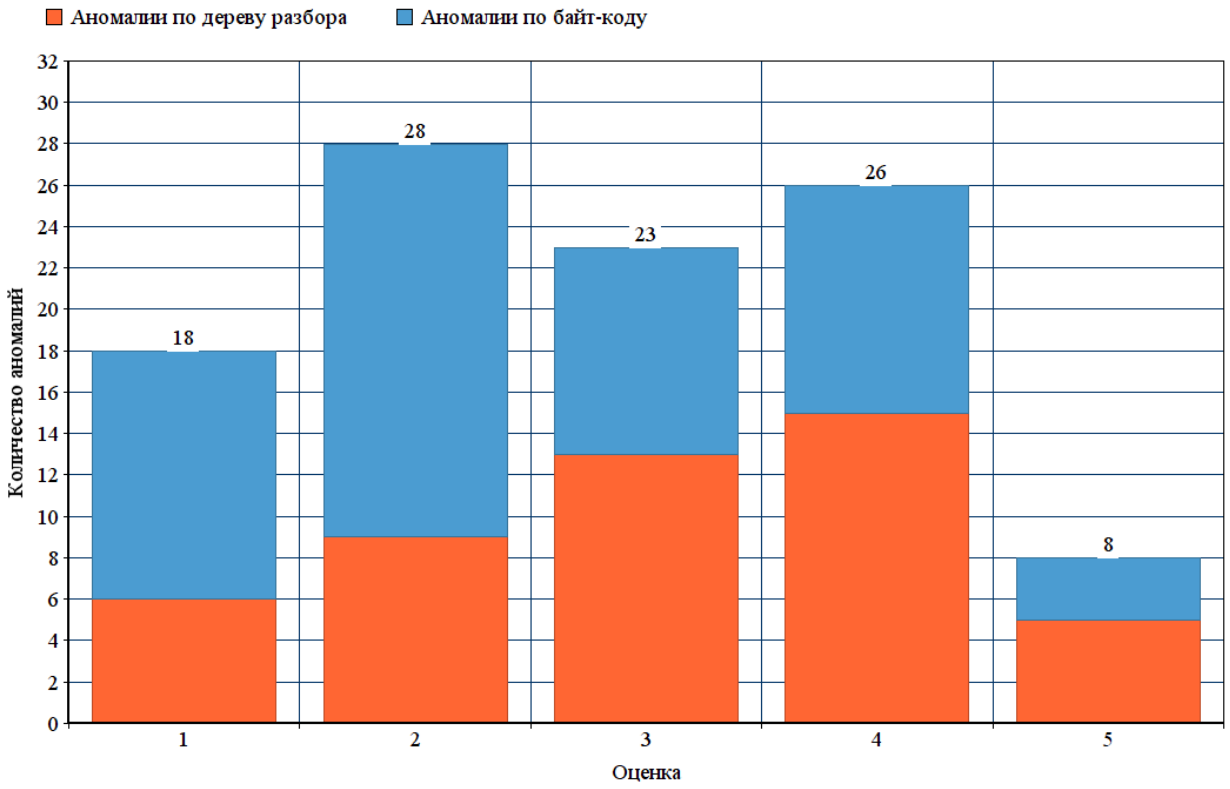

Рисунок 6.3.2. Экспертные оценки аномалий.

В Приложении А и Приложении Б приведены примеры аномалий из разных классов, получивших оценки 5.

6.4 Использование результатов исследования по малому набору данных

По результатам исследования по малому набору данных примеры-аномалии с оценками 4 и 5 были включены в тесты на производительность компилятора Kotlin: по окончании разработки той или иной возможности компилятора будет происходить запуск данных тестов на найденных аномалиях, где определяется, не повлекла ли разработка ухудшение производительности.

6.5 Разбор результатов по большому набору данных

По большому набору данных предполагалось, что будет произведен поиск относительных аномалий: аномалий на дереве разбора PSI относительно байт-кода и наоборот.

Для решения такой задачи был разработан инструмент[26], который по файлу с найденными аномалиями по одном источнику (дереву разбора или байт-коду), набору расстояний между входными и выходными векторами автоэнкодера по другому источнику копировал примеры-аномалии в указанную папку, отсортированные по разнице между «числом аномальности» дерева разбора и байт-кода. «Числа аномальности» в свою очередь были предварительно нормированы по максимальным значениям. Таким образом, «числа аномальности» находились в интервале $[-1; 1]$.

Каждому набору аномалий (по дереву разбора и байт-коду) был сопоставлен набор соответствующих примеров по другому источнику и были вычислены разности в «числах аномальности» примеров. Трактовка данных «чисел аномальности» была приведена в разделе 1.2 настоящей работы (условным аномалиям соответствуют пункты 2 и 3 в списке случаев, потенциально интересных разработчикам Kotlin). Агрегирующие значения приведены в таблице 6.5.1.

Таблица 6.5.1 — агрегирующие значения по разностям «чисел аномальности» между разными источниками данных.

Источник данных	**Среднее арифметическое**	**Среднеквадратическое отклонение**	**Медиана**
Дерево разбора относительно байт-кода	0.0186	0.1574	−0.0113
Байт-код относительно дерева разбора	0.0364	0.0821	0.0256

[26] https://github.com/PetukhovVictor/bytecode-anomalies-source-finder

Из данных аномалий были отобраны те, где вычисленные разности превосходили $1/4$ СКО. Итого было получено 38 условных аномалий: 6 аномалий по байт-коду относительно примеров по дереву разбора и 32 аномалии по дереву разбора относительно байт-кода.

Разности у 2 из 6 аномалий по байт-коду смещены в большую сторону от среднего, у остальных 4 — в меньшую. У 14 аномалий по дереву разбора разности смещены в большую сторону от среднего, у остальных 18 — в меньшую.

В Приложении В (исходный код) и Приложении Г (байт-код) приведен пример файла-аномалии, «число аномальности» по байт-коду которой сильно превышают «число аномальности» по дереву разбора.

6.6 Использование результатов исследования по большому набору данных

Для каждого из случаев, потенциально интересных разработчикам Kotlin, перечисленных в разделе 1.2 настоящей работы, были найдены примеры аномалий. Найденные примеры были переданы разработчикам компилятора Kotlin для детального изучения: для анализа корректности кодогенерации и отработавших оптимизаций.

ЗАКЛЮЧЕНИЕ

По окончании проделанной работы можно отметить следующие результаты.

Было разработано решение, включающее инструменты для выполнения следующий задач:

1. сбор исходного кода и байт-кодом с ресурса GitHub;
2. сопоставление и парсинг кода;
3. факторизация деревьев разбора и байт-кода путем разложения на n-граммы;
4. запуск автоэнкодера на подготовленных наборах данных;
5. отбор примеров и формирование условных аномалий;
6. оценивание предоставленных аномалий и их классов (в виде веб-сайта[27]).

С использованием разработанного инструментария было проведено два исследования: на малом и большом наборе данных.

По результатам первого исследования было найдено 513 примеров-аномалий, они были классифицированы и переданы разработчикам Kotlin для экспертного оценивания. 34 из 103 аномалий получили высокие оценки (4 и 5) и были включены в тесты на производительность компилятора Kotlin.

По результатам второго исследования было получено 5380 аномалий, на основе которых в соответствии с разностями «чисел аномальности» было сформировано 38 условных аномалий, которые были переданы на детальное изучение также разработчикам компилятора Kotlin.

На основе полученных оценок по результатам первого исследования можно сделать вывод о том, что предложенный подход можно успешно

[27] http://victor.am/code-anomaly-detection/

использовать для поиска аномалий как по исходному коду на Kotlin, так и по соответствующему сгенерированному байт-коду. А факт того, что аномалии с высокими оценками были включены в тесты на производительность компилятора Kotlin говорит о том, что данные аномалии, полученные с помощью предложенного подхода, помогут в будущем отслеживать производительность скомпилированных с помощью компилятора Kotlin программ, и, таким образом, внесут свой вклад в улучшение его качества.

Результаты же второго исследования должны будут также помочь в выявлении проблем производительности компилятора — но уже не посредством включения их в тесты на производительность, а посредством детального анализа работы кодогенератора и оптимизатора компилятора Kotlin на коде найденных примеров.

В качестве направления для дальнейшего исследования можно предложить автоматизацию классификации найденных аномалий (что позволит сконцентрироваться на обнаружении новых классов), автоматическую компиляцию проектов для получения большего количества байт-кода для соответствующего исходного кода и улучшение механизма отбора признаков (для отбора ещё более релевантных признаков).

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Tracking down software bugs using automatic anomaly detection / Sudheendra Hangal, Monica S. Lam // ICSE '02 Proceedings of the 24th International Conference on Software Engineering — P. 291-301 — URL: https://dl.acm.org/citation.cfm?id=581377.

[2] Graph-based mining of multiple object usage patterns / Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, Tien N. Nguyen // ESEC/FSE '09 Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering — P. 383-392 — URL: https://dl.acm.org/citation.cfm?id=1595767.

[3] Detecting object usage anomalies / Andrzej Wasylkowski, Andreas Zeller, Christian Lindig // ESEC-FSE '07 Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering — P. 35-44 — URL: https://dl.acm.org/citation.cfm?id=1287632.

[4] A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors / R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni // SP '01 Proceedings of the 2001 IEEE Symposium on Security and Privacy — P. 144 — URL: https://dl.acm.org/citation.cfm?id=884433.

[5] Anomaly Detection Using Call Stack Information / Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, Weibo Gong // SP '03 Proceedings of the 2003 IEEE Symposium on Security and Privacy — P. 62 — URL: https://dl.acm.org/citation.cfm?id=830554.

[6] Альфред В. Ахо. Компиляторы. Принципы, технологии и инструментарий [Текст] / Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман — Москва: Вильямс, 2016. — 1184 с

[7] The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition / Y.N. Srikant, Priti Shankar // CRC Press — 2007

[8] Convolutional neural networks over tree structures for programming language processing / Lili Mou, Ge Li, Lu Zhang, Tao Wang, Zhi Jin // AAAI'16 Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence — P. 1287-1293 — URL: https://dl.acm.org/citation.cfm?id=3016002.

[9] Building Program Vector Representations for Deep Learning / Lili Mou, Ge Li, Yuxuan Liu, Hao Peng, Zhi Jin, Yan Xu, Lu Zhang // ICSE'14 Proceedings of the 36th International Conference on Software Engineering — arXiv preprint arXiv:1409.3358 — URL: https://arxiv.org/abs/1409.3358.

[10] DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones / Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, Stephane Glondu // ICSE '07 Proceedings of the 29th international conference on Software Engineering — P. 96-105 — URL: https://dl.acm.org/citation.cfm?id=1248843.

[11] De-anonymizing programmers via code stylometry / Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, Rachel Greenstadt // SEC'15 Proceedings of the 24th USENIX Conference on Security Symposium — P. 255-270 — URL: https://dl.acm.org/citation.cfm?id=2831160.

[12] Program Structure Interface (PSI) // IntelliJ Platform SDK Developer Guide — URL: http://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html (online; accessed: 11.11.2017).

[13] Kotlin Language Documentation // Kotlin programming language official site — URL: http://kotlinlang.org/docs/kotlin-docs.pdf (online; accessed: 22.02.2018).

[14] Kotlin 1.2 Released: Sharing Code between Platforms // Kotlin Blog — URL: https://blog.jetbrains.com/kotlin/2017/11/kotlin-1-2-released/ (online; accessed: 12.03.2018).

[15] Building Autoencoders in Keras // The Keras Blog — URL: https://blog.keras.io/building-autoencoders-in-keras.html (online; accessed: 17.01.2018).

[16] Scikit-learn API // Keras Documentation — URL: https://keras.io/scikit-learn-api/ (online; accessed: 22.01.2018).

[17] Novelty and Outlier Detection // scikit-learn — URL: http://scikit-learn.org/stable/modules/outlier_detection.html (online; accessed: 30.01.2018).

[18] The Overview Of Anomaly Detection Methods in Data Streams / Viacheslav Shkodyrev, Kamil Yagafarov, Valentina Bashtovenko, Ekaterina Ilyina // The Second Conference on Software Engineering and Information Management, vol. 1864, 2017.

[19] One class support vector machine for anomaly detection in the communication network performance data / Rui Zhang, Shaoyan Zhang, Sethuraman Muthuraman, Jianmin Jiang // ELECTROSCIENCE'07 Proceedings of the 5th conference on Applied electromagnetics, wireless and optical communications — P. 31-37 — URL: https://dl.acm.org/citation.cfm?id=1503549.1503556.

[20] Data Mining and Knowledge Discovery Handbook — P. 321-352 / Oded Maimon, Lior Rokach // Springer, Boston, MA — 2005.

[21] Statistical Anomaly Detection Technique for Real Time Datasets / Y.A.Siva Prasad, Dr.G.Rama Krishna // International Journal of Computer Trends and Technology (IJCTT), vol. 6 number 2, 2013 — URL: http://www.ijcttjournal.org/Volume6/number-2/IJCTT-V6N2P114.pdf.

[22] Anomaly Detection Using Replicator Neural Networks Trained on Examples of One Class / Ru Hoang Anh Dau, Vic Ciesielski, Andy Song // SEAL 2014 Proceedings of the 10th International Conference on Simulated Evolution and Learning, vol. 8886 — P. 311-322 — URL: https://dl.acm.org/citation.cfm?id=2966216.

[23] How many hidden units should I use? // comp.ai.neural-nets FAQ, Part 3 of 7: Generalization — URL: http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-10.html (online; accessed: 11.04.2018).

[24] Android Oreo adds support for the Kotlin programming language // Android Central — URL: https://www.androidcentral.com/android-o-adds-support-kotlin-programming-language (online; accessed: 02.04.2018).

ПРИЛОЖЕНИЕ А

Пример аномалии класса «Длинные перечисления» по дереву разбора 1 (исходный код)

```
@Suppress("PLATFORM_CLASS_MAPPED_TO_KOTLIN")
    override fun predicate(fromType: Type, toType: Type): Boolean {
        val fromErased = fromType.erasedType()
        return when {
            String::class.isAssignableFrom(fromType) -> when (toType) {
                String::class.java,
                Short::class.java, java.lang.Short::class.java,
                Byte::class.java, java.lang.Byte::class.java,
                Int::class.java, Integer::class.java,
                Long::class.java, java.lang.Long::class.java,
                Double::class.java, java.lang.Double::class.java,
                Float::class.java, java.lang.Float::class.java,
                BigDecimal::class.java,
                BigInteger::class.java,
                CharSequence::class.java,
                ByteArray::class.java,
                Boolean::class.java, java.lang.Boolean::class.java,
                File::class.java,
                URL::class.java,
                URI::class.java -> true
                else -> {
                    val toErased = toType.erasedType()
                    toErased.isEnum
                }
            }
            CharSequence::class.isAssignableFrom(fromType) -> when (toType) {
                CharSequence::class.java,
                String::class.java,
                ByteArray::class.java -> true
                else -> false
            }
            Number::class.isAssignableFrom(fromType) -> when (toType) {
                Short::class.java, java.lang.Short::class.java,
                Byte::class.java, java.lang.Byte::class.java,
                Int::class.java, Integer::class.java,
            (and 9 more similar when statements)
```

Рисунок А.1 – исходный код одной из найденных аномалий по дереву разбора с оценкой 5.

ПРИЛОЖЕНИЕ Б

Пример аномалии класса «Много параметров generic-классов» по дереву разбора (исходный код)

```
@Suppress("UNCHECKED_CAST")
fun <V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13, V14, V15, V16, V17, V18, V19, V20, E> concreteCombine(p1: Promise<V1, E>,
                                                                                                                   p2: Promise<V2, E>,
                                                                                                                   p3: Promise<V3, E>,
                                                                                                                   p4: Promise<V4, E>,
                                                                                                                   p5: Promise<V5, E>,
                                                                                                                   p6: Promise<V6, E>,
                                                                                                                   p7: Promise<V7, E>,
                                                                                                                   p8: Promise<V8, E>,
                                                                                                                   p9: Promise<V9, E>,
                                                                                                                   p10: Promise<V10, E>,
                                                                                                                   p11: Promise<V11, E>,
                                                                                                                   p12: Promise<V12, E>,
                                                                                                                   p13: Promise<V13, E>,
                                                                                                                   p14: Promise<V14, E>,
                                                                                                                   p15: Promise<V15, E>,
                                                                                                                   p16: Promise<V16, E>,
                                                                                                                   p17: Promise<V17, E>,
                                                                                                                   p18: Promise<V18, E>,
                                                                                                                   p19: Promise<V19, E>,
                                                                                                                   p20: Promise<V20, E>):Promise<Tuple20<V1, ... (and 19 similar parameters)
    val deferred = deferred<Tuple20<V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13, V14, V15, V16, V17, V18, V19, V20>, E>()

    val results = AtomicReferenceArray<Any?>(20)
    val successCount = AtomicInteger(20)

    fun createTuple(): Tuple20<V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13, V14, V15, V16, V17, V18, V19, V20> {
        return Tuple20(
                results.get(0) as V1,
                results.get(1) as V2,
                (and 19 similar expressions)
```

Рисунок Б.1 – исходный код одной из найденных аномалий по дереву разбора с оценкой 5.

ПРИЛОЖЕНИЕ В

Пример условной аномалии по байт-коду 1 (исходный код)

```
class ConfigModel(val configs: Config) : ViewModel() {
    val ip = bind { configs.ipProperty }
    val dataBase = bind { configs.dataBaseProperty }
    val rootUser = bind { configs.rootUserProperty }
    val password = bind { configs.passwordProperty }
    val tableName = bind { configs.tableNameProperty }
    val entityName = bind { configs.entityNameProperty }
    val entityPackage = bind { configs.entityPackageProperty }
    val mapperPackage = bind { configs.mapperPackageProperty }
    val servicePackage = bind { configs.servicePackageProperty }
}
```

Рисунок В.1 – исходный код одной из найденных условных аномалий по байт-коду.

ПРИЛОЖЕНИЕ Г

Пример условной аномалии по байт-коду 1 (байт-код)

```
{
  "getIp" : [ "aload_0", "getfield", "areturn" ],
  "getDataBase" : [ "aload_0", "getfield", "areturn" ],
  "getRootUser" : [ "aload_0", "getfield", "areturn" ],
  "getPassword" : [ "aload_0", "getfield", "areturn" ],
  "getTableName" : [ "aload_0", "getfield", "areturn" ],
  "getEntityName" : [ "aload_0", "getfield", "areturn" ],
  "getEntityPackage" : [ "aload_0", "getfield", "areturn" ],
  "getMapperPackage" : [ "aload_0", "getfield", "areturn" ],
  "getServicePackage" : [ "aload_0", "getfield", "areturn" ],
  "getConfigs" : [ "aload_0", "getfield", "areturn" ],
  "<init>" : [ "aload_1", "ldc", "invokestatic", "aload_0", ... (4428 instructions)
}
```

Рисунок Г.1 – байт-код одной из найденных условных аномалий по байт-коду.