ОГЛАВЛЕНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ПО — программное обеспечение.

ЯП — язык программирования.

Целевая программа — программа, получающаяся в результате работы компилятора.

Целевой язык — язык, на котором составлена целевая программа.

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.

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.

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

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

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

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.

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 отсутствовать, либо быть совмещены):
    2.1. лексический анализ,
    2.2. синтаксический анализ,
    2.3. семантический анализ,
    2.4. оптимизации на абстрактном синтаксическом дереве,
    2.5. трансляция в промежуточное представление,
    2.6. машинно-независимые оптимизации,
    2.7. трансляция в конечное представление (машинный код),
    2.8. 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 целевой программы находится лишь одна 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, призванное помочь в контроле и улучшении производительности программ в рамках стадий написания кода и его компиляции.

Результаты такого исследования могут быть использованы разработчиками языка программирования, на котором эти программы составляются, для улучшений дизайна конструкций языка, возможной разработки новых конструкций, анализа и выявления проблем в работе механизмов генерации кода и оптимизаций.

В качестве языка программирования, программы на котором и компилятор которого будут исследоваться, был выбран ЯП Kotlin, разработанный компанией JetBrains, как один из наиболее молодых, быстро развивающихся и ещё недостаточно исследованных языков. Kotlin — это высокоуровневый язык программирования общего назначения, работающий 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, 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 с целью дальнейшего выявления потенциальных проблем производительности разработчиками Kotlin на основе результатов проведенного исследования.

Практическая значимость работы заключается в получении по результатам исследования и разработки сгруппированного 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, как уже было сказано, является анализ исходного кода на языке программирования Kotlin (и других представлений этого исходного кода) с целью обнаружения аномалий и дальнейшего поиска проблем производительности в них.

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.

К сожалению, 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 Постановка задачи

После освещения предметной области — того, как происходит компиляция кода на Kotlin в компиляторе, и в каких представлениях он может находиться, можно определить задачу для исследования и разработки.

Задачей будет являться анализ исходного кода на Kotlin в представлении в виде дерева разбора PSI, а также сгенерированного по нему JVM байт-кода. Такой анализ должен будет быть направлен на обнаружение аномалий. Аномалия является примером нетипичного кода по меркам анализируемого набора данных. Такие аномалии, по мнению разработчиков Kotlin, могут соответствовать проблемам производительности в программах, сгенерированных компилятором.

Предлагается провести исследование на некотором наборе данных: определить способы анализа, оптимальные параметры алгоритмов, обсудить полученные результаты с экспертами, получить результаты при некоторых заданных ограничениях и других параметрах алгоритмов.

Разработчикам Kotlin предполагается, что 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 (также 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 взаимодействие 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 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] [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.

Рассмотрев уже существующие работы в области обнаружения кодовых аномалий и отметив их недостатки и непригодность или лишь частичную пригодность для поставленной задачи, можно сделать вывод о необходимости разработки инструмента, который бы удовлетворял заявленным требованиям, и проведение соответствующего исследования.

ГЛАВА 3. ОБЗОР МЕТОДОВ И СРЕДСТВ РЕШЕНИЯ ЗАДАЧИ

3.1 Определение области решения задачи

Как уже было отмечено, объектом поиска будут являться файлы с исходным кодом на Kotlin, PSI дерево разбора и сгенерированный JVM байт-код которых, по тем или иным параметрам являются аномальными (нетипичными).

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 подразделяют на три вида:

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).

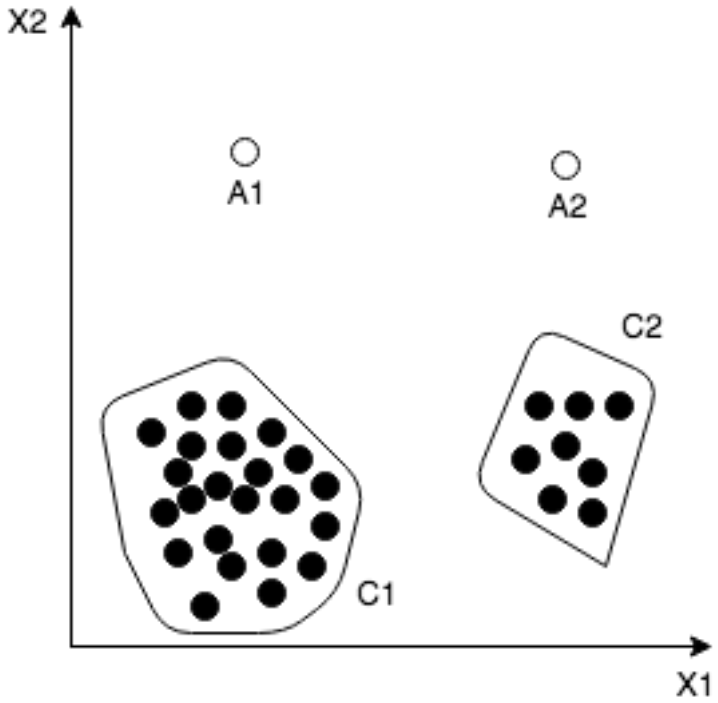

Рисунок 3.2.1 — демонстрация вида точечных аномалий.

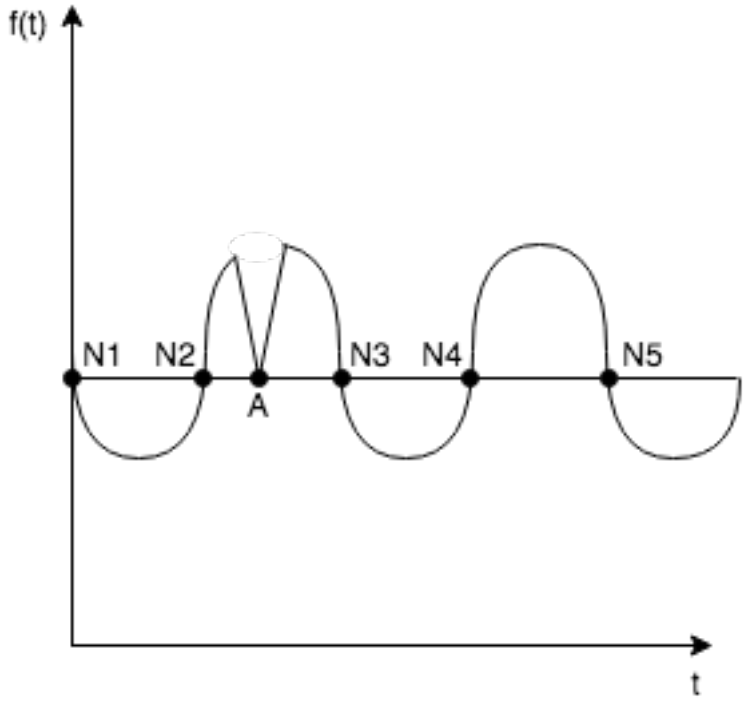

Рисунок 3.2.2 — демонстрация вида контекстуальных аномалий.

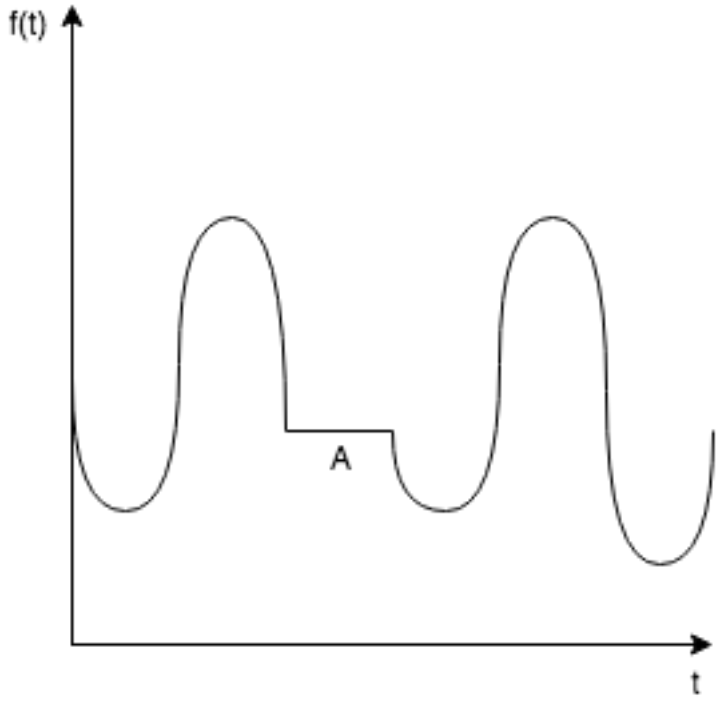

Рисунок 3.2.3 — демонстрация вида коллективных аномалий.

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).

Существующие методы обнаружения аномалий делят на две 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);
2. Обучение без учителя: подразумевается, что подавляющее большинство объектов являются нормальными, а в выборке производится поиск объектов, отличающихся по своим свойствам от нормальных.

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 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 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, также 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 задание 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 Методы, основанные на оценке плотности

В данных методах применительно к задаче поиска аномалий предполагается, что нормальные объекты данных (точки) встречаются вокруг плотной окрестности, а отклонения находятся далеко. Здесь также предполагается обучение без учителя. На выходе можно получить не просто факт принадлежности объекта к группе нормальных объектов или к группе

аномалий, но и численную оценку, показывающую близость принадлежности (отклонение).

Один из наиболее известных методов данной группы — метод k-ближайших соседей.

3.2.4 Статистические методы

Статистические методы 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.

3.2.5 Методы, основанные на применении репликаторных нейронных сетей

Методы данной группы 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 в 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.

3.2.6 Выбор метода для решения задачи обнаружения аномалий

Для решения задачи обнаружения аномалий был выбран автоэнкодер. Эта нейронная сеть по окончании 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.

Помимо этого, 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 (будет требоваться сокращение размерности).

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). 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 равен 2).

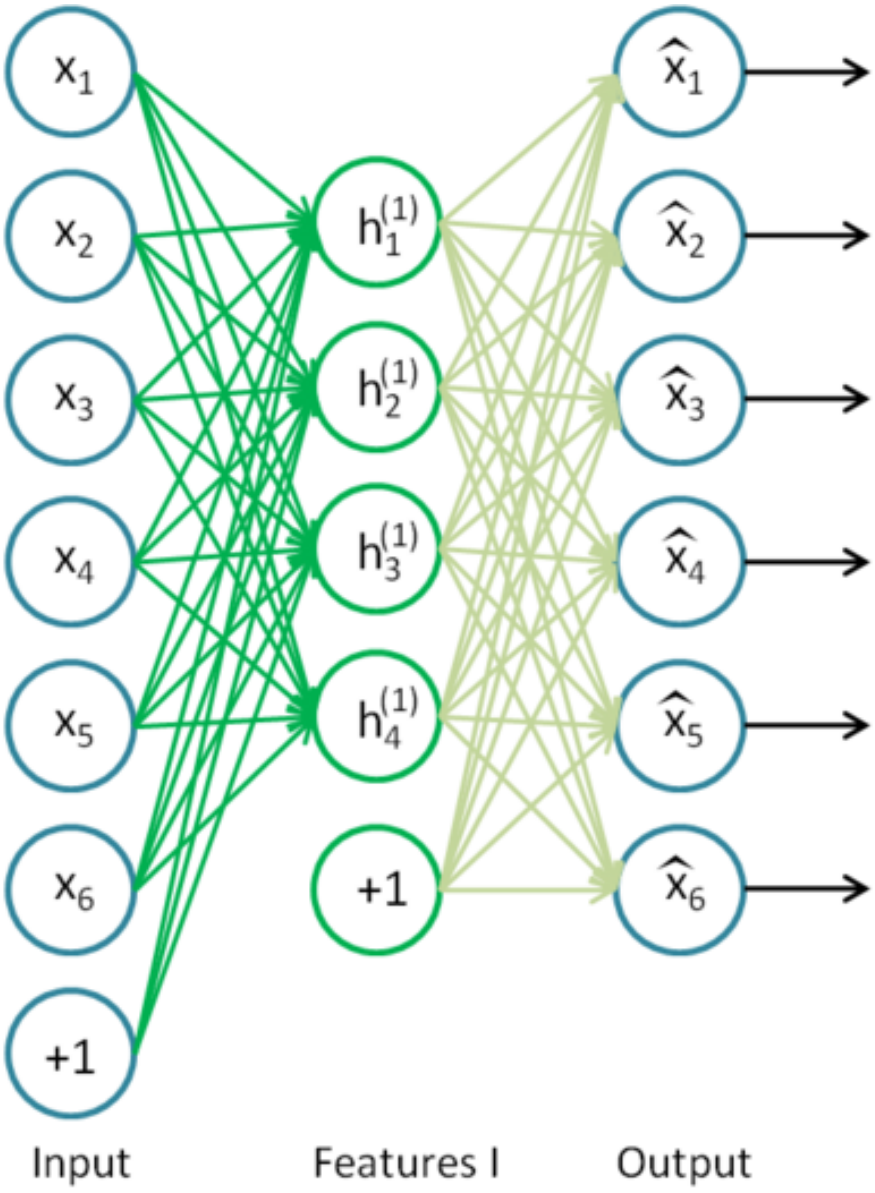

Рисунок 3.2.6.1 — схематичное изображение модели автоэнкодера.

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 неявный признаки, которые сложно интерпретировать и описать ручным образом (для n-грамм с $n \geq 2$), но которые могут внести существенный вклад в формирование аномалии.

После такой факторизации подразумевается, что будет получен набор векторов, компоненты которого соответствуют количествам встречаемости тех или иных n-грамм; а также список идентификаторов n-грамм, порядковые номера которых соответствуют порядковым номерам соответствующих компонентов векторов.

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. ЭКСПЕРИМЕНТ

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.

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. По предварительным подсчетам, всего Github содержит 47 тыс. репозиториев. Стоит отметить, что здесь нам доступны лишь репозитории, у которых Kotlin является основным языком (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 релизных файлов проекта. Будем осуществлять сбор данных релизных файлов для дальнейшего извлечения из них JVM байт-кода.

Для большого набора данных байт-код будем собирать из релизных файлов соответствующих репозиториев. Для малого — из релизных файлов 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.

4.1.1 Инструменты для сбора данных

Перед сбором непосредственно самого кода был предоставлен список всех репозиториев, основным языком которых является Kotlin.

Далее был разработан инструмент[1], который по файлу со списком репозиториев осуществлял скачивание данных репозиториев, а также релизных файлов, если они имелись у соответствующего репозитория.

Для пофайлового сбора кода (для малого набора данных) также были разработаны инструменты, первый из которых[2] путем обращения к API GitHub получает список файлов с кодом на Kotlin и сохраняет их в указанное место, второй[3] — собирает релизные файлы из доступных в соответствии с ограничением GitHub API репозиториев.

4.1.2 Результаты этапа сбора данных

В конечном счете для большого набора данных был сформирован список из 47171 репозиториев. Каждый из репозиториев в данном списке был склонирован. С помощью Github API были получены ссылки на последний релиз склонированных репозиториев, релизные файлы были также загружены. Суммарный размер всех склонированных репозиториев составил 200GB.

[1] https://github.com/PetukhovVictor/github-kotlin-repo-collector
[2] https://github.com/PetukhovVictor/github-kotlin-code-collector
[3] https://github.com/PetukhovVictor/github-kotlin-jar-collector

Итого было получено 930 тыс. файлов с исходным кодом на ЯП Kotlin. Релизные файлы 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.

4.2 Сопоставление и фильтрация и собранных файлов

Поскольку в загруженных релизных файлах репозиториев 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. релизные файлы 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.

4.2.1 Инструменты для сопоставления и фильтрации файлов

Для решения задачи сопоставления файлам с исходными кодами файлов с байт-кодом был разработан инструмент[4], который выполняет фильтрацию файлов репозитория (оставляя только файлы с исходным кодом Kotlin), а также сопоставление этих файлов с файлами с байт-кодом (и удалением не релевантных файлов).

Сопоставление файлов выполняется следующим образом: сначала осуществляется разбор файлов с JVM байт-кодом, из мета-информации которых достается название пакета и название файла, из которого был сгенерировал данный байт-код; далее происходит обход исходных кодов и с

[4] https://github.com/PetukhovVictor/bytecode-to-source-mapper

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.

4.2.2 Результаты этапа сопоставления и фильтрации файлов

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

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.

4.3.1 Инструменты для разбора файлов

Для парсинга исходных кодов на Kotlin был склонирован репозиторий компилятора Kotlin[5], 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[6], преобразующий файлы с байт-кодом в json-файлы (class-файлы), с JVM-инструкциями, сгруппированными по методам.

Таким образом, по результатам данных этапов был получен набор деревьев разбора PSI, json-файлы с JVM инструкциями, сгруппированными по

[5] https://github.com/PetukhovVictor/kotlin-academic
[6] https://github.com/PetukhovVictor/bytecode-parser

методам, а также файлы с сопоставлением файлов с байт-кодом файлам с исходным кодом на Kotlin.

4.4 Факторизация деревьев разбора и байт-кода

Для осуществления факторизации деревьев разбора PSI и JVM байт-кода был разработан инструмент[7], который в зависимости об объекта факторизации (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.

4.4.1 Факторизация дерева разбора PSI

Написанный инструмент в режиме «факторизация по дереву» осуществляет извлечение uni-грамм, bi-грамм и 3-грамм 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. 4.4.1.1.

[7] https://github.com/PetukhovVictor/ngram-generator

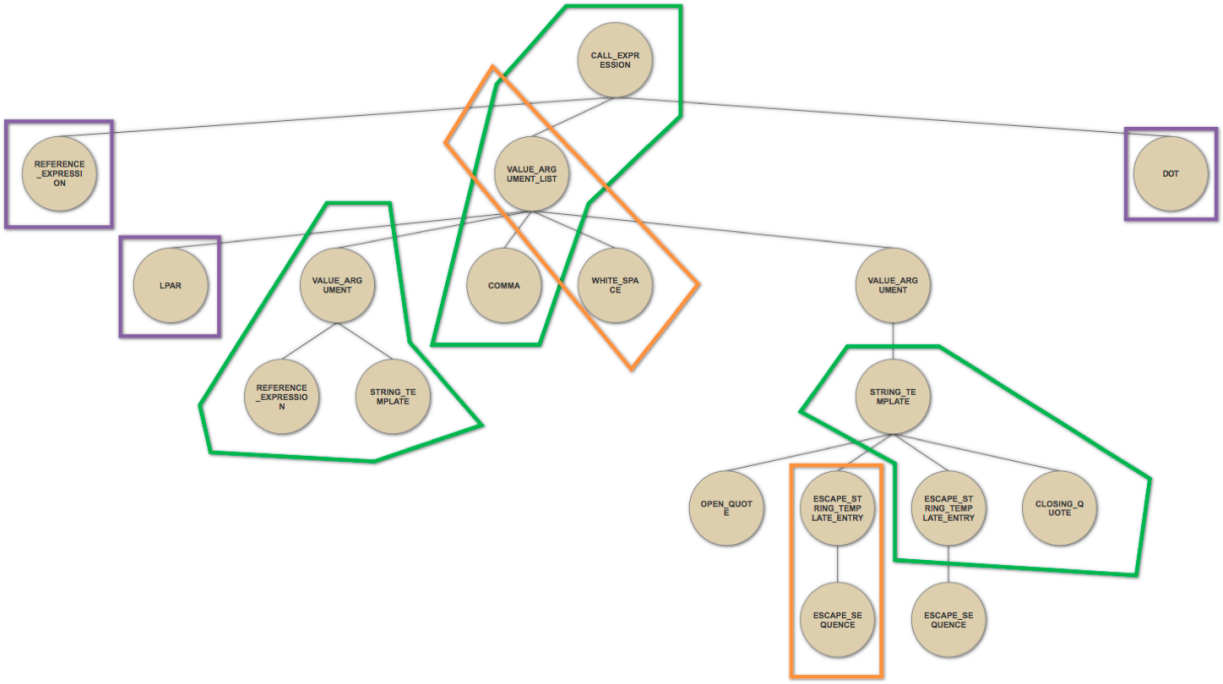

Рисунок 4.4.1.1 — демонстрация процесса факторизации дерева разбора.

В результате факторизации деревьев разбора на большом наборе данных было извлечено 11 тыс. n-грамм (из них uni-грамм — 250, bi-грамм — 1773, 3-грамм — 9068), на малом наборе данных было извлечено 5 тыс. n-gram.

4.4.2 Факторизация JVM байт-кода

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. 4.4.2.1.

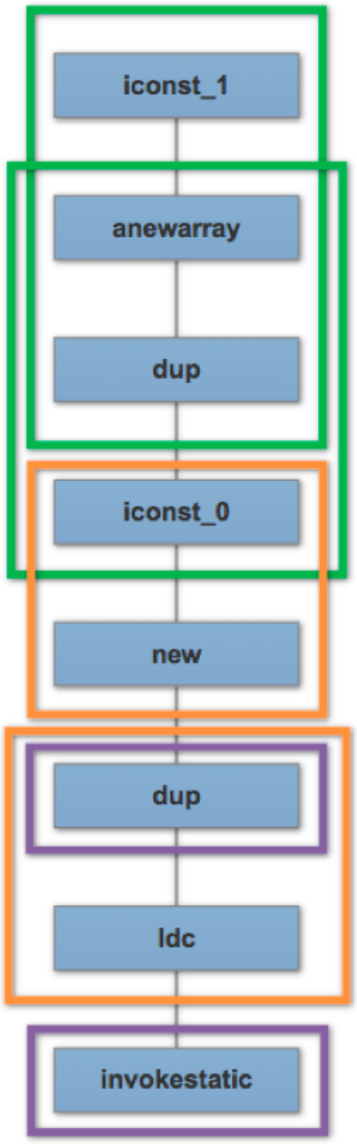

Рисунок 4.4.2.1 — демонстрация процесса факторизации байт-кода.

В результате факторизации байт-кода по большого набору данных было извлечено 111 тыс. n-грамм (из них uni-грамм — 197, bi-грамм — 9382, 3-грамм — 101256), по малому набору данных — 79 тыс. n-грамм.

4.4 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[8], который позволяет отсекать те n-граммы, которые предположительно несут наименьший информационный вклад: наиболее частотные и наименее частотные.

Таким образом, по большого набору данных с байт-кодом было отобрано 25 тыс. n-грамм, по малому набору данных — 2.8 тыс. n-грамм.

4.5 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[9], 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.

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

4.6 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.

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.

4.7 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.

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. Инструмент для запуска автоэнкодера был опубликован на GitHub[10].

4.8 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. 4.8.1 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[11].

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

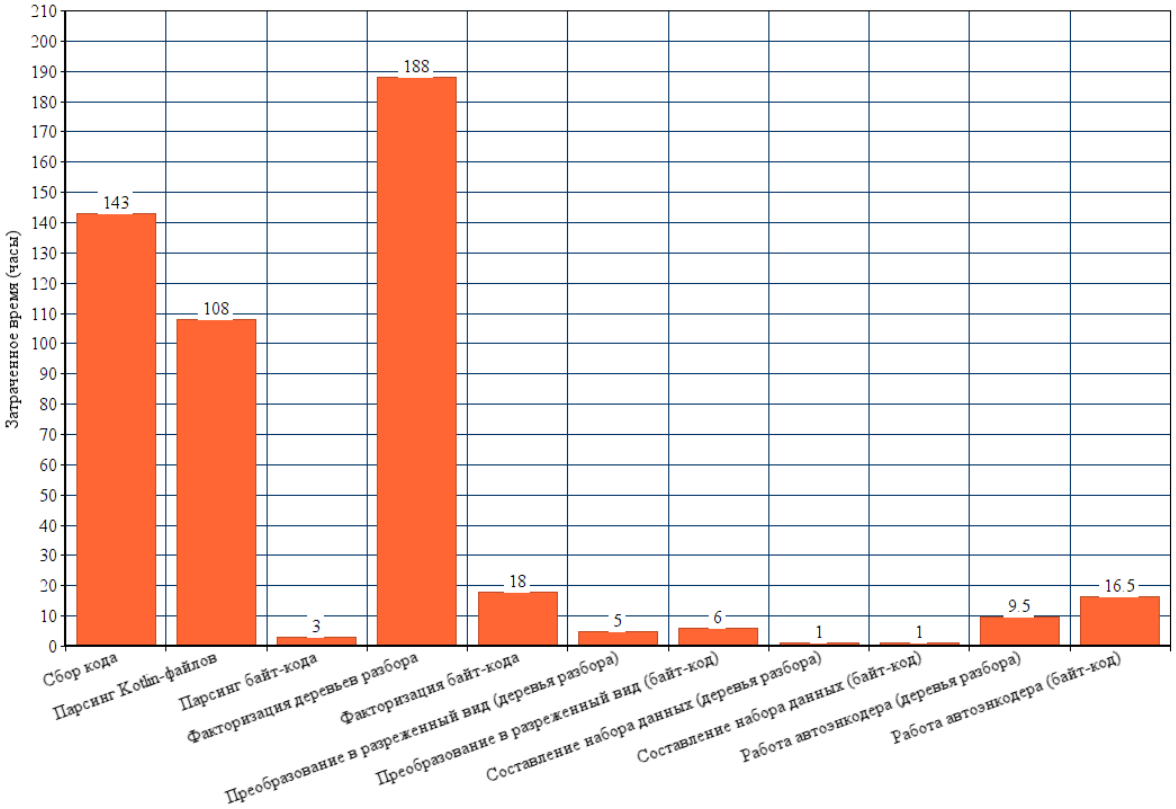

Рисунок 4.8.1 — временные затраты в процессе преобразования и анализа большого набора данных.

ГЛАВА 5. РАЗБОР И АНАЛИЗ РЕЗУЛЬТАТОВ

5.1 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[12] 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.

5.2 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 5.2.1.

Таблица 5.2.1 — количественная статистика аномалий по классам.

Название класса	**Количество аномалий по дереву разбора**	**Количество аномалий по байт-коду**	**Общее количество аномалий**
Big and complex enums	0	7	7
Big code hierarchy	30	6	36
Big companion object	0	2	2

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

Название класса	Количество аномалий по дереву разбора	Количество аномалий по байт-коду	Общее количество аномалий
Big constants set	0	22	22
Big init method in bytecode	0	2	2
Big methods	0	3	3
Big multiline strings	27	14	41
Big static arrays or map	30	3	33
Complex or long logical expressions	6	0	6
Long calls chain	5	2	7
Long enumerations	2	0	2
Many assignment statements	57	9	66
Many case in when	34	3	37
Many concatenations	13	0	13
Many consecutive arithmetic expressions	0	1	1
Many delegate properties	0	2	2
Many function arguments	17	8	25
Many generic parameters	6	0	6
Many if statements	38	5	43
Many inline functions	0	3	3

Название класса	Количество аномалий по дереву разбора	Количество аномалий по байт-коду	Общее количество аномалий
Many literal strings	0	4	4
Many loops	8	1	9
Many nested structures	0	2	2
Many not null assertion operators	0	1	1
Many root function definitions	0	4	4
Many safe calls	0	1	1
Many similar call expressions	86	42	128
Many square bracket annotations	0	3	3
Many type reified params	0	1	1
Nested calls	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: big code hierarchy, many similar call expressions, big multiline strings, big static arrays or map, many assignment statements, many case in when, many if statements и many similar call expressions. 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[13] и инструмент для публикации новых аномалий и новых классов[14].

5.3 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 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.

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

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 4.3.1 представлены экспертные оценки по классам, на рис. 4.3.2 — по аномалиям.

Таблица 5.3.1 — перечень экспертных оценок классов аномалий.

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

Оценка	Название класса
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
5	Many case in when
	Many delegate properties
	Many generic parameters

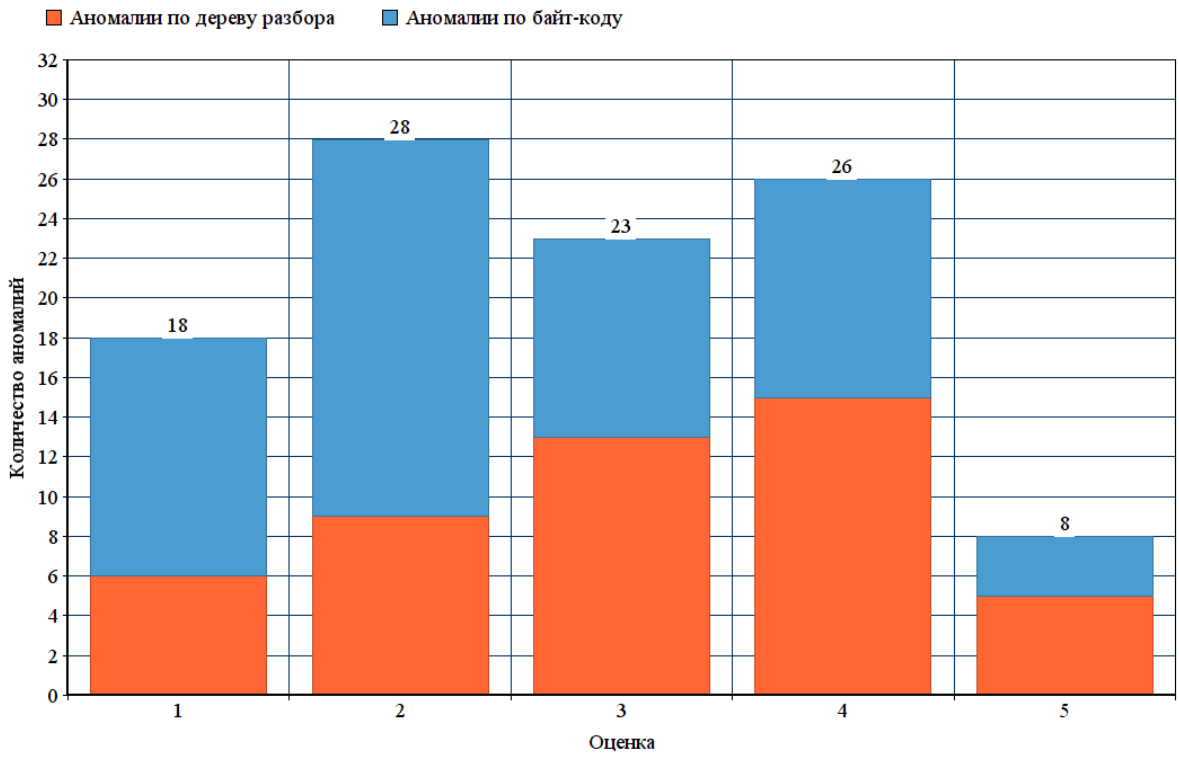

Рисунок 5.3.2 — экспертные оценки аномалий.

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

5.4 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.

5.5 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[15], 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. Агрегирующие значения приведены в таблице 5.5.1.

Таблица 5.5.1 — 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

Из данных аномалий были отобраны те, где вычисленные разности превосходили $1/4$ СКО. Итого было получено 38 условных аномалий: 6

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

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), Приложении Г (байт-код), приложении Д (исходный код) и приложении Е (байт-код) приведены два примера файлов-аномалий, «числа аномальности» по байт-коду которых сильно превышают «числа аномальности» по дереву разбора.

5.6 Использование результатов исследования по большому набору данных

Как уже было сказано, в зависимости от разности «чисел аномальности» условные аномалии можно трактовать по-разному:

1. если «число аномальности» по байт-коду сильно превосходит «число аномальности» по дереву разбора, то:
    1.1. либо было сгенерировано больше байт-кода, чем ожидалось, был сгенерирован сложный и нетипичный байт-код, что может свидетельствовать о наличии каких-либо проблем в кодогенераторе или оптимизациях компилятора Kotlin;
    1.2. либо было сгенерировано большое количество байт-кода вследствие удачной реализации некоторой конструкции языка: 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);

2. если «число аномальности» по дереву разбора превосходит «число аномальности» по байт-коду, то, скорее всего, можно говорить об удачно сработавших оптимизациях в компиляторе (которые либо отсекли часть кода, либо заменили его более коротким без потери семантики) — данные примеры также могут оказаться полезными: тут можно говорить о качестве отработавших оптимизаций, а также заниматься поиском проблемно отработавших оптимизаций.

Для каждого случая 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. Был разработан инструментарий, включающий инструменты для выполнения следующий задач:

1. сбор исходного кода и архивов с байт-кодом с ресурса GitHub;
2. фильтрация и парсинг исходного кода в деревья разбора PSI;
3. распаковка архивов с байт-кодом и декомпиляция (требуется для архивов приложений для ОС Android);
4. фильтрация байт-кода путем его сопоставления с файлами исходного кода с попутным формированием файлов с отображением файлов исходного кода на файлы с байт-кодом;
5. парсинг байт-кода в файлы в JSON формате с набором JVM-инструкций;
6. факторизация деревьев разбора PSI и байт-кода путем разложения на n-граммы;
7. отсечение предположительно малоинформативных n-грамм (с начала и конца частотного списка) из факторизованных представлений;
8. формирование набора данных в формате для алгоритмов машинного обучения из факторизованных представлений;
9. запуск автоэнкодера на подготовленных наборах данных;
10. осуществление отбора примеров в соответствии с полученными разностями между входными и выходными векторами автоэнкодера (формирование списка аномалий);
11. вычисление для аномалий по каждому источнику разностей «чисел аномальности» и формирование списка условных аномалий;

12. оценивание предоставленных аномалий и их классов (в виде веб-сайта[16]).

Исходный код всех разработанных инструментов был опубликованы на ресурсе GitHub[17]. В каждом репозитории содержится подробное описание работы инструмента, формат входных и выходных данных и примеры команды запуска.

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 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.

[16] http://victor.am/code-anomaly-detection/
[17] https://github.com/PetukhovVictor?tab=repositories

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).

ПРИЛОЖЕНИЕ А

Пример аномалии класса Long enumerations 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 Many generic parameters 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.

ПРИЛОЖЕНИЕ Д

Пример условной аномалии по байт-коду 2 (исходный код)

```
public object CabalTokelTypes {
    val COLON                   : IElementType  = HaskellTokenType(":")
    val COMMA                   : IElementType  = HaskellTokenType(",")
    val COMMENT                 : IElementType  = HaskellTokenType("COMMENT")
    val OPEN_PAREN              : IElementType  = HaskellTokenType("(")
    val STRING                  : IElementType  = HaskellTokenType("string")
    val NUMBER                  : IElementType  = HaskellTokenType("number")
    val TAB                     : IElementType  = HaskellTokenType("TAB")
    val NEGATION                : IElementType  = HaskellTokenType("!")
    val COMMENTS                : TokenSet = TokenSet.create(END_OF_LINE_COMMENT, COMMENT)
    val WHITESPACES             : TokenSet = TokenSet.create(TokenType.WHITE_SPACE)
    val PROPERTY_KEY            : IElementType = CabalCompositeElementType("PROPERTY_KEY", ::PropertyKey)

   (and 78 more similar assignment statements)
```

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

ПРИЛОЖЕНИЕ Е

Пример условной аномалии по байт-коду 2 (байт-код)

```
{
  "<clinit>" : [ "ldc", "invokestatic", "putstatic", "new", "invokespecial", "return" ],
  "getCOLON" : [ "getstatic", "dup", "ifnonnull", "new", "dup", "ldc_w", "ldc_w", "anewarray", "dup", ... (20 instructions)
  "getCOMMA" : [ "getstatic", "dup", "ifnonnull", "new", "dup", "ldc_w", "ldc_w", "anewarray", "dup", ... (20 instructions)
  "getCOMMENT" : [ "getstatic", "dup", "ifnonnull", "new", "dup", "ldc_w", "ldc_w", "anewarray", "dup", ... (20 instructions)
  "getOPEN_PAREN" : [ "getstatic", "dup", "ifnonnull", "new", "dup", "ldc_w", "ldc_w", "anewarray", "dup", ... (20 instructions)
  "getSTRING" : [ "getstatic", "dup", "ifnonnull", "new", "dup", "ldc_w", "ldc_w", "anewarray", "dup", ... (20 instructions)
  "getNUMBER" : [ "getstatic", "dup", "ifnonnull", "new", "dup", "ldc_w", "ldc_w", "anewarray", "dup", ... (20 instructions)
  "getTAB" : [ "getstatic", "dup", "ifnonnull", "new", "dup", "ldc_w", "ldc_w", "anewarray", "dup", ... (20 instructions)
  "getNEGATION" : [ "getstatic", "dup", "ifnonnull", "new", "dup", "ldc_w", "ldc_w", "anewarray", "dup", ... (20 instructions)
  "getCOMMENTS" : [ "getstatic", "dup", "ifnonnull", "new", "dup", "ldc_w", "ldc_w", "anewarray", "dup", ... (20 instructions)
  "getWHITESPACES" : [ "getstatic", "dup", "ifnonnull", "new", "dup", "ldc_w", "ldc_w", "anewarray", "dup", ... (20 instructions)
  "getPROPERTY_KEY" : [ "getstatic", "dup", "ifnonnull", "new", "dup", "ldc_w", "ldc_w", "anewarray", "dup", ... (20 instructions)

  (and 78 more nethods with similar bytecode)

  "<init>" : [ "aload_0", "invokespecial", "aload_0", "checkcast", "putstatic", "new", "dup", ... (846 instructions)
}
```

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