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.

ВВЕДЕНИЕ

В области разработки программного обеспечения довольно 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:

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

- исполнение кода функций библиотек поддержки времени исполнения, используемых в программе,

- 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, связанное с анализом влияния на производительность программ групп факторов в рамках стадий написания кода и его компиляции (в терминах теории компиляторов — в рамках стадии анализа, опуская стадию синтеза).

Хочется сразу отметить, что результаты такого анализа могут быть использованы как разработчиками целевых программ, так и разработчиками языка программирования, на котором эти программы составляются.

В качестве языка программирования, программы на котором и компилятор которого будут исследоваться, был выбран Kotlin, как один из наиболее молодых, быстро развивающихся и ещё недостаточно исследованных языков.

Таким образом, целью данной магистерской диссертации является разработка набора инструментов для анализа исходного кода программ на ЯП Kotlin и выявления потенциальных проблем производительности в них, а также проведение исследования по данной теме с использованием разработанного набора инструментов.

Практическая значимость работы заключается в получении по результатам исследования и разработки:

1. сгруппированного списка файлов с исходным кодом из достаточно объемного набора данных, являющихся с точки зрения тех или иных алгоритмов аномальным;
2. набора инструментов, позволяющего получать аналогичный список файлов на заданном проекте.

Первый результат должен стать важным и полезным в первую очередь разработчикам языка программирования Kotlin; второй же — пользователям языка, программистам, использующих для разработки своих проектов язык программирования Kotlin.

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 потенциальных проблем производительности в программах, написанных на нём.

Компилятор ЯП 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 интепретация или компиляция «на лету» в машинный байт-код JVM и исполнение кода 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 (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);
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 ЯП 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 лексемам (таким образом, список листьев данного дерева является набором лексем, которые, в свою очередь, являются результатом работы лексического анализатора в компиляторе Kotlin). 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 ЯП Kotlin в компиляторе, и в каких представлениях он может находиться, можно определить задачу для исследования и разработки.

Задачей будет являться анализ исходного кода на ЯП Kotlin в представлении в виде дерева разбора PSI, а также сгенерированного по нему JVM байт-кода. Такой анализ должен будет быть направлен на обнаружение потенциальных проблем производительности в программах с анализируемым кодом.

Потенциальные проблемы производительности могут быть выражены в слишком объемном сгенерированном JVM байт-коде, в большом количестве повторяющихся JVM инструкций или наборов инструкций, в слишком объёмном и/или глубоком дереве разбора PSI, в большом количестве повторяющихся узлов или наборов узлов, а также в нетипичном (по меркам анализируемого набора данных) JVM байт-коде или дереве разрбора PSI. Будет называть файлы с исходным кодом, которым соответствует такое дерево разбора PSI и/или JVM байт-код, аномальными.

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

На втором этапе предлагатеся разработать набор инструментов для запуска поиска аномальных файлов на заданном проекте (наборе исходых кодов). Предполагается, что такие файлы будут интересны программистам, занимающихся разработкой этого проекта, как файлы, имеющие потенциальные проблемы производительности.

Разработчикам же ЯП Kotlin предполагается, что будут интересны следующие результаты:

- полный набор найденных файлов-аномалий с исходным кодом на эталонном наборе данных: предлполагается, 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 конструкций языка;

- набор файлов с исходным кодом, сгенерированный JVM байт-код по которым считается аномальным, а дерево разбора PSI не превышает некоторые заданные ограничения (на глубину, либо на количество узлов), такой случай может соответствовать одному из двух вариантов:

    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 компилятора Kotlin;
    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);

- 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 некорректной генерации кода по каким-либо конструкциям).

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 обнозначенные рамки: целью исследования является анализ исходного кода в разных представлениях.

Также предполагается отсутствие заранее предоставленных примеров файлов-аномалий, т. к. их характер может быть самым различным, и предполагается, что может оказаться, что известна лишь малая часть аномалий.

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

Для анализа поведения программы 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:

- необходимость фактического запуска программы в тестовой среде, что, как правило, является более сложно организуемым, чем статический анализ кода;

- 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);

- 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.

2 ОБЗОР ИЗВЕСТНЫХ МЕТОДОВ И СРЕДСТВ РЕШЕНИЯ ЗАДАЧИ

2.1 Определение области решения задачи

Как уже было отмечено, объектом поиска будут являться файлы с исходным кодом на ЯП Kotlin, PSI дерево разбора и сгенерированный JVM байт-код которых, по тем или иным параметрам являются аномальными (нетипичными), поскольку предполагается, что аномальность будет заключается в слишком объемном JVM байт-коде и слишком объемном и/или глубоком дереве разбора PSI.

Задача обнаружения аномалий уже была поставлена в области машинного обучения. Для её решения существует множество техник. Рассмотрим эти техники, оценим их пригодность для поставленной задачи и определим условия для их применения.

2.2 Методы решения задачи обнаружения аномалий

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

- точечные аномалии: соответствуют случаям, когда отдельный объект данных является аномальным по отоношению к остальным данным (см. Рисунок 2.2.1: C1 и C2 — не аномальные группы объектов, A1 и A1 — аномалии);

- котекстуальные аномалии: соответствует случаям, когда отдельный объект данных является аномальным лишь в определенном контексте, определяется контекстуальными аттрибутами — например, определенным временем наблюдения аномальных объектов (см. Рисунок 2.2.2: в точке A наблюдается аномалия в отличие от точек N1-N5 с аналогичным значением функции);

- коллективные аномалии: соответствуют случаям, когда совместное появление некоторого числа объектов является аномальным объектов (см. Рисунок 2.2.2: участок A является коллективной аномалией).

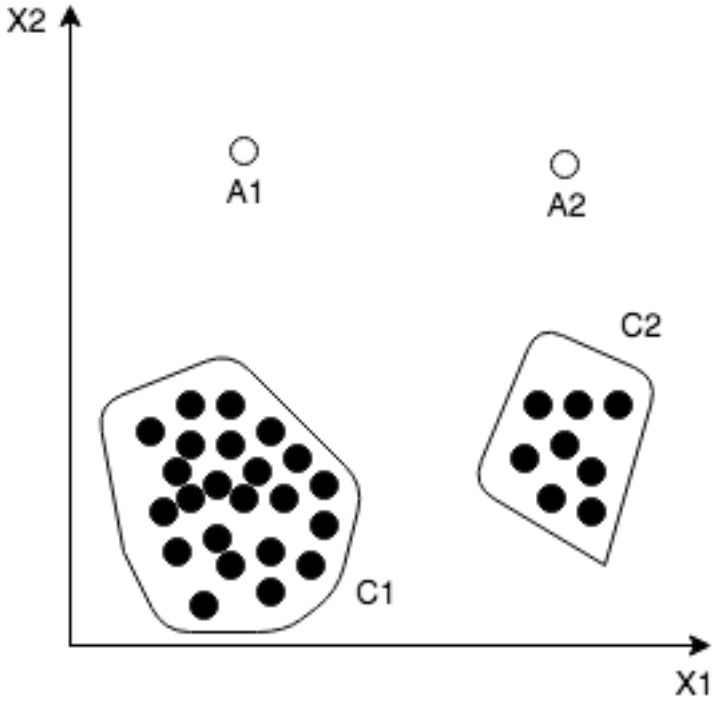

Рисунок 2.2.1 — демонстрация вида точечных аномалий.

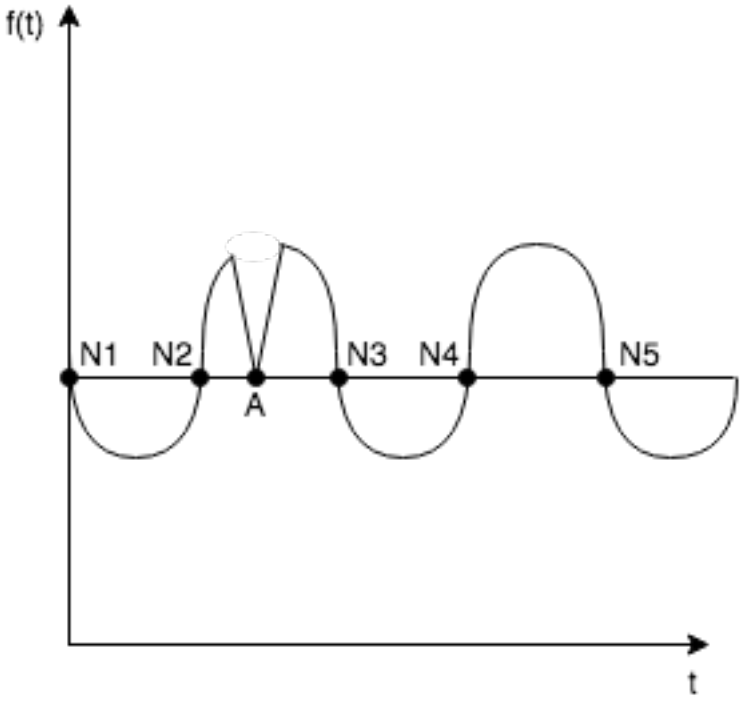

Рисунок 2.2.2 — демонстрация вида контекстуальных аномалий.

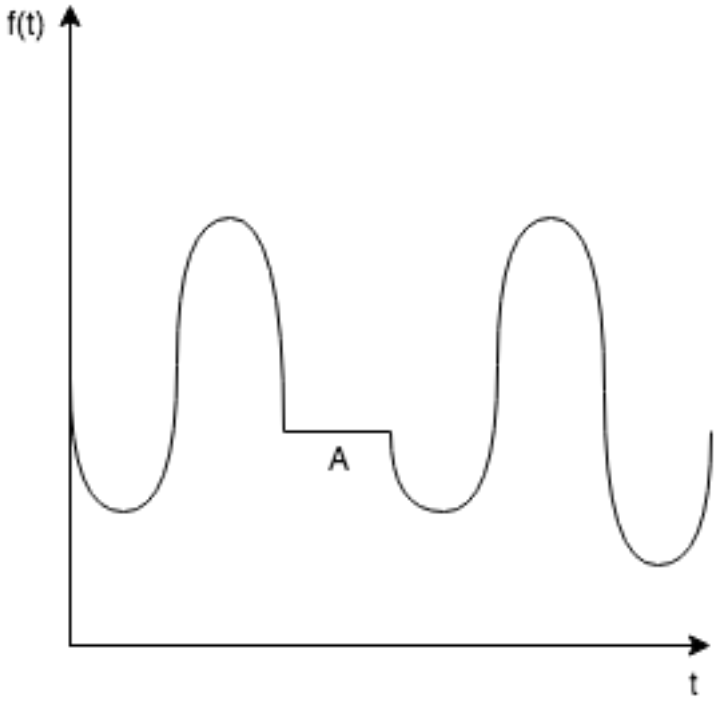

Рисунок 2.2.3 — демонстрация вида коллективных аномалий.

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 приязки 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.

2.2.1 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).

2.2.2 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). 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.

2.2.3 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.

2.2.4 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.

2.2.5 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 количество 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.

2.2.6 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. 2.2.6.1), 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).

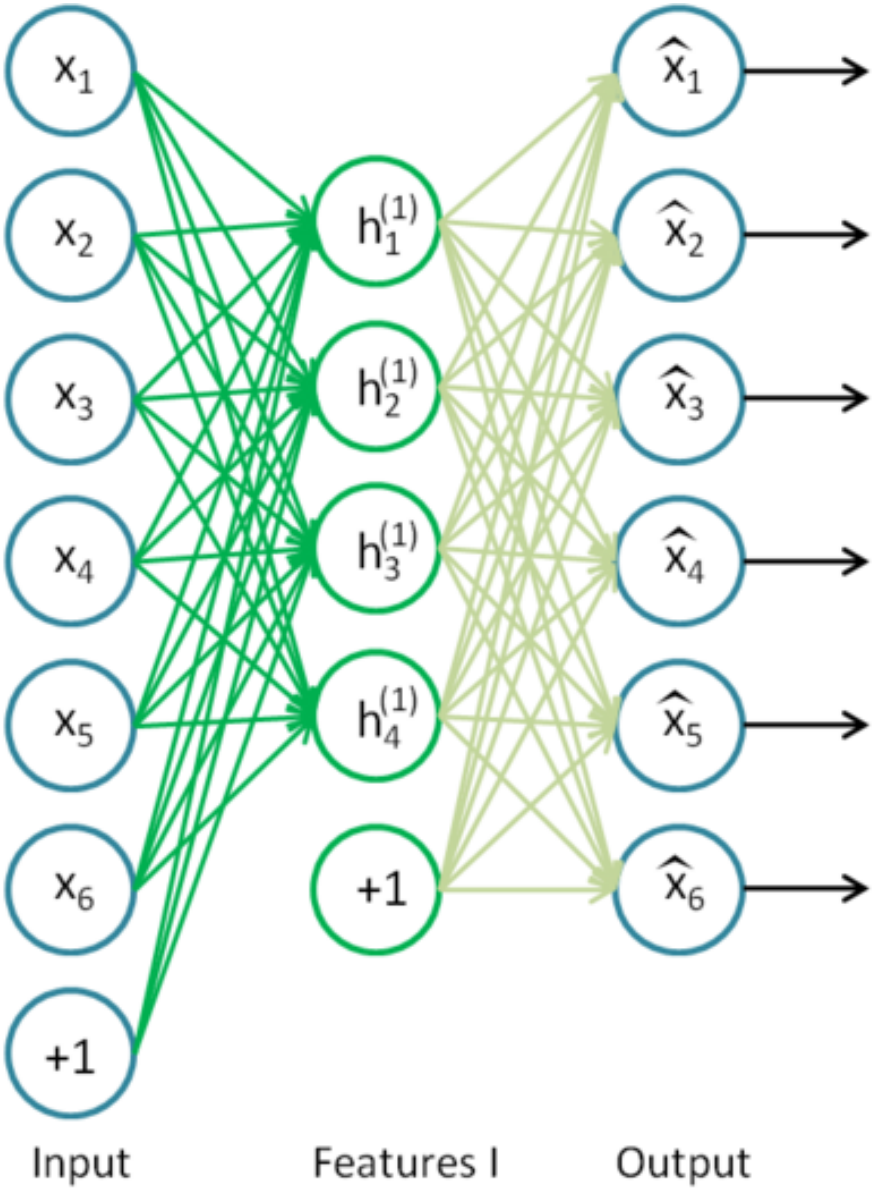

Рисунок 2.2.6.1 — схематичное изображение модели автоэнкодера.

2.3 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 — факторизации дерева разбора PSI и JVM байт-кода.

2.3.1 Выбор способа факторизации дерева разбора PSI и JVM байт-кода

Для факторизации дерева разбора PSI и JVM байт-кода было выбрано разложение их на n-граммы. 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.

Преимущество данного метода факторизации заключается в попутном сборе легко интерпретируемой человеком информации о программе. Например, имея факторизованные 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 функциональности в будущем.

3 ЭКСПЕРИМЕНТ

3.1 Сбор данных

Наибольшее количество открытого кода на ЯП Kotlin содержит ресурс Github. Будет использовать его для сбора кода, факторизации кода и составления набора данных, пригодного для анализа с помощью автоэнкодера. Сбор кода будем осуществлять по репозиториям (для более удобного и простого сопоставления исходного кода и JVM байт-коду в дальнейшем). 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).

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 JVM байт-кода.

3.1.1 Инструменты для сбора данных

Перед сбором непосредственно самого кода был разработан инструмент для формирования списка всех репозиториев, основным языком которых является Kotlin. Инструмент разбивал промежуток времени с 1-го января 2009 года до настоящего времени на такие отрезки, на которых бы находилось не более 1000 репозиториев (дата создания которых лежит в соответствующем промежутке). В конечном итоге с помощью данного инструмента был составлен файл со списком всех репозиториев с ресурса Gihub, основным языком которых является Kotlin.

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

3.1.2 Результаты этапа сбора данных

В конечном счете 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.

Итого было получено 930 тыс. файлов с исходным кодом на ЯП Kotlin. Релизные файлы подвергались фильтрации и сопоставлению файлам с исходным кодом на лету.

3.2 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.

3.2.1 Инструменты для сопоставления и фильтрации файлов

Для решения описанной задачи был разработан инструмент, который выполняет фильтрацию файлов репозитория (оставляя только файлы с исходным кодом Kotlin), а также сопоставление этих файлов с файлами с JVM 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 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.

3.2.2 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.

3.3 Разбор собранных файлов

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

3.3.1 Инструменты для разбора файлов

Для парсинга исходных кодов на ЯП Kotlin был склонирован репозиторий компилятора Kotlin, 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 ЯП Kotlin записывал деревья разбора в файлы с аналогичным именем, но с добавленным расширением json.

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

3.4 Факторизация деревьев разбора и байт-кода

Для осуществления факторизации деревьев разбора PSI и JVM байт-кода был написан инструмент, 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 количествам встречаемости.

3.4.1 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. количество 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. 3.4.1.1.

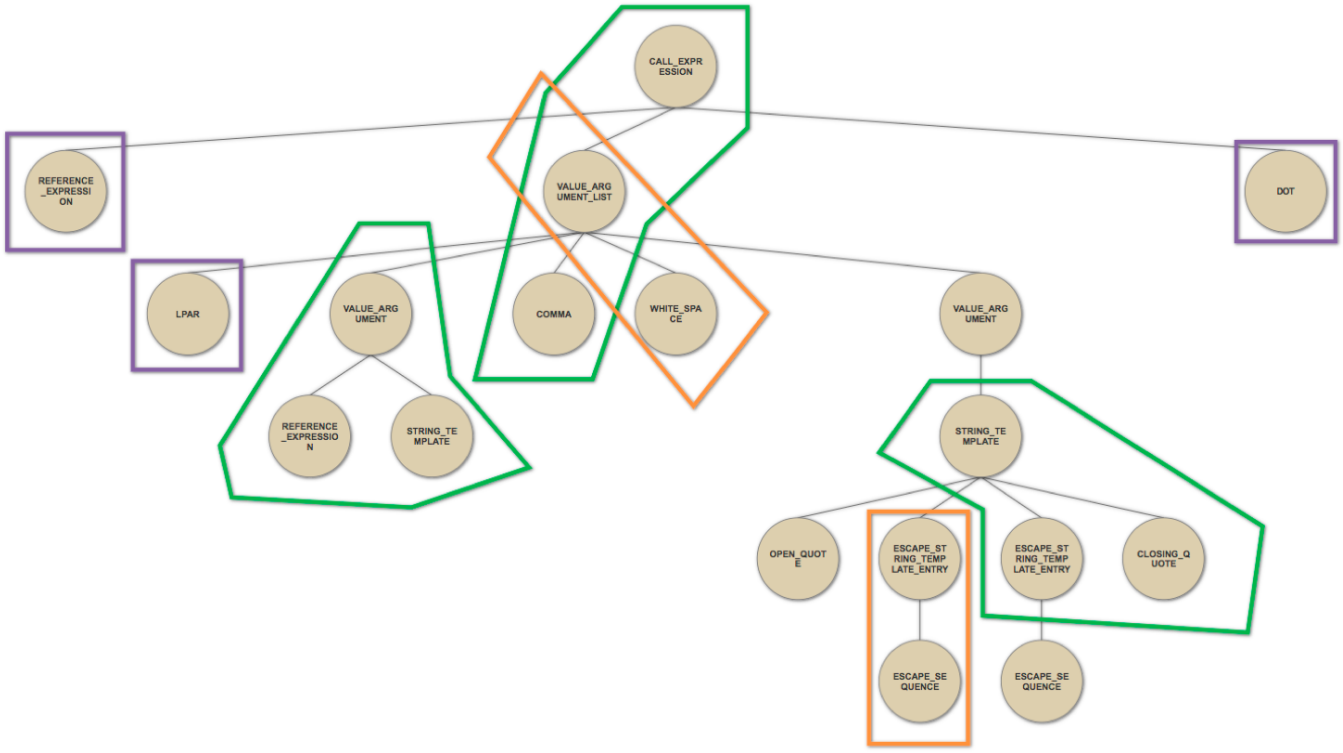

Рисунок 3.4.1.1 — демонстрация процесса факторизации дерева разбора.

В результате факторизации деревьев разбора было извлечено 11091 n-грамм. Из них uni-грамм — 250, bi-грамм — 1773, 3-грамм — 9068.

3.4.2 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 3 uni-граммы (данные числа соответствуют количеству сочетаний без повторений).

Процесс факторизации проиллюстрирован на рис. 3.4.2.1.

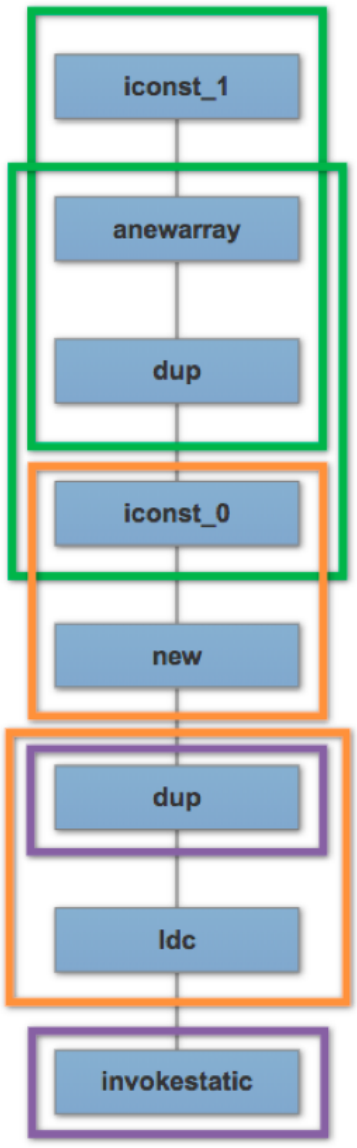

Рисунок 3.4.2.1 — демонстрация процесса факторизации байт-кода.

В результате факторизации байт-кода было извлечено 110835 n-грамм. Из них uni-грамм — 197, bi-грамм — 9382, 3-грамм — 101256.

3.5 Преобразование данных в разреженный вид

Перед непосредственным 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, принимающих на вход путь к директории с факторизованными деревьями разбора или байт-кодом, и записывающий в другую указанную директории по аналогичному пути разреженное представление вектора 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.

3.6 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 после загрузки в память (перед запуска 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 10% случайных примеров тестовой выборки).

3.7 Запуск автоэнкодера

На полученном наборе данных по деревьям разбора был запущен автоэнкодер в соответствии с составленной ранее моделью (см. раздел 2.2.6) со следующими параметрами:

- количество эпох (epochs) — 5;
- размер партии (batch size) — 1024.

Время обучения составило около 8 часов, время предсказания — 1.5 часа.

На полученном наборе данных по байт-коду автоэнкодер был запущен с таким же количество эпох, но с batch size, равным 128 (был уменьшен пропорционально размера набору данных). Время обучения составило около [?] часов, время предсказания — [?] часа.

3.8 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. 3.8.1 в виде гистограммы изображены временные затраты на этапы преобразования и анализа данных.

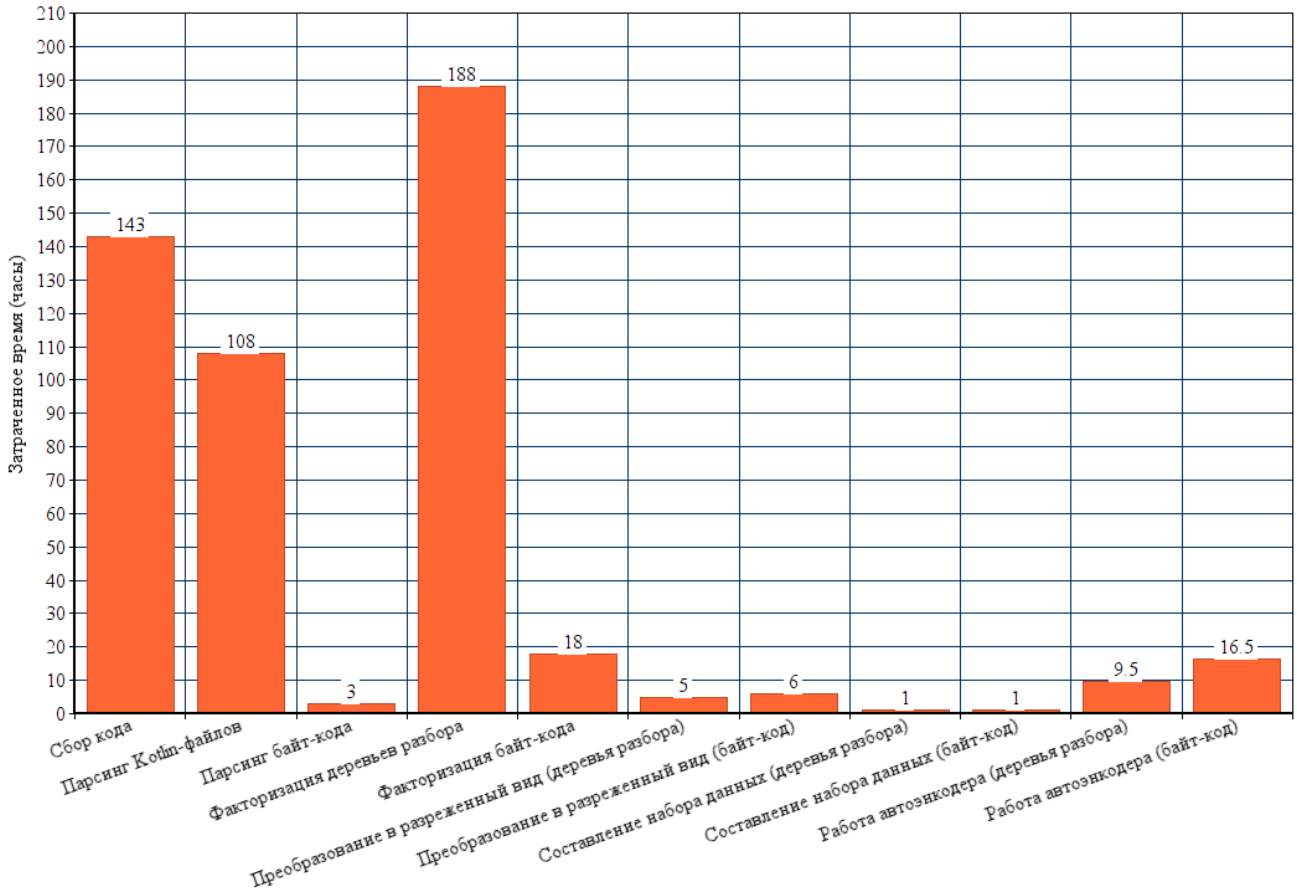

Рисунок 3.8.1 — временные затраты в процессе преобразования и анализа данных.

4 РАЗБОР И АНАЛИЗ РЕЗУЛЬТАТОВ

4.1 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 стандартных квадратичных отклонения. Данные примеры были условно названы «аномалиями». Для данного отбора был написал соответствующий инструмент с возможностью задания произвольного требуемого минимального отклонения.

Таким образом было получено ? аномалий по дереву разбора и ? аномалий по байт-коду. Набор расстояний был сопоставлен исходным файлам с помощью написанного скрипта.

4.2 Классификация аномалий

Полученный набор файлов-аномалий с исходных кодом и байт-кодом был вручную классифицирован. Для файлов-аномалий по байт-коду были получены соответствующие файлы с исходным кодом, которые служили вспомогательным фактором при классификации. Было сформировано 30 классов (513 аномалий).

Таблица 4.2.1 — 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
Big constants set	0	22	22
Big init method in bytecode	0	2	2
Big methods	0	3	3

Название класса	Количество аномалий по дереву разбора	Количество аномалий по байт-коду	Общее количество аномалий
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
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

Название класса	Количество аномалий по дереву разбора	Количество аномалий по байт-коду	Общее количество аномалий
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

Стоит отметить, что 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 ЯП Kotlin.

Предполагается, что примеры-аномалии по дереву разбора являются представителями случаев неправильного использования языка, и будет 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).

Отдельно следует выделить файлы-аномалии по байт-коду с объемным байт-кодом, но с относительно небольшим деревом разбора. В Приложении А (исходный код), Приложении Б (байт-код), приложении В (исходный код) и приложении Г (байт-код) приведены два примера файлов-аномалий из класса Big init method in bytecode.

Такие примеры могут свидетельствовать о каких-либо проблемах в кодогенераторе или оптимизациях компилятора Kotlin и нуждаются в более глубоком изучении.

Для просмотра файлов-аномалий по типам и классам был разработан сайт и инструментарий для публикации новых аномалий и новых классов.

4.3 Сбор экспертных оценок по аномалиям и их классам

Для сбора экспертных оценок предварительно на разработанном сайте были выделены лишь наиболее показательные аномалии-представители классов (из группы похожих аномалий был отобран лишь один экземпляр; были исключены аномалий, в которых особенность класса была выражена в меньшей степени).

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. …

ПРИЛОЖЕНИЕ А

Пример 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 А.1 – исходный код одной из найденных аномалий по байт-коду

ПРИЛОЖЕНИЕ Б

Пример 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 Б.1 – байт-код одной из найденных аномалий по байт-коду

ПРИЛОЖЕНИЕ В

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

ПРИЛОЖЕНИЕ Г

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