Санкт-Петербургский государственный университет

Программная инженерия

Смиренко Кирилл Петрович

# Детектор аномалий в программах на языке Kotlin

Бакалаврская работа

Научный руководитель:
доц., к.т.н. Брыксин Т. А.

Рецензент:
аналитик ООО "Интеллиджей Лабс" Поваров Н. И.

Санкт-Петербург
2018

SAINT PETERSBURG STATE UNIVERSITY

Software engineering

Kirill Smirenko

# Anomaly detection in Kotlin code

Graduation Thesis

Scientific supervisor:
Candidate of Engineering Sciences Timofey Bryksin

Reviewer:
analyst, IntelliJ Labs Co. Ltd. Nikita Povarov

Saint Petersburg
2018

# Оглавление

# Введение

В настоящее время индустрия программного обеспечения активно развивается. Так, согласно отчёту компании CodeDx, ежегодно создаётся более 111 миллиардов строк исходного кода [1]. В связи с этим естественным образом встаёт проблема контроля правильности работы программного кода, его безопасности и быстродействия. При этом важно отслеживать качество как исходного кода, так и используемого транслятора, потому что дефекты последнего могут негативно влиять на конечный программный продукт.

Одним из новых и динамично развивающихся языков программирования является язык Kotlin[1], разработанный компанией JetBrains. Это высокоуровневый язык программирования общего назначения, применяемый для разработки на платформах Java Virtual Machine (JVM), Android и некоторые другие. Разработчики языка Kotlin заинтересованы в улучшении экосистемы языка, в том числе компилятора. По их мнению, для этих целей могут быть полезны примеры необычного кода на Kotlin, выделяющегося своей нестандартной структурой — так называемые *кодовые аномалии*.

Под кодовой аномалией в данной работе понимается фрагмент исходного кода, по тем или иным причинам нетипичный для Kotlin. Это не означает, что такой код содержит ошибки или компилятор обрабатывает его некорректно; кодовая аномалия вполне может быть синтаксически и семантически корректна. Аномальность заключается лишь в том, что подавляющее большинство разработчиков не пишут код таким образом. Кодовые аномалии могут возникать по разным причинам: необычный подход отдельного программиста, предыдущий опыт разработки на другом, непохожем языке или в рамках другой парадигмы программирования.

Кодовые аномалии могут быть полезны разработчикам языка Kotlin по следующим соображениям. Во-первых, некоторые аномалии как примеры нетипичного, но формально корректного кода могут обратить

[1] https://kotlinlang.org

внимание разработчика языка на отдельные аспекты работы компилятора и вскрыть какие-либо неучтённые случаи. Во-вторых, некоторые кодовые аномалии могут служить тестами производительности компилятора и других языковых инструментов. В-третьих, анализ кодовых аномалий может привести к выводу о необходимости изменения или дополнения самого языка программирования. Всё это способствует развитию языка и его экосистемы.

Таким образом, имеется актуальная задача поиска кодовых аномалий на языке Kotlin. Её решением мог бы стать программный инструмент, позволяющий выявлять кодовые аномалии в базе открытого исходного кода на Kotlin и формировать наглядные отчёты. Созданию такого инструмента и посвящена настоящая работа.

## Постановка задачи

Целью данной работы является создание системы поиска кодовых аномалий на языке Kotlin. Для достижения этой цели были сформулированы следующие задачи:

1. провести обзор предметной области;
2. выполнить проектирование системы поиска кодовых аномалий, выбрать подходящие для этой задачи алгоритмы;
3. реализовать требуемую систему;
4. провести апробацию реализованной системы и предоставить отчёт разработчикам языка программирования Kotlin.

# 1. Обзор

Проблема поиска аномалий давно известна в области анализа данных [3]. Различают две разновидности этой задачи: детектирование выбросов и обнаружение новизны. В последние десятилетия ряд исследований показал эффективность методов машинного обучения при решении задач анализа данных в разных предметных областях, в частности, при поиске аномалий [8]. Задачу поиска кодовых аномалий можно свести к "традиционной" проблеме поиска аномалий в данных. Для этого следует выделить две подзадачи: преобразование фрагментов исходного программного кода к векторному виду и поиск выбросов в векторизованных данных.

## 1.1. Методы векторизации исходного кода

Существуют различные подходы к векторизации программного кода. Их условно можно разделить на две группы: извлечение явных признаков путём подсчёта метрик [5, 14, 18, 20, 21] и получение неявных признаков посредством хеширования, автокодирования или иного автоматического преобразования исходного кода и/или синтаксического дерева [2, 4, 19]. В данной работе было решено использовать явные синтаксические признаки, вычисляемые с помощью метрик исходного кода и конкретного синтаксического дерева. При таком подходе сохраняется явная связь векторного представления с оригинальным кодом, что потенциально позволяет не только находить, но и объяснять кодовые аномалии.

С другой стороны, одновременно с данной работой в Университете ИТМО выполнялась дипломная работа "Обнаружение проблем производительности в программах на языке программирования Kotlin с использованием статического анализа кода", в рамках которой также проводился поиск кодовых аномалий на Kotlin. В той работе проводилось извлечение неявных кодовых признаков двумя способами: автокодированием синтаксического дерева программы и автокодированием байт-кода, получаемого на позднем этапе работы компилятора Kotlin.

Было решено провести совместные эксперименты и объединить результаты поиска кодовых аномалий. Это будет описано в главе 5 настоящей работы.

Ниже приведён обзор наиболее релеватных работ, связанных с выбранным способом векторизации кода — подсчётом кодовых метрик.

В обзорной статье [20] рассматриваются работы, опубликованные в 2010–2015 годах, в которых используются кодовые метрики. Анализируются метрики, используемые для поиска ошибок в программном коде, оценки качества кода, его читаемости, а также в других областях. Для парадигмы объектно-ориентированного программирования (наиболее релевантной для данной работы) популярны такие метрики, как связность (cohesion) и сопряжение (coupling) методов класса, глубина дерева наследования, число строк кода, количество атрибутов элемента кода, цикломатическая сложность и другие.

Milepost GCC [14] — компилятор C/C++, содержащий 65 синтаксических метрик кода: количество блоков определённого вида (например, пустых или с одним потомком), количество определённых констант и другие. В оригинальной работе эти метрики используются для автоматического подбора параметров компилятора методами машинного обучения. Существует также работа [21], в которой средства Milepost GCC используются для получения признаков кода, на основе которых методами машинного обучения проводится детектирование плагиата в решениях задач по программированию.

Статья [5] посвящена автоматическому определению авторов программ на C/C++ при помощи методов машинного обучения. Из исходного кода извлекаются лексические и синтаксические признаки, в числе которых высота абстрактного синтаксического дерева (AST), частота использования ключевых слов языка C++, частота отдельных узлов AST и другие. Большинство из этих метрик можно вычислять на любом уровне организации кода, от функции до файла.

С учётом рассмотренных работ и специфики поставленной задачи, для векторизации кода решено использовать кодовые метрики, вычисляемые на уровне функции языка Kotlin. С одной стороны, имен-

но в функциях сосредоточено подавляющее большинство осмысленного кода, так как Kotlin предназначен прежде всего для объектно-ориентированного программирования. С другой стороны, функций существенно больше, чем классов или файлов исходного кода, а сами функции меньше по объёму. Конкретные кодовые метрики будут описаны в главе 3.

## 1.2. Поиск аномалий

Простейшим подходом поиска аномалий является выделение отклонений от статистических параметров распределения. Так, изолированная точка на границе области распределения с большой вероятностью является аномалией. Однако, далеко не все аномалии могут быть обнаружены таким образом, поэтому требуется рассмотреть более продвинутые подходы.

Существует достаточно много методов поиска выбросов в данных [3, 8]. К настоящей задаче лучше всего применимы методы, работающие с полностью непомеченными данными и не требующие предварительных знаний об их распределении. В данной работе было решено выбрать наиболее популярные алгоритмы поиска выбросов, для которых имеется проверенная реализация в рамках библиотеки Scikit-Learn [17].

### 1.2.1. Local Outlier Factor

Фактор локальных выбросов (Local Outlier Factor) — метод машинного обучения, основанный на понятии локальной плотности, которая определяется расстоянием до $k$ ближайших соседей. Точки, имеющие существенно более низкую плотность, чем их соседи, считаются выбросами [11].

### 1.2.2. Elliptic Envelope

Метод эллипсоидальной аппроксимации данных (Elliptic Envelope) моделирует облако точек как внутренность эллипсоида. Степень аномальности каждой точки определяется по расстоянию Махаланобиса до

остального множества точек. Расстояние Махаланобиса от вектора $x = (x_1, x_2, \ldots, x_N)^T$ до множества со средним значением $\mu = (\mu_1, \mu_2, \ldots, \mu_N)^T$ и ковариационной матрицей $S$ определяется по формуле:

$$D_M(x) = \sqrt{(x - \mu)^T * S^{(-1)} * (x - \mu)} \quad (1)$$

Данный метод лучше всего работает на одномодально распределённых (в частности, нормально распределённых) данных, однако заслуживает внимания в экспериментальном порядке [16].

### 1.2.3. Isolation Forest

Изолирующий лес (Isolation Forest) — разновидность случайного леса. Строится множество деревьев; для каждого ветвления выбирается случайный признак и случайное пороговое значение. Дерево строится до тех пор, пока каждый объект не окажется в отдельном листе. При таком подходе средняя длина пути от корня до листа является мерой аномальности соответствующего объекта, поскольку выбросы попадают в листья на более ранних этапах, чем типичные точки [12, 13].

### 1.2.4. One-class SVM

Метод опорных векторов для одного класса (One-class SVM) — разновидность метода опорных векторов, которая отделяет выборку от начала координат путём построения разделяющей гиперплоскости в пространстве более высокой размерности. Этот метод больше подходит для поиска новизны, чем поиска выбросов, так как разделяющая гиперплоскость строится на основании всей тренировочной выборки; однако возможен и поиск выбросов, поскольку после построения “положительного” класса часть данных оказывается за его пределами [6].

## 1.3. GitHub как источник данных

Для поиска кодовых аномалий необходимо располагать очень большим количеством исходного кода на Kotlin. Во-первых, кодовые анома-

лии встречаются редко, и чем больше исходного кода имеется в распоряжении, тем больше аномалий удастся обнаружить. Во-вторых, размер набора данных напрямую влияет на качество работы некоторых методов машинного обучения. В связи с этим разумным представляется использовать как источник данных GitHub — крупнейший веб-сервис для размещения ПО с открытым исходным кодом[2]. По оценкам на конец ноября 2017 года объём исходного кода на языке Kotlin, опубликованного на GitHub, достиг 25 миллионов строк кода и продолжает расти [10].

GitHub предоставляет удобный API для поиска и скачивания репозиториев с исходным кодом, имеющий, однако, ряд ограничений [3]. Так, GitHub не предоставляет данные о языковом составе репозиториев, а лишь автоматически определяет один основной язык. Кроме того, на каждый поисковый запрос предоставляется не более, чем 1000 детализованных результатов.

В рамках работы [7] был разработан проект GHTorrent, предоставляющий зеркало данных с GitHub посредством СУБД MySQL и MongoDB. Однако этот проект не гарантирует доступность данных за период ранее 2012 года, и, кроме того, для целей данной работы оказалось достаточно использовать оригинальный GitHub API.

## 1.4. PSI как средство анализа кода на Kotlin

IntelliJ Platform SDK предоставляет удобный инструментарий для синтаксического анализа программного кода на языках, используемых в средах разработки в рамках этой платформы — в том числе на Kotlin. Компилятор языка позволяет позволяет получить конкретное синтаксическое дерево кода — Program Structure Interface (PSI) [15]. Каждому типу узла конкретного синтаксического дерева соответствует определённый класс, реализующий интерфейс PsiElement. Поддерживается обход PSI с применением шаблона проектирования “посетитель”: для

[2]https://github.com

[3]https://developer.github.com/v3/

этого требуется унаследовать класс PsiElementVisitor, предоставляемый библиотекой компилятора Kotlin.

Среди инструментов, использующих PSI, наиболее релевантным является MetricsReloaded[4] — плагин к среде IntelliJ IDEA для подсчёта синтаксических метрик Java-кода. Доступно несколько десятков метрик, вычисляемых на уровне проекта, файла, пакета Java, класса или метода. В их числе такие метрики, как число строк кода с комментариями или без, цикломатическая и проектировочная сложность, метрики Холстеда, количество отдельных синтаксических конструкций. MetricsReloaded — проект с открытым исходным кодом, т.е. доступен как список метрик, так и реализация. Следовательно, можно сравнительно легко адаптировать Java-метрики для языка Kotlin, при трансляции которого используются другие элементы PSI.

[4]https://github.com/BasLeijdekkers/MetricsReloaded

# 2. Архитектура системы

Архитектура системы изображена на рисунке 1.

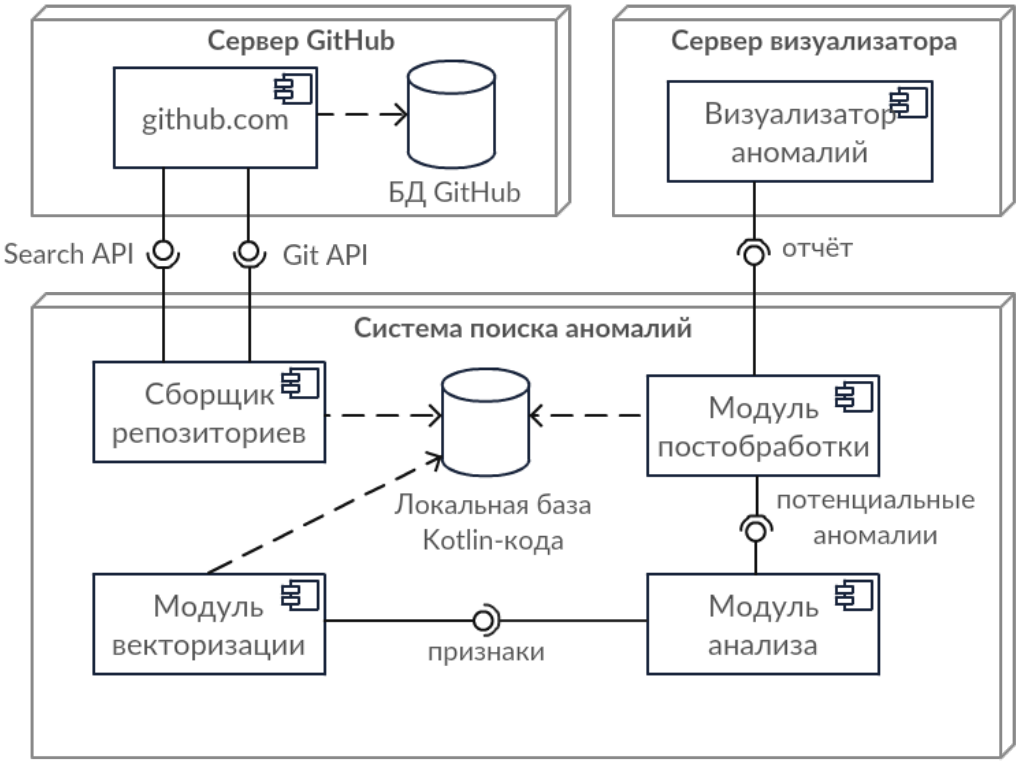


Рис. 1: Архитектура системы

Рассмотрим назачение основных компонент.

## 2.1. Сборщик репозиториев

Сборщик репозиториев осуществляет поиск и скачивание подходящих репозиториев с портала GitHub. Таким образом заполняется локальная база кода на Kotlin, которая служит источником данных для других компонент системы. Скачиваются репозитории, удовлетворяющие следующим условиям:

- репозиторий создан не позднее 18 февраля 2018 года;
- язык Kotlin указан для репозитория как основной;
- репозиторий не является ответвлённым от другого репозитория.

Инструмент реализован на языке Python с использованием библиотеки jq[5] для обработки принимаемых данных в формате JSON. Ниже представлен алгоритм работы.

1. Заданный глобальный промежуток времени разбивается на промежутки, в течение каждого из которых было создано не более 1000 репозиториев. Это делается путём последовательного подбора опорных временных точек, запросов к GitHub Search API и подсчёта количества результатов поиска.

2. Для каждого получившегося промежутка запрашиваются все репозитории, созданные в этих временных рамках. Результаты сохраняются в файлах формата JSON.

3. Последовательно скачиваются все репозитории, упомянутые в результатах поиска. При этом все файлы, кроме файлов исходного кода на Kotlin, удаляются с локального диска.

С помощью данного инструмента был собран набор из 47 751 репозиториев, содержащий 926 516 файлов исходного кода на Kotlin.

## 2.2. Модуль векторизации

Модуль векторизации исходного кода обрабатывает локальную кодовую базу и предоставляет на выходе набор численных данных в формате значений, разделённых запятыми (CSV). Единицей кода, из которой извлекаются признаки, является функция; это позволяет проводить достаточно тонкий поиск аномалий и упрощает процесс экспертной оценки результатов, так как оценивать нужно будет небольшие фрагменты кода.

Детали реализации данного модуля приведены в главе 3.

[5]https://stedolan.github.io/jq/

## 2.3. Модуль анализа

Модуль анализа осуществляет поиск аномалий в векторизованных данных. Для этого обучаются и используются классификаторы, основанных на разных методах машинного обучения: Elliptic Envelope, Local Outlier Factor, Isolation Forest. Результатом работы модуля являются списки потенциальных аномалий — элементов данных, которые каждый классификатор по отдельности помечает как аномальные с наибольшей уверенностью. Более подробно модуль описан в главе 4.

## 2.4. Модуль постобработки

Модуль постобработки предназначен для предварительной оценки потенциальных аномалий, полученных модулем анализа, и подготовки *отчёта* — набора файлов с кодовыми аномалиями, отправляемых разработчикам Kotlin. Модуль представляет собой консольное приложение с двумя режимами работы. В первом режиме оно восстанавливает исходный код функции — потенциальной аномалии по её сигнатуре, извлекая нужный фрагмент кода из файла, путь к которому задаётся в первой части сигнатуры. Текст каждой функции сохраняется в отдельном файле. Во втором, интерактивном режиме приложение позволяет проводить ручную разметку потенциальных аномалий как “полезные” (подтверждённые аномалии) либо “бесполезные” (не аномалии). Таким образом отсеиваются ложные аномалии и повышается качество отчёта, отправляемого разработчикам Kotlin. Экспертное оценивание отчёта с аномалиями проводится удалённо с помощью стороннего визуализатора кодовых аномалий, созданного в Университете ИТМО.

# 3. Векторизация исходного кода на Kotlin

Архитектура модуля представлена на рисунке 2. В рамках модуля доступно несколько *калькуляторов*: один из них (PrettyPrinter) просто выводит конкретное синтаксическое дерево в структурированном виде для отладки; другой (MethodFeatureCalculator) извлекает признаки из функций путём вычисления определённых признаков. Параметры запуска модуля определяют, какие калькуляторы будут использованы.

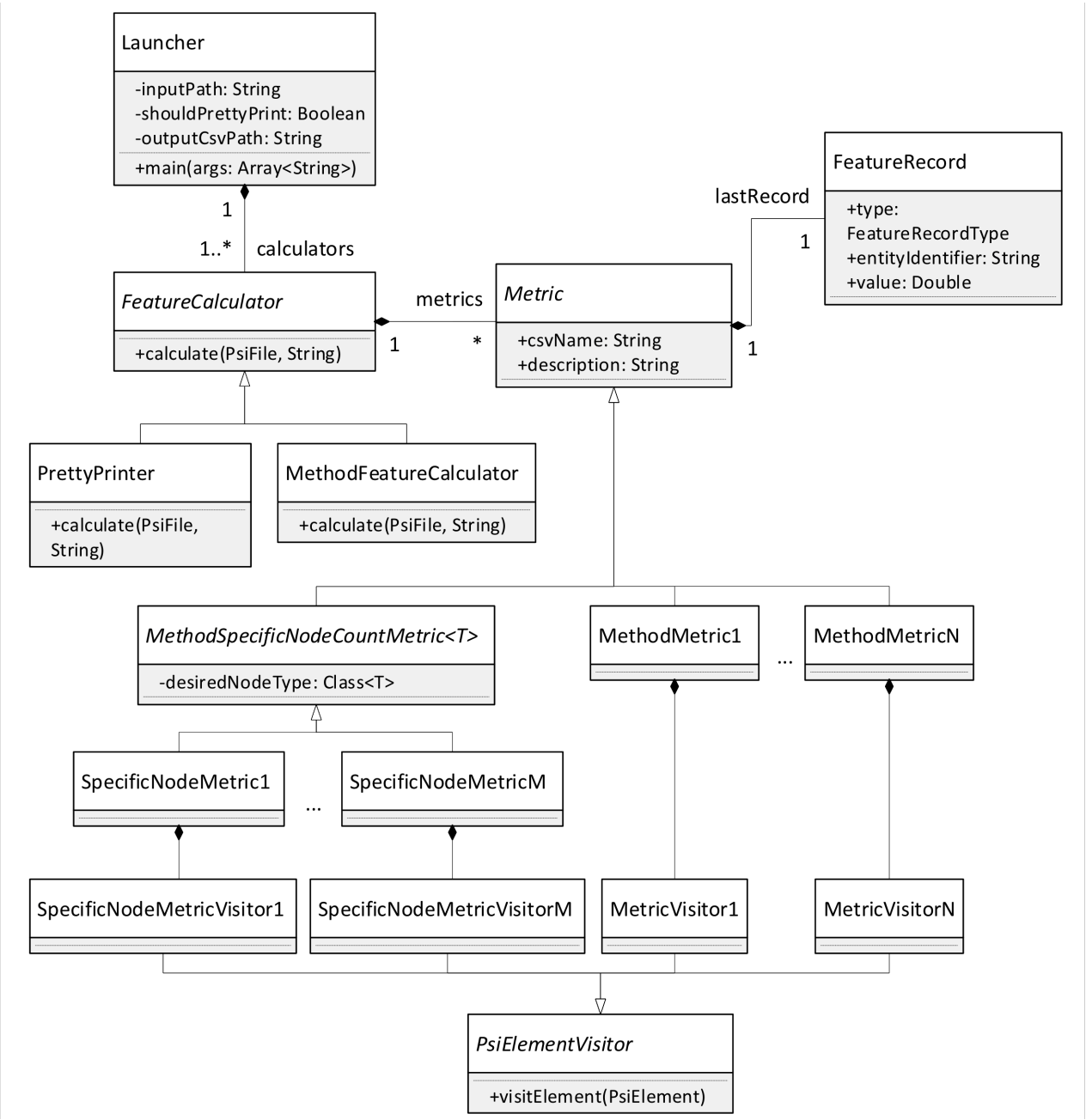


Рис. 2: Архитектура модуля векторизации кода

Признаки извлекаются с помощью *метрик* — классов, каждый из которых отвечает за подсчёт какого-то конкретного признака, например, цикломатической сложности или глубины вложенности циклов. Реализована и используется 51 метрика; метрики подсчитываются калькулятором последовательно и формируют вектор признаков. Список метрик приведён в таблице 1. Их можно условно разделить на следующие группы:

- общие характеристики фрагмента кода (число строк кода, количество узлов и высота CST);
- внешние характеристики Kotin-функции (количество формальных аргументов, типовых параметров, аннотаций; наличие модификатора suspend и др.);
- структурные характеристики (цикломатическая сложность, глубина вложенности циклов, среднее и максимальное число веток в when-выражении);
- число определённых элементов языка (выражений, операторов, ключевых слов, вызовов функций, строковых шаблонов и др.).

Метрики реализованы с применением шаблона проектирования "посетитель": каждой метрике соответствует свой подкласс абстрактного класса PsiElementVisitor. Такие подклассы содержат логику подсчёта метрики на основании данных в узлах PSI разного типа, а логика обхода самого синтаксического дерева возложена на существующий библиотечный код.

Метрика *MethodSpecificNodeCountMetric* является обобщённой и подсчитывает число узлов PSI определённого типа, который задаётся типовым параметром метрики. В целом выбранная архитектура обеспечила удобное добавление метрик, а разделение метрик по отдельным классам облегчило отладку системы.

Помимо извлечения признаков каждой обрабатываемой функции присваивается последовательный номер-идентификатор и *сигнатура* —

| **Группа** | **Обозначение** | **Краткое описание** |
|---|---|---|
| Общие | cstHeight | Высота CST |
| | nodeCount | Число узлов CST |
| | relativeLoc | Доля строк кода в объемлющем классе |
| | sloc | Число строк кода |
| Внешние | isSuspend | Наличие модификатора suspend |
| | isVoid | Возвращает ли void |
| | numAnnotations | Число аннотаций |
| | numTypeParameters | Число типовых параметров функции |
| | numValueParameters | Число формальных аргументов функции |
| Структурные | avgNumBlockChildren | Ср. число дочерних узлов блока |
| | avgNumWhenEntries | Ср. число веток в when-выражении |
| | cyclomaticComplexity | Цикломатическая сложность |
| | designComplexity | Проектировочная сложность |
| | maxLoopNestingDepth | Макс. глубина вложенности циклов |
| | maxNumBlockChildren | Макс. число дочерних узлов блока |
| | maxNumWhenEntries | Макс. число веток в when-выражении |
| | numBlocks | Число блоков |
| | numEmptyBlocks | Число пустых блоков |
| По типу элементов | numAssigns | Число операторов присваивания |
| | numBlockStringTemplates | Число блочных строковых шаблонов |
| | numCatch | Число catch-выражений |
| | numClassLiterals | Число выражений-ссылок на классы |
| | numCollectionLiterals | Число коллекций-литералов |
| | numConstExpr | Число константных выражений |
| | numDeclarations | Число KtDeclaration |
| | numDistinctKeywords | Число различных ключевых слов языка |
| | numEmptyStringLiterals | Число пустых строковых литералов |
| | numExpressions | Число выражений |
| | numFinally | Число finally-секций |
| | numForceUnwraps | Число использований оператора !! |
| | numIfExprs | Число if-выражений |
| | numKeywords | Число использований ключевых слов |
| | numLambdas | Число лямбда-выражений |
| | numLoopStatements | Число циклов |
| | numMethodCalls | Число вызовов методов |
| | numNestedClasses | Число вложенных классов |
| | numNestedFuns | Число вложенных функций |
| | numOneChildBlocks | Число блоков с одним дочерним узлом |
| | numOneConstants | Число констант-единиц |
| | numOperationReferences | Число операций |
| | numPlusOperations | Число операций сложения и конкатенации |
| | numReferences | Число ссылок |
| | numReturns | Число операторов возврата |
| | numSafeExpressions | Число null-безопасных вызовов |
| | numStatementExpressions | Число операторных выражений |
| | numStringLiteralTemplates | Число строковых шаблонов-литералов |
| | numStringTemplates | Число строковых шаблонов |
| | numThrows | Число throw-операторов |
| | numTry | Число try-выражений |
| | numTypecastExpr | Число выражений преобразования типов |
| | numZeroConstants | Число констант-нулей |

Таблица 1: Метрики Kotlin-функций

строка, содержащая путь к файлу, содержащему функцию, имя объемлющего класса, название функции и её параметры. Идентификатор и сигнатура необходимы для связывания текста потенциальной аномалии с вектором её признаков.

Таким образом, результат обработки одной функции языка Kotlin — вектор, содержащий идентификатор, сигнатуру и 51 извлечённый признак. Так формируется набор данных для последующей обработки модулем анализа. Из собранной кодовой базы с 47 751 репозиторием были извлечены признаки 4 044 790 функций, которые и составили основной набор данных.

# 4. Поиск выбросов

На рисунке 3 представлен алгоритм работы модуля анализа.

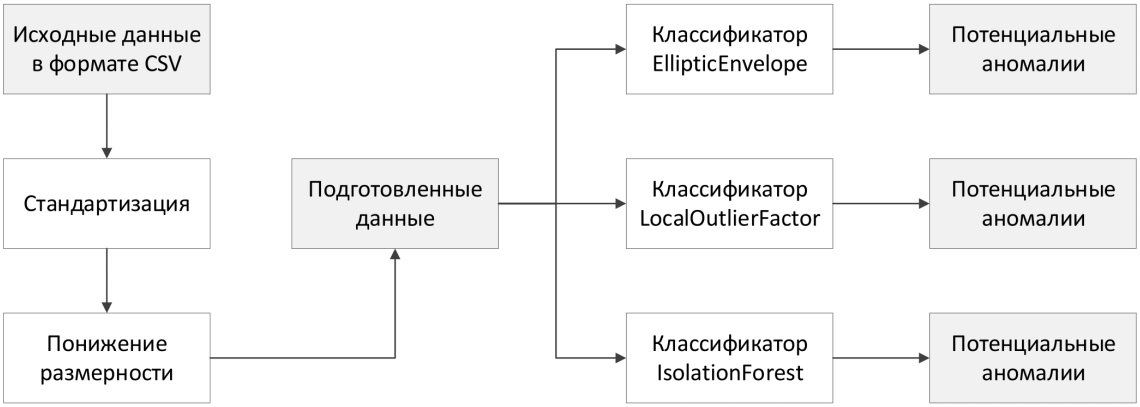


Рис. 3: Алгоритм работы модуля анализа

Входные данные в формате CSV содержат 2 бинарных и 49 количественных признаков. После загрузки данных проводится стандартизация количественных признаков (приведение к среднему значению 0 и дисперсии 1), чтобы уравнять влияние каждого на конечный результат. Затем производится понижение размерности с 51 до 20 методом главных компонент [9]. Число главных компонент было выбрано экспериментальным путём как компромисс между временем обучения моделей и сохранением объясняемой дисперсии (0,8).

Далее проводится обучение классификаторов с разными решающими функциями: EllipticEnvelope, LocalOutlierFactor, IsolationForest. В ходе предварительных экспериментов было решено отказаться от использования метода One-class SVM в связи с тем, что на основном наборе данных соответствующий классификатор работал слишком долго и так и не завершил вычисления.

Параметр зашумлённости указывается для каждого из выбранных трёх алгоритмов и определяет, какую примерно долю набора данных классификатор должен пометить как потенциальные аномалии. В связи с тем, что разрабатываемая система нацелена на поиск существенного, но обозримого человеком набора аномалий, параметр зашумлённости был подобран так, чтобы классификатор помечал 0,01% данных, или примерно 400 из 4 миллионов. Конкретные значения пара-

метра составили 0,0001 для EllipticEnvelope и IsolationForest и 0,001 для LocalOutlierFactor.

Для подбора других наиболее важных параметов алгоритмов было проведено несколько раундов экспериментов, при этом каждый раз производилась визуальная оценка подвыборки потенциальных аномалий размером около 100. Так был выбран параметр n_neighbors=20 для LocalOutlierFactor (число соседей, относительно которых для точки считается фактор локальных выбросов) и параметр n_estimators=200 для IsolationForest (число деревьев, которые строит алгоритм).

Дополнительно в рамках модуля была реализована возможность выделить потенциальные аномалии со значениями отдельных элементов вектора ниже процентили 0,1 или выше процентили 99,9 относительно всего набора данных. Это было использовано на промежуточных этапах для получения небольшого отчёта с аномалиями и упрощённой классификации потенциальных аномалий, однако в конечном итоге от этого шага было решено отказаться для большего охвата данных.

# 5. Апробация

## 5.1. Метод

С учётом того, что важнейшей задачей данной работы было формирование одного определённого набора кодовых аномалий для разработчиков Kotlin, был установлен следующий процесс апробации системы.

1. В сформированной локальной базе кода на Kotlin производится поиск потенциальных аномалий.

2. Автор работы проводит предварительную оценку потенциальных аномалий, отсеивая наименее интересные примеры.

3. Формируется отчёт и отправляется эксперту из команды разработчиков Kotlin.

4. Эксперт оценивает отчёт и предоставляет обратную связь.

## 5.2. Локальные эксперименты

Для финального раунда экспериментов был произведён запуск трёх классификаторов в конфигурациях, описанных в главе 4. Все потенциальные аномалии, полученные на этом этапе, были оценены автором работы, из них отобрано суммарно 322 уникальных аномалии и включено в отчёт. Результаты последних экспериментов приведены в таблице 2.

| **Метод** | **Потенциальных аномалий** | **Отобрано** | **Доля** |
|---|---|---|---|
| EllipticEnvelope | 405 | 17 | 4,2 % |
| IsolationForest | 405 | 179 | 44,2 % |
| LocalOutlierFactor | 405 | 128 | 31,6 % |
| **Итого (без дубликатов)** | 1213 | 322 | 26,5 % |

Таблица 2: Результаты последних локальных экспериментов

## 5.3. Экспертная оценка

322 отобранные потенциальные аномалии были разделены на 23 класса. Из них и отчёта, сформированного в рамках работы по поиску кодовых аномалий, выполненной в Университете ИТМО, был сформирован совместный отчёт, который был отправлен разработчикам языка Kotlin (при этом некоторые классы аномалий присутствовали в обоих отчётах). Отдельно была предоставлена выборка из совместного отчёта размером 146 аномалий, содержащая по несколько примеров каждого класса. В эту выборку вошло 43 кодовые аномалии из настоящей работы; это связано с тем, что в рамках другой работы тестировалось два разных подхода (упомянутые в главе 1.1) и было взято в два раза больше примеров.

Эксперт из команды Kotlin провёл ручную оценку указанной выборки. Полезность каждого класса и каждой аномалии по отдельности оценивалась по пятибалльной шкале.

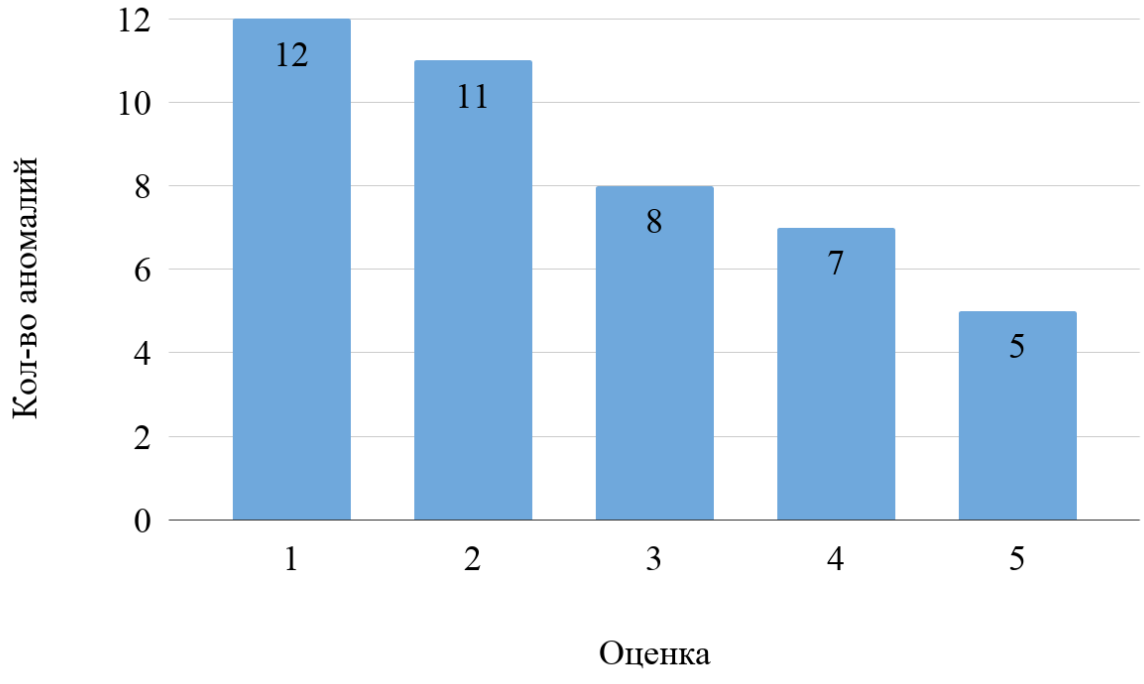


Рис. 4: Экспертная оценка отдельных аномалий

Результаты оценки 43 кодовых аномалий приведены на рис. 4. Результаты оценки классов приведены в таблице 3. В столбце $N_{\text{совм}}$ ука-

зано количество аномалий соответствующего класса в выборке из совместного отчёта по двум работам, т. е. число аномалий, на основании которых оценивался класс. В столбце $Q$ указана оценка. Семь классов аномалий, получившие оценки 4 и 5, выделены полужирным шрифтом.

| № | Краткое описание | $N_{\text{совм}}$ | $Q$ |
|---|---|---|---|
| **1** | **Много типовых параметров** | 3 | 5 |
| **2** | **Много веток в when-выражении** | 8 | 5 |
| **3** | **Много делегированных свойств** | 3 | 5 |
| **4** | **Необычные кодовые конструкции** | 3 | 4 |
| **5** | **Сложные аннотации функции** | 2 | 4 |
| **6** | **Много if-выражений** | 9 | 4 |
| **7** | **Много схожих вызовов** | 10 | 4 |
| 8 | Сложная структура кода | 8 | 3 |
| 9 | Большое тело функции | 5 | 3 |
| 10 | Много конструкций try-catch | 1 | 3 |
| 11 | Много циклов | 4 | 2 |
| 12 | Большие литеральные коллекции | 10 | 2 |
| 13 | Много throw-операторов | 2 | 2 |
| 14 | Много ссылок на класс (н-р, “A::class.java”) | 2 | 2 |
| 15 | Много схожих фрагментов кода | 2 | 2 |
| 16 | Много лямбда-выражений | 2 | 2 |
| 17 | Много пустых строковых литералов | 2 | 1 |
| 18 | Много преобразований типов | 2 | 1 |
| 19 | Много assert-операторов | 1 | 1 |
| 20 | Код из руководств по Kotlin | 2 | 1 |
| 21 | Много строковых литералов | 4 | 1 |
| 22 | Много локальных переменных | 2 | 1 |
| 23 | Много вложенных функций | 2 | 1 |

Таблица 3: Экспертная оценка классов аномалий

После получения обратной связи из 322 аномалий индивидуального отчёта были отдельно выбраны аномалии, принадлежащие семи классам с высокими оценками; таких оказалось 111 штук. Таким образом был сформирован и передан разработчикам отчёт с наиболее интересными кодовыми аномалиями.

## 5.4. Сравнение с аналогичной работой

В таблице 4 приведены результаты экспертной оценки классов кодовых аномалий из всего совместного отчёта по двум работам. Знаком

‘+’ отмечено, присутствовал ли класс в отчёте по настоящей работе ($W_1$) и в отчёте по аналогичной работе, упоминавшейся ранее ($W_2$). В столбце $Q$ приведена оценка; классы, получившие высокие оценки (4 и 5), выделены полужирным шрифтом.

| Краткое описание | $W_1$ | $W_2$ | $Q$ | Краткое описание | $W_1$ | $W_2$ | $Q$ |
|---|---|---|---|---|---|---|---|
| Много веток в when-выражении | + | + | 5 | Много схожих фрагментов | + | | 2 |
| Много делегированных свойств | + | + | 5 | Много inline-функций | | + | 2 |
| Много типовых параметров | + | + | 5 | Много опер. конкатенации | | + | 2 |
| Большая иерархия вызовов | | + | 4 | Много ссылок на класс | + | | 2 |
| Много if-выражений | + | + | 4 | Много лямбда-выражений | + | | 2 |
| Большой набор констант | | + | 4 | Много throw-операторов | + | | 2 |
| Сложные аннотации функции | + | | 4 | Много reified тип. парам. | | + | 2 |
| Длинные цепочки вызовов | | + | 4 | Сложные лог. выражения | | + | 2 |
| Длинные перечисления (enum) | | + | 4 | Много циклов | + | + | 2 |
| Необычные кодовые конструкции | + | | 4 | Большие литералы-коллекции | + | + | 2 |
| Сложная иерархия enum-классов | | + | 4 | Много локальных переменных | + | | 1 |
| Много аннотаций с кв. скобками | | + | 4 | Много вложенных функций | + | | 1 |
| Много схожих вызовов | + | + | 4 | Большой companion-объект | | + | 1 |
| Много функций на уровне файла | | + | 4 | Длинный init-метод | | + | 1 |
| Сложная структура кода | + | + | 3 | Много null-безопасных вызовов | | + | 1 |
| Большое тело функции | + | + | 3 | Много преобразований типов | + | | 1 |
| Длинные строковые литералы | | + | 3 | Много пустых стр. литералов | + | | 1 |
| Много конструкций try-catch | + | | 3 | Много строковых литералов | + | + | 1 |
| Много присваиваний | | + | 3 | Много assert-операторов | + | | 1 |
| Много послед. арифм. выражений | | + | 3 | Много вложенных структур | | + | 1 |
| Много параметров у функции | | + | 3 | Код из руководств по Kotlin | + | | 1 |
| Много операторов !! | | + | 2 | | | | |

Таблица 4: Экспертная оценка классов аномалий из двух работ

Анализ результатов показал, что оба подхода — подсчёт метрик и автокодирование — применимы для поиска кодовых аномалий, и нельзя однозначно выделить лучший подход. С одной стороны, использование кодовых метрик упрощает анализ найденных аномалий, а набор этих метрик можно легко редактировать, подводя под более интересные типы аномалий. С другой стороны, автокодирование позволяет извлекать неявные кодовые признаки, для которых нет соответствующих метрик либо такие метрики достаточно сложны в реализации.

## 5.5. Вывод

Эксперименты показали, что несмотря на сравнительно невысокую долю полезных данных среди автоматически найденных потенциаль-

ных аномалий, подход, реализованный в данной работе, вполне применим для формирования ограниченных наборов кодовых аномалий, представляющих интерес для разработчиков языка.

# Заключение

В ходе данной работы были получены следующие результаты.

- Создана архитектура системы для поиска и анализа кодовых аномалий на языке Kotlin.
- Реализована система поиска кодовых аномалий, в том числе следующие компоненты:
  - модуль векторизации исходного кода;
  - модуль поиска выбросов в векторизованных данных.

  Исходный код опубликован на портале GitHub[6]; автор работал под учётной записью ksmirenko.
- Проведена апробация системы на 47 751 проектах на Kotlin с GitHub; отчёт с наиболее интересными аномалиями предоставлен разработчикам Kotlin.

Из полученных результатов можно сделать вывод о том, что выбранный подход можно с успехом применять для поиска кодовых аномалий на языке Kotlin. В текущем виде подход требует существенного человеческого вмешательства на этапе анализа и отбора потенциальных аномалий. Следовательно, дальнейшие исследования в этой области могут быть направлены на повышение полезности обнаруживаемых аномалий посредством улучшения процесса векторизации (добавление метрик либо использование других подходов), тюнинга или смены используемых методов машинного обучения.

[6] https://github.com/ml-in-programming/kotlin-code-anomaly/

# Список литературы


[1] 111 Billion Lines of New Software Code Will Need to be Secured in 2017. — URL: https://codedx.com/2017/01/23/111-billion-lines-new-software-code-will-need-secured-2017/ (online; accessed: 26.01.2018).

[2] Building Program Vector Representations for Deep Learning / Lili Mou, Ge Li, Yuxuan Liu et al. // CoRR. — 2014. — Vol. abs/1409.3358. — 1409.3358.

[3] Chandola Varun, Banerjee Arindam, Kumar Vipin. Anomaly Detection: A Survey // ACM Comput. Surv. — 2009. — 07. — Vol. 41, no. 3. — P. 15:1–15:58. — URL: http://doi.acm.org/10.1145/1541880.1541882.

[4] Chilowicz M., Duris E., Roussel G. Syntax tree fingerprinting for source code similarity detection // 2009 IEEE 17th International Conference on Program Comprehension. — 2009. — 05. — P. 243–247.

[5] De-anonymizing Programmers via Code Stylometry / Aylin Caliskan-Islam, Richard Harang, Andrew Liu et al. // Proceedings of the 24th USENIX Conference on Security Symposium. — SEC'15. — Berkeley, CA, USA : USENIX Association, 2015. — P. 255–270. — URL: http://dl.acm.org/citation.cfm?id=2831143.2831160.

[6] Estimating the Support of a High-Dimensional Distribution / Bernhard Schölkopf, John C. Platt, John C. Shawe-Taylor et al. // Neural Comput. — 2001. — 07. — Vol. 13, no. 7. — P. 1443–1471. — URL: https://doi.org/10.1162/089976601750264965.

[7] Gousios Georgios. The GHTorrent dataset and tool suite // Proceedings of the 10th Working Conference on Mining Software Repositories. — MSR '13. — 2013. — 05. — P. 233–236. — Best data showcase paper award. URL: /pub/ghtorrent-dataset-toolsuite.pdf.

[8] Hodge V.J., Austin J. A survey of outlier detection methodologies // Artificial Intelligence Review. — 2004. — October. — P. 85–126. — URL: http://eprints.whiterose.ac.uk/767/.

[9] Jolliffe I.T. Principal Component Analysis. Springer Series in Statistics. — Springer, 2002. — ISBN: 9780387954424. — URL: https://books.google.de/books?id=_olByCrhjwIC.

[10] Kotlin 1.2 Released // Kotlin Blog. — URL: https://blog.jetbrains.com/kotlin/2017/11/kotlin-1-2-released/ (online; accessed: 26.01.2018).

[11] LOF: identifying density-based local outliers / Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, Jörg Sander // ACM sigmod record / ACM. — Vol. 29. — 2000. — P. 93–104.

[12] Liu F. T., Ting K. M., Zhou Z. H. Isolation Forest // 2008 Eighth IEEE International Conference on Data Mining. — 2008. — 08. — P. 413–422.

[13] Liu Fei Tony, Ting Kai Ming, Zhou Zhi-Hua. Isolation-Based Anomaly Detection // ACM Trans. Knowl. Discov. Data. — 2012. — 03. — Vol. 6, no. 1. — P. 3:1–3:39. — URL: http://doi.acm.org/10.1145/2133360.2133363.

[14] Milepost GCC: Machine Learning Enabled Self-tuning Compiler / Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon et al. // International Journal of Parallel Programming. — 2011. — Jun. — Vol. 39, no. 3. — P. 296–327. — URL: https://doi.org/10.1007/s10766-010-0161-2.

[15] PSI Files // IntelliJ Platform SDK Developer Guide. — URL: http://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi_files.html (online; accessed: 01.04.2018).

[16] Rousseeuw Peter J., Driessen Katrien Van. A Fast Algorithm for the Minimum Covariance Determinant Estimator // Technometrics. —

1999. — 08. — Vol. 41, no. 3. — P. 212–223. — URL: http://dx.doi.org/10.2307/1270566.

[17] Scikit-learn: Machine Learning in Python / F. Pedregosa, G. Varoquaux, A. Gramfort et al. // Journal of Machine Learning Research. — 2011. — Vol. 12. — P. 2825–2830.

[18] Shippey Thomas. Exploiting Abstract Syntax Trees to Locate Software Defects. — 2015. — 05.

[19] Source Code Authorship Attribution Using Long Short-Term Memory Based Networks / Bander Alsulami, Edwin Dauber, Richard Harang et al. — 2017. — 08. — P. 65–82.

[20] Source Code Metrics: A Systematic Mapping Study / Alberto Varela, Hector Perez-Gonzalez, Francisco Martinez, C Soubervielle-Montalvo. — 2017. — 04. — Vol. 128.

[21] Unsupervised Learning Based Approach for Plagiarism Detection in Programming Assignments / Jitendra Yasaswi, Sri Kailash, Anil Chilupuri et al. // Proceedings of the 10th Innovations in Software Engineering Conference. — ISEC '17. — New York, NY, USA : ACM, 2017. — P. 117–121. — URL: http://doi.acm.org/10.1145/3021460.3021473.