Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Санкт-Петербургский государственный университет»

Кафедра системного программирования

Приходько Станислав Витальевич

Обнаружение аномалий в программах на языке Kotlin методами статического анализа кода

Выпускная квалификационная работа

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

Консультант:
аналитик ООО “Интеллиджей Лабс” Поваров Н. И.

Рецензент:
инженер по тестированию ПО ООО “Интеллиджей Лабс” Петухов В. А.

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering

Stanislav Prikhodko

Anomaly detection in Kotlin programs by static code analysis methods

Graduation Thesis

Scientific supervisor:
Assistant Professor Timofey Bryksin

Adviser:
Analyst, IntelliJ Labs Co. Ltd. Nikita Povarov

Reviewer:
QA Engineer, IntelliJ Labs Co. Ltd. Victor Petukhov

Saint-Petersburg
2019

Оглавление

Введение

В настоящее время активно развивается область разработки программного обеспечения. Каждый день создается огромное количество строк программного кода. Миллионы людей по всему миру занимаются написанием кода, его тестированием, проводят ревью и отладку. Однако результатом деятельности программистов является программный продукт, к которому, как правило, предъявляется достаточно много требований, которые позволяют судить о его качестве и работоспособности. В связи с этим необходимо быть уверенным, что на всех этапах на пути от исходного кода до программного продукта в него не вносится дефектов. Одной из важных задач является тестирование компилятора программы на предмет наличия ошибок, которые могут негативно отразиться на работе программного продукта.

Зачастую при рассмотрении большого количества исходного кода программ можно обнаружить отдельные экземпляры, по тому или иному признаку выделяющиеся на общем фоне рассматриваемого набора данных. Такие примеры нетипичного кода в рамках данной работы будем называть кодовыми аномалиями. Природа их возникновения может быть совершенно различной. Возможно программа содержит проблему, которая кроется в логике разработанного кода, и тогда груз ответственности за возможные неполадки ложится на программиста, написавшего этот код. Однако возможны случаи, когда код не содержит ошибок и полностью корректен, но в силу своей специфики попадает под определение кодовой аномалии. Аномалии в своей сущности показывают, как не принято большинством разработчиков на том или ином языке программирования писать программный код. В связи с этим обнаружение кодовых аномалий может помочь разработчикам языка программирования в решении целого ряда задач. Например, аномалии могут помочь выявить недостатки в дизайне языка программирования, могут указывать на проблемы производительности программ, оптимизаций компилятора, вывода типов, генерации кода, анализа потока данных.

Kotlin[1] — это достаточно молодой и один из самых активно развивающихся языков программирования с быстро растущим сообществом пользователей и значительной экосистемой разнообразных проектов с открытым исходным кодом. Kotlin представляет собой высокоуровневый статически типизированный язык программирования, используемый для разработки на платформе JVM (Java Virtual Machine), также компилируемый в Javascript и в код других платформ. Язык программирования разработан компанией JetBrains [2]. Разработчики языка Kotlin регулярно совершенствуют экосистему языка, в том числе компилятор. Их заинтересованность в выявлении аномалий в программах на языке программирования Kotlin состоит в том, что кодовые аномалии позволят обратить внимание на нестандартные подходы к применению языковых конструкций, дадут информацию о том, как стоит усовершенствовать компилятор, а также сами кодовые аномалии могут стать тестовыми примерами в процессе тестирования компилятора.

В связи с этим задача обнаружения аномалий в программах на языке Kotlin достаточно актуальна и важна как для пользователей, так и для разработчиков языка программирования. В лаборатории JetBrains Research [3] уже не первый год ведутся исследования на тему поиска кодовых аномалий в программах на языке Kotlin. Уже получены первые результаты с примерами кодовых аномалий, часть из которых уже включена в тесты для компилятора языка Kotlin. В связи с актуальностью решения такой задачи в настоящей работе предлагается провести исследование, направленное на расширение существующего решения и создание инструмента для обнаружения новых специфических примеров кодовых аномалий.

[1] http://kotlinlang.org/

[2] http://www.jetbrains.com/

[3] https://research.jetbrains.org/ru/groups/ml_methods/

Постановка задачи

Цель данной работы заключается в том, чтобы расширить существующую систему обнаружения кодовых аномалий в программах на языке Kotlin с целью обнаружения новых классов примеров кода, выделяющихся своим нестандартным содержанием.

Для достижения поставленной цели нужно:

- разработать и реализовать систему обнаружения кодовых аномалий на основе токенов;
- провести апробацию разработанной системы и получить набор кодовых аномалий;
- получить экспертные оценки полезности найденных аномалий.

1. Обзор

1.1. Обнаружение аномалий в различных задачах

Обнаружение аномалий относится к задаче поиска экземпляров данных, не соответствующих ожидаемому представлению. Также аномалией называют отклонение поведения системы от ожидаемого состояния [1]. Задача обнаружения аномалий поставлена в различных областях знаний для решения широкого круга проблем. Например, весьма актуальна задача обнаружения распространения кибератак в сети [2, 3, 4]. Аномальное значение трафика в компьютерной сети может означать, что взломанный компьютер отправляет конфиденциальные данные в неавторизованный пункт назначения. Также не менее значимыми сферами применения методов обнаружения аномалий являются медицина, где происходит определение патологий по медицинским данным [5, 6], а также финансовая сфера, в которой происходит определение подозрительных активностей, фиксация хищений персональных данных и денежных средств [7, 8]. В космической сфере аномальные показания датчиков космических кораблей могут свидетельствовать о неисправности какого-либо компонента космического корабля [9].

В программной инженерии задача обнаружения аномалий ставится с целью поиска ошибок [10], обнаружения вторжений [11], поиска архитектурных недостатков [12], ошибок синхронизации в параллельных программах [13] и т.д. Разработчики различных инструментов, направленных на поиск аномалий в программах, вводят свои собственные определения кодовых аномалий, поэтому их цели, методы и результаты также различаются.

Система GrouMiner [14] предназначена для обнаружения аномальных взаимодействий объектов в программах на языке программирования Java. Подход, описанный в данной работе, заключается в моделировании взаимодействия объектов путем построения ориентированного ациклического графа, узлами которого являются вызовы методов и обращения к полям объектов, а ребра представляют зависимости между

ними. В рамках этой работы используется статический анализ кода. Построение графа взаимодействия объектов происходит на основе абстрактного синтаксического дерева программы. Методы обнаружения аномалий позволяют выявить нестандартные вызовы методов и нетипичные области графа потока управления.

В работе [15] описывается система прогнозирования дефектов в программном обеспечении на языке Java. Дефекты предсказываются для двух различных ситуаций: в ситуации, когда дефект содержится внутри одно проекта, и в ситуации межпроектных дефектов. В обоих случаях процесс обнаружения дефектов состоит из нескольких этапов: этапа векторизации исходных данных, обучения модели классификации и предсказания аномальности поступаемых объектов данных. Данный подход подразумевает статический анализ кода, при котором программа представлена в виде абстрактного синтаксического дерева. Векторизация осуществляется на основе информации о вершинах дерева. В данной работе извлечение семантических признаков на этапе векторизации осуществляется автоматически с использованием алгоритма DBN (Deep Belief Network), которому и посвящена большая часть работы. В качестве алгоритмов классификации в работе используются методы DTree, наивный байесовский классификатор, и логистическая регрессия.

Совершенно другой подход применен группой исследователей при реализации системы DIDUCE [16], где для поиска аномалий в программах на языке Java используется метод динамического анализа кода. При запуске приложения сохраняются значения всех выражений программы. Система генерирует правила для этих выражений, начиная от самых строгих, и ослабляет их по мере появления новых значений выражения. Как только правила нарушаются, это означает, что значение выражения существенно отличается от всех предыдущих, и такое значение выражения объявляется аномальным.

Стоит отметить, что методы динамического анализа кода направлены в первую очередь для обнаружения логических или архитектурных проблем в программах, поэтому они ориентированы в большей степени на пользователей языка программирования. Разработчикам языка про-

граммирования скорее более интересны методы статического анализа программного кода, потому что они направлены на исследование кода без привязки к среде исполнения программы.

В рамках поставленной задачи предлагается производить статический анализ программного кода. Статический анализ кода заключается в анализе программного обеспечения без выполнения запуска исследуемых программ в исполняемой среде. Данный подход является одним из способов анализа кода и широко применяется в различных исследованиях. Одним из преимуществ такого подхода является отсутствие необходимости в подготовке тестового окружения запуска программы в отличие от более сложно организуемого динамического анализа программы. К тому же статический анализ не зависит от сторонних компонентов инфраструктуры и среды исполнения программы, что позволяет изолированно рассматривать программный код без привязки к его окружению, а также в перспективе позволяет находить ошибки, которые могут себя проявить только через длительный срок. Статический анализ кода позволяет выявить ошибки и дефекты на ранних стадиях разработки программного обеспечения, а именно написании кода и его компиляции, что существенно снижает стоимость устранения проблем в программе и упреждает убытки различного характера.

1.2. Система поиска кодовых аномалий для языка Kotlin

Задача обнаружения кодовых аномалий в программах на языке Kotlin уже решалась ранее группой исследователей команды JetBrains Research [17]. Была создана система поиска кодовых аномалий, которая позволяет находить нестандартные примеры программ на языке Kotlin. Архитектура системы изображена на рисунке 1.

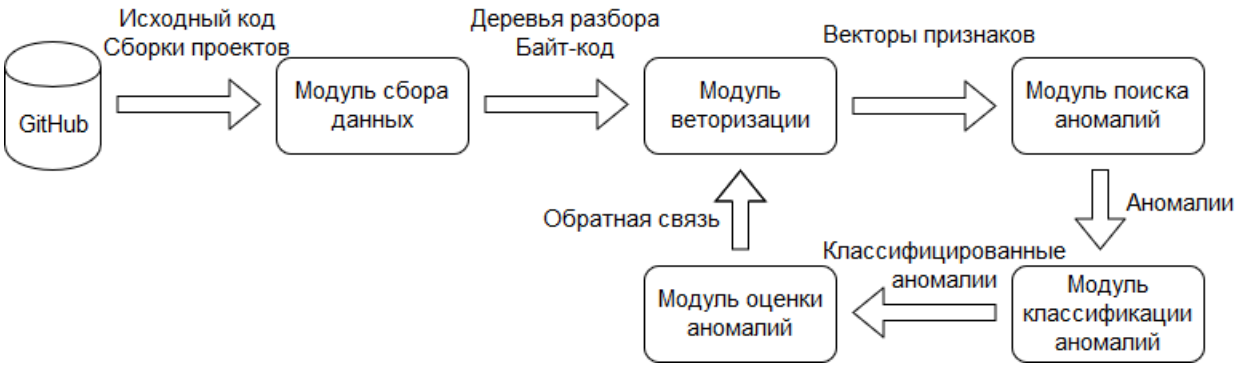

Рис. 1: Архитектура системы

Процесс обнаружения аномалий состоит из нескольких этапов:

- сбор набора данных с исходным кодом программ и сборок проектов с сервиса GitHub [4] с использованием предоставляемого API [5];
- преобразование собранных данных в деревья разбора и списки инструкций JVM;
- извлечение числовых векторов из полученных структур;
- применение методов обнаружения аномалий к извлеченным векторам признаков;
- постобработка данных, заключающаяся в классификации обнаруженных аномалий в соответствии с их типом.

Задача системы состоит в синтаксическом анализе программного кода на языке Kotlin. Данное решение направлено на анализ таких структур представления программы, как дерева разбора и списка инструкций JVM. Дерево разбора является одной из основных структур, описывающих внутреннее представление программы, строящееся на этапе синтаксического анализа в процессе работы компилятора. Инструкции JVM являются структурой, получаемой в результате работы компилятора языка Kotlin. Такой выбор обусловлен целью авторов решения

[4] https://github.com/
[5] https://developer.github.com/v3/

провести поиск нетипичных примеров программного кода, выделяющихся своей нестандартной структурой.

В системе рассмотрено два подхода к векторизации дерева разбора: явное извлечение признаков и автокодирование. Оба подхода обладают рядом преимуществ и недостатков, однако в совокупности они позволяют сделать широкий охват различных видов аномалий.

В качестве алгоритмов обнаружения аномалий в системе используются локальный уровень выброса, изолирующий лес, метод эллипсоидальной аппроксимации данных, нейронный автоэнкодер. Все перечисленные алгоритмы относятся к классу алгоритмов обучения без учителя. Работа первых трех алгоритмов основана на кластеризации данных. Сущность автоэнкодера состоит в сжатии данных с потерями и их последующим восстановлением.

Все найденные аномалии разделяются на классы схожих по смыслу примеров кода. Классификация аномалий в системе происходит вручную путем визуального осмотра обнаруженных кодовых аномалий.

В результате работы системы формируется набор примеров кода на языке программирования Kotlin, которые объявлены аномальными. Также система обладает удобным механизмом визуализации найденных аномалий, которая позволяет красочно представить результат и получить обратную связь от экспертов.

Данная система послужила основой для разрабатываемого в рамках данной работы решения. Для расширения системы обнаружения аномалий с целью получения новых видов нестандартных примеров кода на языке Kotlin принято решение реализовать новые подходы к структуре представления программ, векторизации данных и новые методы обнаружения аномалий.

1.3. Компилятор языка программирования Kotlin

Основным применением языка программирования Kotlin является разработка приложений для платформы JVM. Kotlin является компилируемым в байт-код JVM — список инструкций виртуальной машины

языка программирования Java. Компилятор языка Kotlin состоит из следующих компонентов:

- лексический анализатор, преобразующий последовательность символов исходного кода программы в последовательность токенов;
- синтаксический анализатор, преобразующий набор токенов в дерево разбора, которое в компиляторе языка Kotlin именуется PSI (Program Structure Interface);
- генератор кода инструкций JVM на основе дерева разбора PSI.

Важно заметить, что до этапа генерации байт-кода JVM компилятор способен работать даже с синтаксически и семантически некорректными конструкциями. В связи с этим стоит отметить, что обнаружение ошибок на более ранних стадиях компиляции программы позволит сократить как временные, так и многие другие ресурсоемкие издержки устранения дефектов.

В рамках данной работы предлагается исследовать этап лексического анализа программы, а именно представление программы в виде списка токенов. Токен представляет собой структуру, включающую в себя тип токена и последовательность символов лексемы, который выделяется из исходного кода программы. В связи с этим токен содержит информацию об исходном представлении смысловой единицы языка программирования (лексемы) в программе и о его идентификаторе, который позволяет узнать, какую смысловую нагрузку данный токен несет. В рамках поставленной задачи структура списка токенов содержит всю необходимую информацию о представлении программы для анализа его содержания и выявления нестандартных с точки зрения содержания исходного кода примеров программ.

1.4. Методы векторизации данных

Алгоритмы поиска аномалий, как и большинство методов машинного обучения, принимают в качестве входных данных наборы числовых

векторов. В связи с этим необходимо обозначить ряд методов, которые способны преобразовать данные в векторы числовых признаков.

В случае рассмотрения кода в текстовом виде или в виде списка токенов для векторизации применимы методы обработки естественного языка (NLP), такие как мешок слов (bag-of-words) и мешок n-грамм (bag of n-grams) [18, 19]. Однако существуют и другие техники векторизации, такие как применение функций хэширования и подсчета метрик TF-IDF. При представлении кода в виде дерева разбора существует большое количество способов явного и неявного извлечения признаков. Подход явного извлечения признаков [20, 21] хорош тем, что обычно полученные признаки легко интерпретировать и понять. Неявные признаки [22, 23] более трудны для понимания, однако такой подход в отдельных случаях способен определять сложные свойства кода, такие как, например, семантические зависимости.

В связи с рассмотрением представления программы в виде структуры списка токенов в рамках данной работы принято решение использовать подход к векторизации данных, использующий методы обработки естественного языка (Natural Language Processing) в вариации их применения к структуре токенов. Такой подход, как правило, позволяет исследовать семантику имен функций и переменных без учета структуры программы.

1.5. Алгоритмы обнаружения аномалий

Для анализа построенных векторов и выделения аномальных данных необходимо применить методы обнаружения аномалий. В области машинного обучения под аномалией понимают отклонение поведения системы от ожидаемого.

Аномалии можно разделить на 3 вида [24]:

- точечные аномалии соответствуют случаям, когда отдельный экземпляр данных является аномальным по отношению к остальным данным (на рисунке 2 точки A_1 и A_2, а также область точек A_3 являются аномальными по отношению к областям R_1 и R_2);

- контекстуальные аномалии соответствуют случаям, когда отдельный экземпляр данных является аномальным только в определенном контексте; аномальное поведение определяется с помощью значений контекстных атрибутов, например, временем наблюдения (на рисунке 3 в точке A прослеживается аномалия в контексте с точками $N_1 - N_5$);

- коллективные аномалии соответствуют случаям, когда совокупность экземпляров данных аномальна по отношению ко всему набору данных (на рисунке 4 область A является коллективной аномалией).

Для решения поставленной задачи предполагается осуществлять поиск точечных аномалий, т.к. объекты анализа, списки токенов, рассматриваются изолированно друг от друга и не содержат контекстных атрибутов. В качестве контекстуальных аномалий могут рассматриваться, например, отдельные части списка токенов в контексте всего списка или отдельные токены в контексте других токенов. Однако, в соответствии с поставленной задачей, будут рассматриваться аномалии, представляющие собой список токенов целиком.

В рамках поставленной задачи обнаружения новых классов аномалий все данные рассматриваются как неразмеченные, т.е. без информации о том, какие примеры кода являются аномальными и к какому классу аномалий они принадлежат. А значит для решения задачи поиска аномалий уместно использовать алгоритмы обучения без учителя. Рассмотрим существующие методы обучения без учителя, позволяющие осуществлять обнаружение точечных аномалий.

Одноклассовый метод опорных векторов (One-class SVM) [25] — представитель методов опорных векторов, который относится к алгоритмам классификации и предполагает наличие только одного класса. Данный алгоритм позволяет проводить обучение без учителя. Метод состоит в том, что набор исходных векторов отделяется от точки начала координат с помощью построения разделяющей гиперплоскости в пространстве более высокой размерности. По результатам работы метода пред-

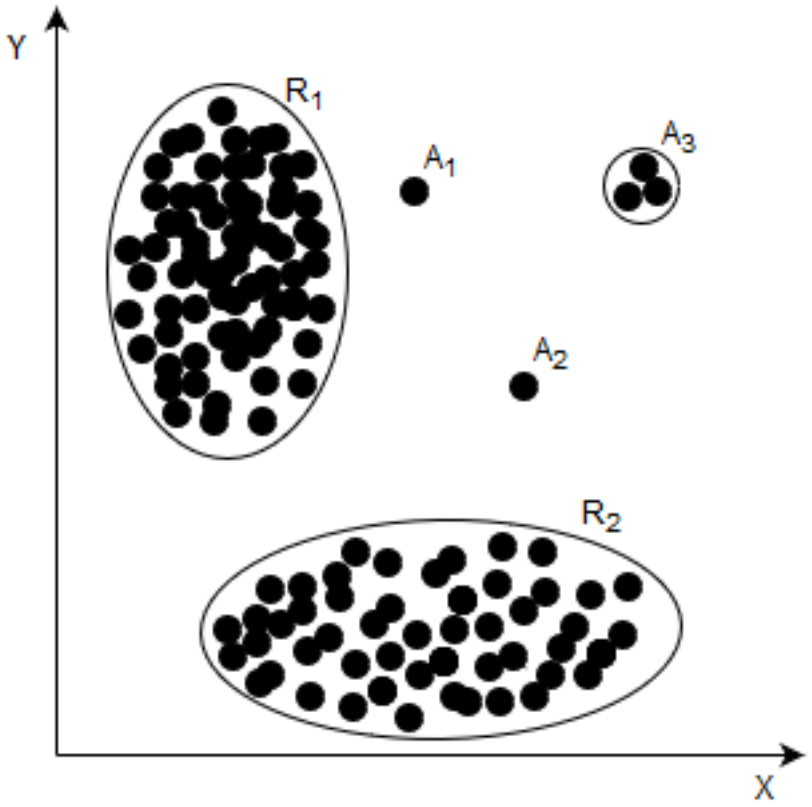

Рис. 2: Пример точечных аномалий

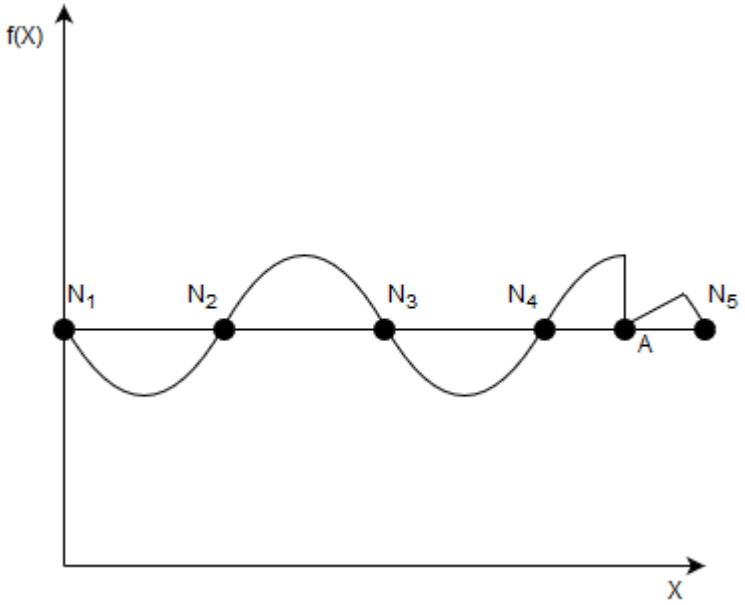

Рис. 3: Пример контекстуальных аномалий

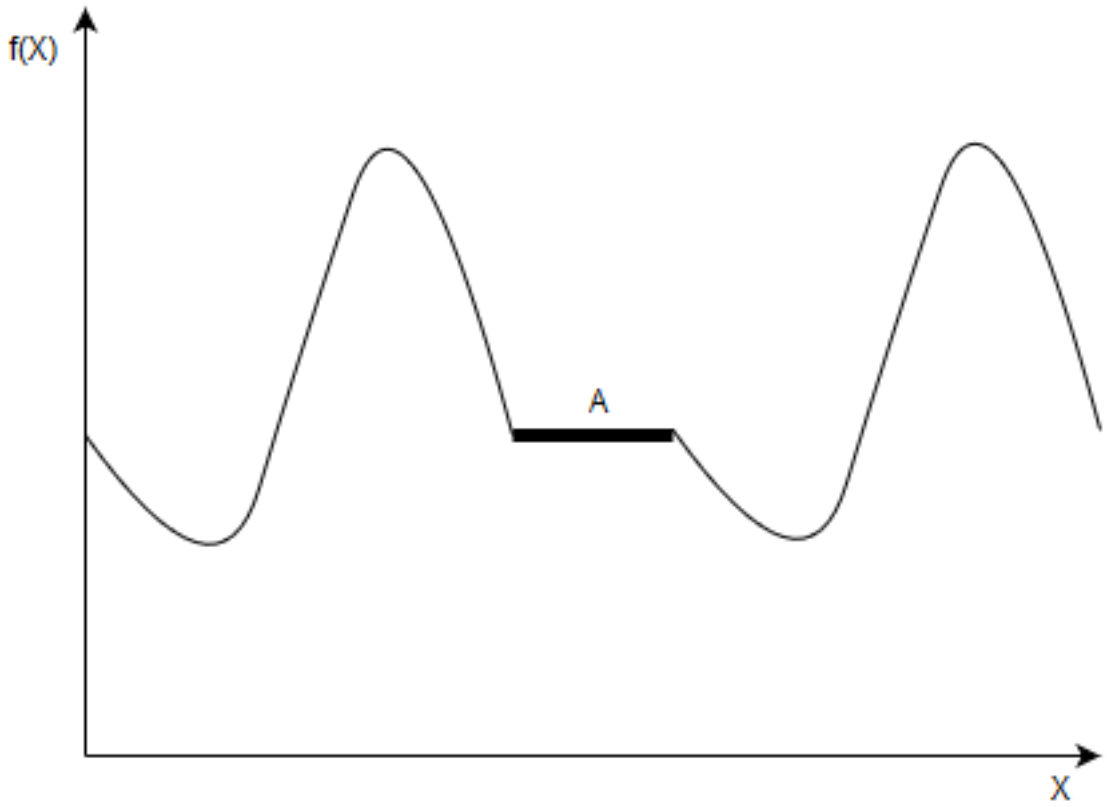

Рис. 4: Пример коллективных аномалий

полагается получение двух групп данных: первая группа состоит из объектов, которые относятся к единственному классу, и эти данные не являются аномальными, а вторая группа состоит из объектов, которые не удалось определить к этому классу, и они объявляются аномальными. В дополнение к этому метод предполагает получение числовой оценки аномальности.

Основанная на плотности пространственная кластеризация для приложений с шумами (Density-based spatial clustering of applications with noise, DBSCAN) [26] — алгоритм кластеризации, основанной на плотности расположения точек. Суть алгоритма состоит в том, что в пространстве исходных векторов алгоритм группирует вместе точки, которые тесно расположены, т.е. те точки, у которых много близких соседей, помечая как аномалии точки, которые находятся одиноко в областях с малой плотностью, т.е. ближайшие соседи которых расположены далеко. При этом количество кластеров не задается явно, а вычисляется в результате работы алгоритма.

Иерархическая основанная на плотности пространственная кластеризация для приложений с шумами (Hierarchical Density-based spatial clustering of applications with noise, HDBSCAN) [27] — алгоритм кластеризации, расширяющий алгоритм DBSCAN путем преобразования его в иерархический алгоритм кластеризации, использующий методы извлечения плоской кластеризации на основе стабильности кластеров.

Метод k-средних (k-means) [28] — алгоритм кластеризации, который состоит в разбиении набора элементов на заданное количество кластеров. Метод заключается в итеративном вычислении центра масс кластеров с целью минимизации суммарного квадратичного отклонения точек кластеров от их центров. Итерации останавливаются тогда, когда расстояния внутри кластеров перестают изменяться.

Локальный уровень выброса (local outlier factor, LOF) [29] — алгоритм обнаружения аномалий, суть которого состоит в оценке локальной плотности объектов и сравнения с локальными плотностями их соседей. Точки, в которых плотность существенно отличается от плотности соседей, объявляются аномальными.

Репликаторные нейронные сети [30] могут использоваться с целью обнаружения аномалий в данных. Суть работы данного алгоритма состоит в восстановлении исходных данных из промежуточного представления, сгенерированного автоэнкодером. Степень аномальности в данном случае определяется размером потерь, полученных при декодировании данных.

Статистические методы [31] основаны на идее сопоставления данных некоторому статистическому распределению. Степень аномальности определяется как величина отклонения экземпляра данных от проверяемого распределения. Некоторые методы данной группы требуют предположения о распределении данных, другие могут вычислять возможное распределение данных самостоятельно.

Все перечисленные методы подходят для решения задачи обнаружения точечных аномалий в неразмеченных данных. Они все относятся к классу задач обучения без учителя и все они позволяют оценить в том или ином виде степень аномальности объекта данных. Однако в

данной работе предлагается сделать акцент на использовании методов локальный уровень выброса, одноклассовый метод опорных векторов, DBSCAN и HDBSCAN. Такой выбор алгоритмов обусловлен стремлением использовать в решении методы, ранее не применяемые в системе обнаружения кодовых аномалий на языке Kotlin, поэтому их использование может позволить находить новые классы аномалий и расширит инструментарий системы поиска кодовых аномалий. Несмотря на это метод локальный уровень выброса ранее использовался системой. На него пал выбор по причине сравнительно небольших временных затрат, которые необходимы для его работы, и высокого качества распознавания аномалий, показанного ранее в рамках апробации системы обнаружения кодовых аномалий.

2. Разработанное решение

Схематичное изображение решения задачи обнаружения аномалий изображено на рисунке 5. Рассмотрим поэтапно решение данной задачи.

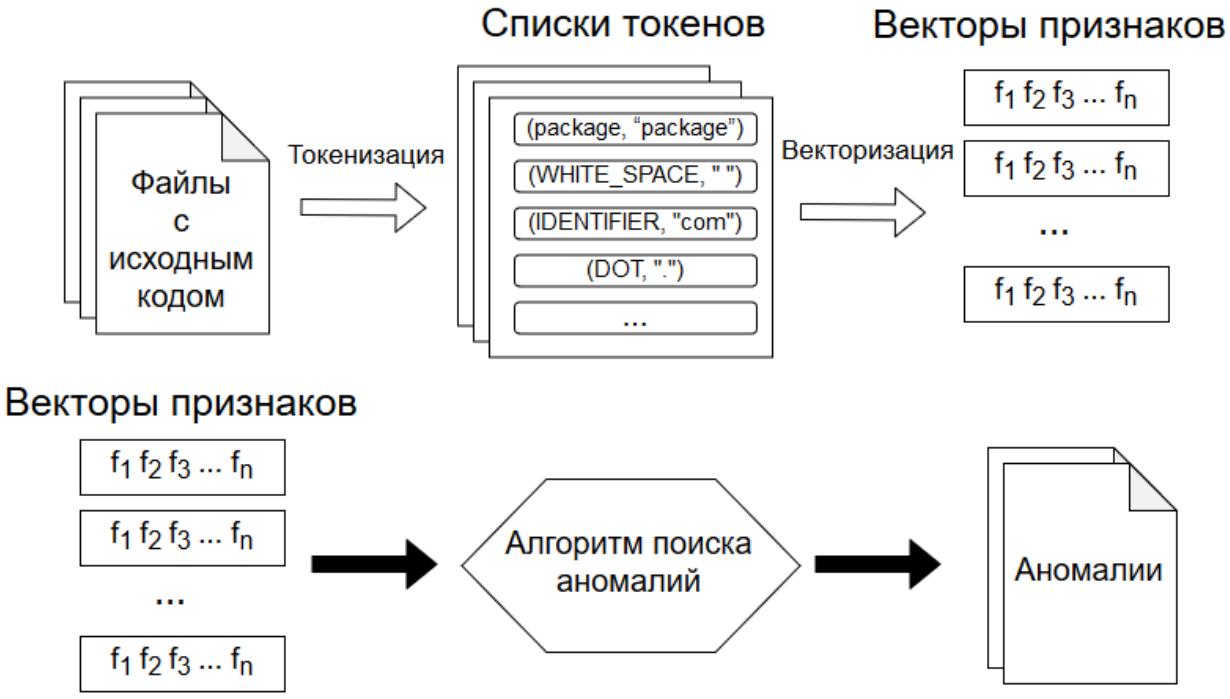

Рис. 5: Решение задачи

2.1. Токенизация программы

Для построения представления программы в виде структуры "списка токенов" используется компилятор языка программирования Kotlin. С целью преобразования исходного кода программы на языке Kotlin в список токенов был склонирован репозиторий с компилятором языка Kotlin, расположенный на общедоступном ресурсе GitHub [6]. В данном репозитории в рамках тестовой инфраструктуры была добавлена возможность вывода списка токенов в процессе компиляции программы. Путь к директории, в которой необходимо запустить генерацию списка токенов по исходному коду, задается отдельным параметром. В результате работы данного модуля рядом с каждым экземпляром исходного кода на языке Kotlin создается файл с расширение "txt", в котором в

[6]https://github.com/PrikhodkoStanislav/kotlin/tree/cad-sandbox-tokens

виде списка на каждой новой строке перечислены токены. Каждый токен представляет собой структуру из двух элементов: (тип токена, лексема). В результирующем файле элементы разделены пробелом. Для удобства дальнейшей обработки данных был разработан инструмент [7], который на основе полученного файла создает два новых файла: один со списком типов токена, имеющий расширение “token”, второй со списком значений лексем, имеющий расширение “value”.

2.2. Векторизация данных

С целью охватить более широкий круг потенциальных аномалий, которые будут найдены с помощью анализа списка токенов, принято решение рассмотреть два подхода к представлению программы: список типов токенов и список значений лексем. В каждом из этих подходов предлагается применять различные методы векторизации. Схема преобразования списка токенов в набор векторов представлена на рисунке 6.

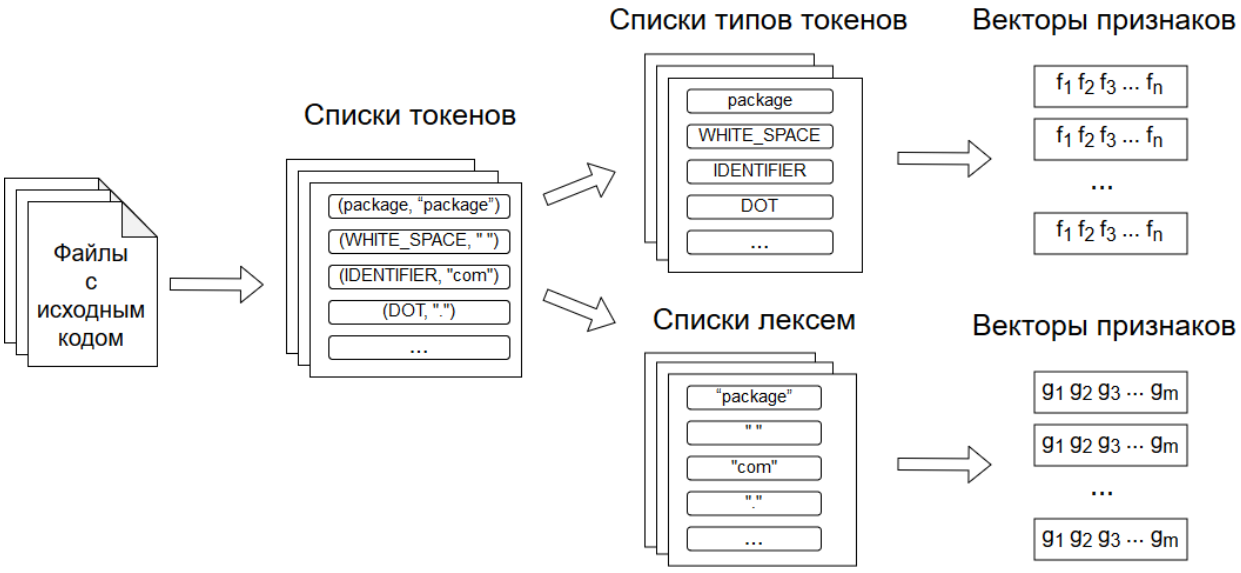

Рис. 6: Векторизация токенов

[7]https://github.com/PrikhodkoStanislav/code-anomaly-detection/tree/master/TokenExtractor

2.2.1. Векторизация списка типов токенов

Для применения методов обнаружения аномалий необходимо собранный набор списков типов токенов преобразовать в вид числовых векторов. Для решения данной задачи применяется подход к векторизации текстов, используемый в области обработки естественного языка (NLP). Модель, называемая “мешок слов” (bag-of-words), содержит идею представления данных в виде “мешка” или множества слов. Порядок и грамматика слов в данной модели не учитываются. Применительно к нашим данным словом будет являться идентификатор типа токена. Множество слов будет состоять из всевозможных типов токенов, которые встречаются во всем наборе данных.

Одна из базовых моделей векторизации — заполнение вектора значениями частоты вхождения слова в файле. Для реализации данного метода векторизации необходимо получить множество всех слов. Для этого разработан отдельный модуль, совершающих обход всего набора данных и построение множества слов датасета. Затем модуль для каждого файла строит вектор с подсчетом количества вхождений слов.

Однако, метод подсчета частоты вхождения слов в документе или файле обладает недостатком, связанным со значительным влиянием слов, которые встречаются в файле очень часто. Решением этой проблемы является метод однократного кодирования (one-hot encoding) или метод логического векторного кодирования. Он заключается в том, что для каждого файла строится вектор из всевозможных слов, в котором помечается, входит ли в данный файл слово или нет. Другими словами, каждый элемент вектора отражает либо наличие либо отсутствие слова в файле. Метод однократного кодирования уменьшает проблему дисбаланса распределения слов, упрощая документ до его составляющих компонентов.

Модель “мешок слов” описывает файл автономно, без учета контекста других файлов набора данных. Возникает идея рассмотреть относительную частоту или редкость вхождения слов в файлах против частоты их вхождения в других файлах. Для воплощения этой идеи суще-

ствует метод TF-IDF (Term Frequency – Inverse Document Frequency). TF-IDF представляет собой статистическую меру, используемую для оценки важности слова в наборе данных. Мера состоит из двух других мер: TF и IDF.

TF представляет собой отношение числа вхождений слова к общему числу слов в файле. Вычисляется по формуле:

$$TF(t,d) = \frac{n_t}{\sum_k n_k}, \tag{1}$$

где n_t есть число вхождений слова t в файл, в знаменателе — общее число слов в файле.

IDF — это инверсия частоты, с которой слово встречается в файле. Вычисляется по формуле:

$$IDF(t,D) = \log \frac{|D|}{|\{d_i \in D | t \in d_i\}|}, \tag{2}$$

где |D| — общее число файлов, $|\{d_i \in D | t \in d_i\}|$ — число файлов, в которых встречается слово t.

Мера TF-IDF является произведением двух мер и вычисляется по формуле:

$$TF - IDF(t,d,D) = TF(t,d) \times IDF(t,D). \tag{3}$$

Для каждого слова в файле происходит подсчет метрик TF, IDF и TF-IDF. Все три значения заносятся в результирующий вектор. В случае отсутствия слова в файле, соответствующие значения в векторе заполняются нулями.

Таким образом, для каждого типа токена в каждом файле вычисляется 5 метрик: частота вхождения, присутствие в файле, метрики TF, IDF, TF-IDF. В результате получается вектор состоящий из $5 \cdot n$ элементов, где n — количество различных типов токенов во всем наборе данных.

Существуют реализации описанных способов векторизации в библиотеках Scikit-learn и Gensym, однако они предполагают загрузку все-

го набора данных в оперативную память для обработки, что не представляется возможным при достаточно большом объеме данных. В связи с этим был разработан независимый инструмент векторизации, созданный на языке программирования Python с использованием стандартных библиотек. Работа инструмента заключается в вычислении метрик путем последовательного обхода всего набора данных. Сперва происходит обход всех списков токенов с целью формирования общего множества, состоящего из всех типов токенов, содержащихся в векторизуемом наборе данных. Затем в процессе повторного прохода всего набора токенов осуществляется подсчет метрик для построения вектора. Все построенные вектора записываются в результирующий файл, имеющий расширение “tsv”.

Пример токенизации исходного кода программы изображен на рисунке 7. Пример векторизации типов токенов показан в таблице 1.

2.2.2. Векторизация списка лексем

Для векторизации списка лексем предлагается использовать схожие идеи с теми, которые применялись для векторизации списка типов токенов. Однако, количество лексем в достаточно большом наборе данных будет запредельно большим. В связи с этим предлагается строить векторы, соответствующие суммарным и усредненным статистикам по лексемам для каждого типа токена. В качестве числовых статистик, которые характеризуют значения лексем, предлагается считать длину лексемы и ее хэш-код.

Существует несколько подходов к вычислению усредненных значений статистик. Будем использовать среднее арифметическое и логарифм среднего арифметического для нормализации значений. Такой выбор функций усреднения сделан по причине необходимости унифицировать большой набор чисел, которые весьма велики по своему значению. Хэш-код лексемы стандартной библиотеки языка программирования Python содержит 19 знаков, что делает число очень большим и громоздким. Применение функции логарифмирования позволяет сократить число до вещественного числа с двузначной целой частью.

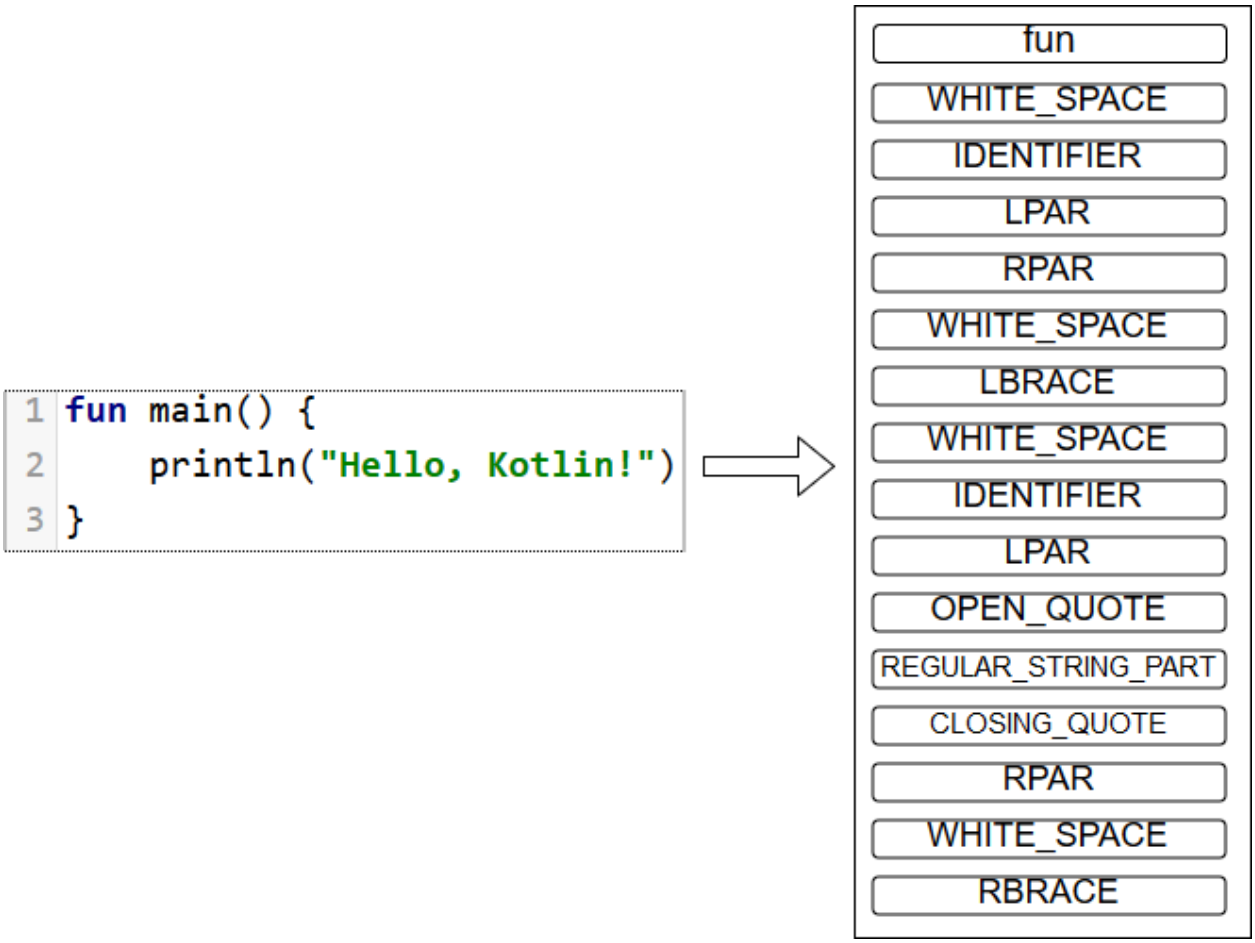

Рис. 7: Пример токенизации программы

№	Тип токена	Count	Bin	TF	IDF	TF-IDF
1	fun	1	1	0.06	0.4	0.03
2	WHITE_SPACE	4	1	0.25	0.1	0.03
3	IDENTIFIER	2	1	0.13	0.1	0.01
4	LPAR	2	1	0.13	0.1	0.01
5	RPAR	2	1	0.13	0.1	0.01
6	LBRACE	1	1	0.06	0.4	0.03
7	OPEN_QUOTE	1	1	0.07	0.5	0.03
8	REGULAR_STRING	1	1	0.06	0.8	0.05
9	CLOSING_QUOTE	1	1	0.07	0.5	0.03
10	RBRACE	1	1	0.06	0.4	0.03
11	package	0	0	0	0	0
12	DOT	0	0	0	0	0

Таблица 1: Пример векторизации типов токенов

Таким образом, для каждого типа токена будет посчитано 7 статистик: присутствие в файле, суммарные длина и значение хэш-кода, среднее арифметическое длины и модуля хэш-кода и логарифм среднего арифметического длины и модуля хэш-кода. Всего вектор для каждого файла будет содержать $7 \cdot n$ элементов, где n — количество различных типов токенов во всем наборе данных.

Пример преобразования исходного кода программы в список лексем изображен на рисунке 8. Пример векторизации списка лексем показан в таблице 2. Условными обозначениями показаны статистики, которые вычисляются на основе построенных лексем. В графе “B” указано значение, обозначающее присутствует ли данный тип токена в файле или нет. В графе “SL” указана суммарная длина значений данного типа, “AL” — средняя длина значений данного типа, “LL” — логарифм средней длины данного типа. В графе “SH” размещается суммарное значение хэш-кодов лексем данного типа, “AH” — среднее значение хэш-кодов данного типа, “LH” — логарифм среднего значения хэш-кодов данного типа.

2.3. Обнаружение аномалий

Для обнаружения аномалий выбраны методы, основанные на кластеризации. Это методы локальный уровень выброса, одноклассовый метод опорных векторов, DBSCAN и HDBSCAN. Каждый из перечисленных алгоритмов запускается на двух наборах данных: один для списков типов токенов, другой для списков лексем. В результате работы каждого метода обнаружения аномалий образуется два набора примеров программного кода, объявленного аномальным. В качестве реализации методов обнаружения аномалий была выбрана стабильная реализация алгоритмов, входящая в состав библиотеки Scikit-learn.

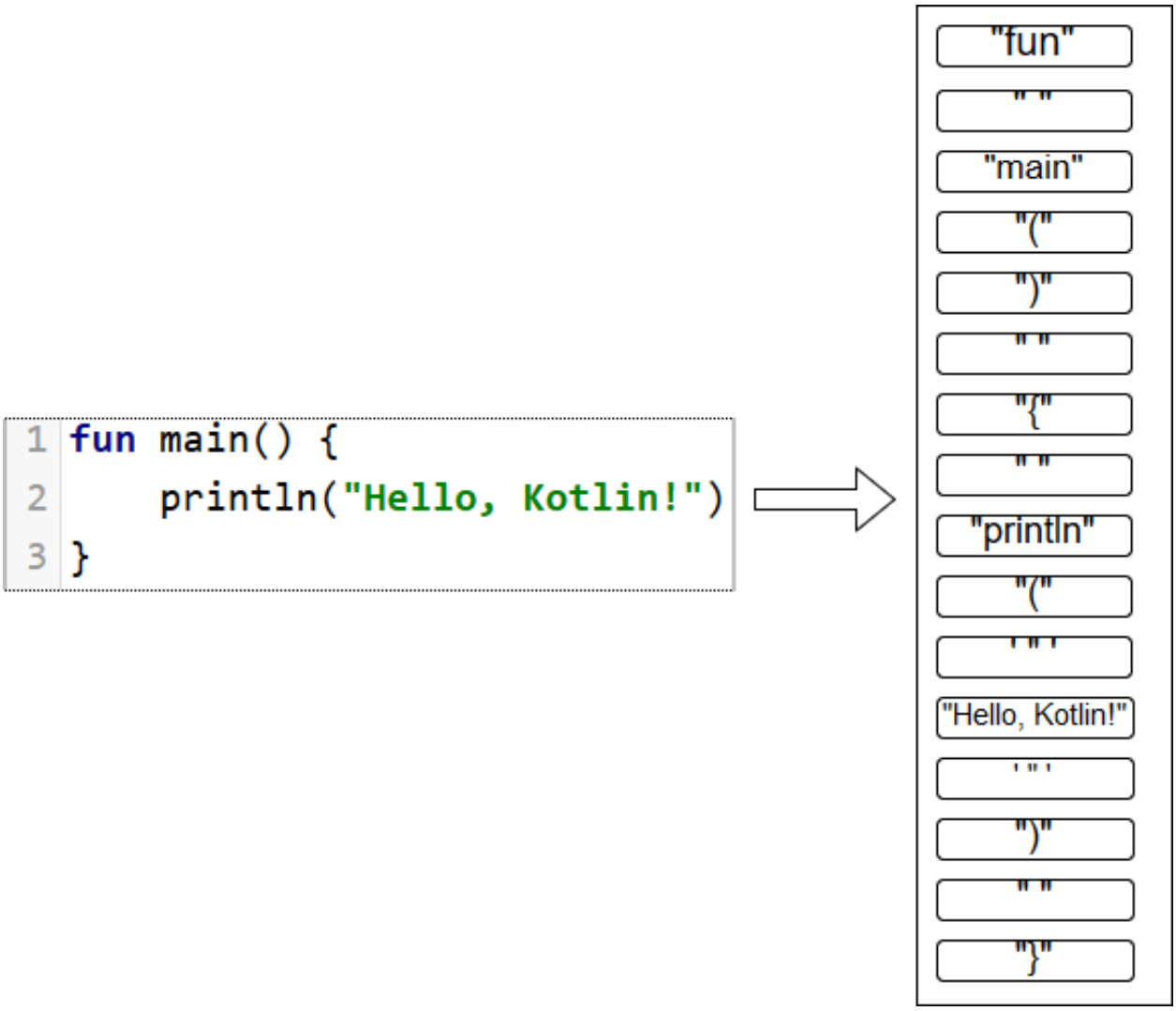

Рис. 8: Пример лексемизации программы

№	Тип токена	B	SL	SH	AL	AH	LL	LH
1	fun	1	3	3..3410	3.0	3..3410	1.1	42.75
2	WHITE_SPACE	1	4	2..1200	1.0	7..7800	0.0	43.41
3	IDENTIFIER	1	11	4..5222	5.5	2..2611	1.6	42.26
4	LPAR	1	2	8..6732	1.0	4..3366	0.0	42.89
5	RPAR	1	2	1..7168	1.0	7..8584	0.0	43.45
6	LBRACE	1	1	3..2890	1.0	3..2890	0.0	42.62
7	OPEN_QUOTE	1	1	2..4926	1.0	2..4926	0.0	42.27
8	REGULAR_STRING	1	14	5..6593	14.0	5..6593	2.6	43.24
9	CLOSING_QUOTE	1	1	2..4926	1.0	2..4926	0.0	42.27
10	RBRACE	1	1	4..3746	1.0	4..3746	0.0	42.99
11	package	0	0	0	0.0	0	0.0	0.0
12	DOT	0	0	0	0.0	0	0.0	0.0

Таблица 2: Пример векторизации списка лексем

2.4. Кластеризация аномалий

Для облегчения процесса разбора найденных аномалий разработан инструмент [8], позволяющий автоматизированно разделить собранный набор примеров исходного кода на группы. В основе работы инструмента заключен метод кластеризации данных k-means, который позволяет сгруппировать экземпляры набора данных в кластеры наиболее похожих друг на друга элементов путем минимизации суммарного квадратичного отклонения точек кластеров от центров этих кластеров. Для векторизации исходного кода применен метод токенизации программы и подсчет статистики оценки важности токенов TF-IDF. В качестве параметра метод кластеризации принимает количество кластеров, на которые будет произведено разделение исходного набора данных. Данный инструмент позволяет разделить найденные аномалии на группы близких по содержанию примеров исходного кода, что упрощает процесс анализа полученных аномалий за счет сокращения примеров, которые необходимо рассмотреть для формирования описания найденных аномалий. Особенно существенна польза данного инструмента при обнаружении большого количества примеров аномального кода.

[8] https://github.com/PrikhodkoStanislav/code-anomaly-detection/tree/master/CodeClustering

3. Апробация

3.1. Сбор данных

Для проведения апробации первоочередной задачей становится сбор набора исходного кода программ, в котором будет происходить обнаружение кодовых аномалий. С помощью модуля сборки репозиториев системы поиска кодовых аномалий удалось получить 96101 репозиторий с ресурса GitHub. Каждый такой репозиторий содержит код на языке программирования Kotlin. Всего было собрано 1.8 млн файлов с исходным кодом на Kotlin. На основе каждого файла был построен список токенов, который затем был преобразован в два списка: список типов токенов и список лексем. Итого получилось два набора данных: один состоит из 1.8 млн. списков типов токенов, другой из 1.8 млн. списков лексем.

3.2. Векторизация полученных данных

Перед векторизацией полученных данных был собран список, состоящий из различных типов токенов, присутствующих в собранном наборе данных. Всего получилось 130 различных типов токенов. Наиболее часто употребляемыми типами токенов оказались WHITE_SPACE, IDENTIFIER, DOT, LPAR, package.

Затем к каждому из наборов данных был применен метод векторизации. На основе списка типов токенов получился набор векторов размерности $5 \cdot n = 5 \cdot 130 = 650$. Для списка лексем набор векторов получил размерность $7 \cdot n = 7 \cdot 130 = 910$.

3.3. Запуск алгоритмов обнаружения аномалий

На полученных наборах векторов были запущены алгоритмы обнаружения аномалий: локальный уровень выброса, одноклассовый метод опорных векторов, DBSCAN и HDBSCAN. В результате нескольких раундов экспериментов путем визуальной оценки подвыборки потенци-

альных аномалий размером около 50 были выбраны следующие параметры для запуска алгоритмов.

Локальный уровень выброса: $n\_neighbors = 20$ (количество соседей, относительно которых считается фактор локальных выбросов), $contamination = 0.0001$ (доля данных, объявляющихся аномалиями).

Одноклассовый метод опорных векторов: $kernel = ``rbf''$ (тип ядра, используемый в алгоритме). В результате работы алгоритма была выделена доля аномалий, составляющая 0.0001 от всего набора данных.

DBSCAN: $eps = 0.5$ (максимальное расстояние между двумя объектами, которые будут рассматриваться как элементы одного кластера).

HDBSCAN: $min\_cluster\_size = 15$ (наименьшее количество выделяемых кластеров).

Параметры выделения доли аномальных точек для всех алгоритмов были выбраны таким образом, чтобы в результате работы алгоритма было выделено не более 200 экземпляров данных, объявленных аномалиями. Это сделано по причине того, что система обнаружения аномалий направлена на выделение обозримого человеком числа аномальных примеров кода.

3.4. Кластеризация найденных аномалий

С помощью разработанного инструментария удалось собрать набор из 1237 уникальных примеров аномального кода. Все найденные аномалии были разделены на 23 группы при помощи инструмента автоматизированной кластеризации исходного кода. Такой выбор количества групп обусловлен тем, что экспертная оценка результатов весьма ресурсоемкий процесс, поэтому необходимо получить достаточно небольшую выборку наиболее показательных примеров кодовых аномалий.

3.5. Составление отчета о найденных аномалиях

В процессе формирования отчета для разработчиков языка программирования Kotlin с целью получения экспертной оценки полезности найденных аномалий был произведен отбор 47 наиболее показа-

тельных примеров среди обнаруженных кодовых аномалий. Из каждой группы в зависимости от ее численности было выбрано по 2 – 3 примера исходного кода программ. Каждая группа примеров аномалий была сопровождена кратким описанием, характеризующим отличительные черты данного класса.

3.6. Получение экспертных оценок

Сформированный отчет с примерами аномального кода на языке Kotlin был передан для оценки двум независимым экспертам, которые являются разработчиками языка программирования Kotlin. Оценка производилась независимо на подмножестве всего набора аномалий, включенного в отчет, по пятибалльной шкале. На основе оценок, выставленных экспертами конкретным примерам аномального кода, была сформирована усредненная оценка для каждого класса найденных аномалий. Результаты экспертной оценки представлены в таблице 3. В графе "Описание класса" указана краткая характеристика оцениваемого класса аномалий. В графе "T" знаком "+" обозначены те группы аномалий, элементы которых были обнаружены в результате анализа типов токенов, в графе "L" знаком "+" обозначены группы, элементы которых были обнаружены в процессе анализа лексем. Графы "$\mathbf{Q_1}$" и "$\mathbf{Q_2}$" содержат усредненные оценки соответствующих экспертов для входящих в состав группы аномалий. Знак "–" указывает на то, что ни один экземпляр соответствующего класса не попал в оцениваемую выборку для соответствующего эксперта. В графе "Q" представлена средняя оценка для класса аномалий, рассчитываемая как среднее арифметическое значение оценок экспертов.

По результатам экспертной оценки можно сделать вывод, что достаточно небольшое количество классов и примеров аномалий заинтересовало экспертов, проводящих оценку отчета с аномалиями. Лишь 2 класса из 23 получили результирующие оценки 4 и 5. Оценку 3 и выше удалось получить 7 из 23 полученных классов. Стоит отметить, что оценки экспертов довольно серьезно разнятся. Если рассматривать

№	Описание класса	T	L	Q_1	Q_2	Q
1	Много выражений expect	+	+	5	–	5
2	Много перечислений (enum)	+		4	–	4
3	Много типов аргументов	+		5	2	3.5
4	Много классов	+		4	3	3.5
5	Арифметические выражения	+		3	3	3
6	Динамические выражения		+	3	–	3
7	Длинные строки		+	3	–	3
8	Много длинных циклов	+		4	1	2.5
9	Функции со свойствами	+		4	1	2.5
10	Большие массивы данных	+		2	3	2.5
11	Строковое интерполирование	+	+	3	1	2
12	Строковые подстановки		+	3	1	2
13	Много выражений when	+		3	1	2
14	Много присвоений	+		2	2	2
15	Замена имен типов		+	2	2	2
16	Много условных выражений	+		2	–	2
17	Шестнадцатеричные числа		+	2	–	2
18	Выражения assert	+	+	1	1	1
19	Интервалы	+	+	1	1	1
20	Именованные аргументы	+		1	1	1
21	Многострочные строки		+	1	1	1
22	Проверки на null		+	1	1	1
23	Нестандартные кодировки		+	–	1	1

Таблица 3: Экспертная оценка классов аномалий

оценки только первого эксперта, то результаты становятся более положительными. 2 класса получили наивысшую оценку, 6 из 23 классов получили оценки 4 и 5, а 12 классов, то есть больше половины, получили оценки 3 и выше. Что касается оценок второго эксперта, то они оказались существенно ниже оценок первого, и не превышают 3-х баллов. Если обобщить результаты, то можно сделать вывод, что аномалии, полученные на основе анализа типов токенов, оказались более интересными для экспертов, нежели аномалии, полученные при анализе лексем. Часть найденных аномалий получила довольно высокие оценки от экспертов.

3.7. Сравнение результатов с похожими работами

Параллельно с описываемой работой выполнялась работа на тему "Исследование аномалий работы компилятора языка Kotlin", в рамках которой проводилось исследование поиска условных кодовых аномалий на основе байт-кода, генерируемого компилятором языка Kotlin для платформы JVM. В результате был также сформирован отчет с набором аномалий, найденных с помощью анализа JVM байт-кода, и передан экспертам для оценки полезности аномалий. Экспертами выступили разработчики языка программирования Kotlin, которые оценивали аномалии на основе анализа токенов. В результате работы был получен набор из 18 классов аномалий.

Ранее были получены оценки для работ, направленных на поиск аномалий в программах на языке Kotlin по дереву разбору и байт-коду, основанные на явном извлечении признаков (работа "Детектор аномалий в программах на языке Kotlin", в результате которой было выделено 23 класса аномалий) и автокодировании (работа "Обнаружение проблем производительности в программах на языке программирования Kotlin с использованием статического анализа кода", в результате которой было обнаружено 29 классов аномалий) в процессе векторизации данных. Однако, оценки результатов данных работ были выставлены другими экспертами, не теми, которые оценивали аномалии, найденные на основе анализа токенов и байт-кода.

Сравнительная таблица результатов исследований представлена в таблице 4. Оценки описываемой работы представлены в графе $\mathbf{Q_1}$. В графе $\mathbf{Q_2}$ перечислены оценки исследования в рамках поиска аномалий на основе байт-кода. В графе $\mathbf{Q_3}$ указаны оценки работы по поиску аномалий по дереву разбора и байт-коду, основанные на векторизации автокодированием. В графе $\mathbf{Q_4}$ перечислены оценки работы обнаружения аномалий на основе явного выделения признаков по дереву разбора. Оценки перечислены для классов, которые были получены в рамках данной работы.

Как видно из результатов сравнения, большинство обнаруженных

№	Описание класса	Q_1	Q_2	Q_3	Q_4
1	Много выражений expect	5	–	–	–
2	Много перечислений (enum)	4	–	4	–
3	Много типов аргументов	3.5	–	–	–
4	Много классов	3.5	–	–	–
5	Арифметические выражения	3	1	–	–
6	Динамические выражения	3	–	–	–
7	Длинные строки	3	–	–	–
8	Много длинных циклов	2.5	2	2	2
9	Функции со свойствами	2.5	–	–	–
10	Большие массивы данных	2.5	–	–	–
11	Строковое интерполирование	2	–	–	–
12	Строковые подстановки	2	–	–	–
13	Много выражений when	2	–	–	–
14	Много присвоений	2	–	–	–
15	Замена имен типов	2	–	–	–
16	Много условных выражений	2	–	–	–
17	Шестнадцатеричные числа	2	–	–	–
18	Выражения assert	1	–	–	–
19	Интервалы	1	–	–	–
20	Именованные аргументы	1	–	–	–
21	Многострочные строки	1	–	–	–
22	Проверки на null	1	–	–	–
23	Нестандартные кодировки	1	–	–	–

Таблица 4: Сравнение результатов

классов новые и не были выделены ранее. Среди тех классов, которые удалось также выделить в результате других исследований, прослеживается улучшение экспертных оценок. Это говорит о том, что подход к обнаружению аномалий на основе анализа токенов успешно применим к задаче обнаружения кодовых аномалий.

3.8. Вывод

В результате проведенных экспериментов удалось получить набор новых классов аномалий, что являлось одной из первоочередных задач исследования. Аномалии, получившие оценки 4 и 5, были приняты раз-

работчиками языка программирования Kotlin в рассмотрение с целью включения в тесты для компилятора языка Kotlin.

Заключение

В ходе работы были получены следующие результаты:

- разработана и реализована система [9] обнаружения кодовых аномалий на основе токенов, включающая модуль векторизации токенов, модуль поиска аномалий и модуль автоматизированной кластеризации обнаруженных аномалий;
- проведена апробация разработанной системы на наборе данных, содержащих 1.8 млн файлов с ресурса GitHub с исходным кодом на языке Kotlin, в результате которой получен набор примеров аномального кода;
- получены экспертные оценки полезности наиболее интересных примеров найденных аномалий от разработчиков Kotlin.

На основе полученных результатов можно сделать вывод о том, что предложенный подход позволяет успешно осуществлять поиск кодовых аномалий на основе списка токенов, сгенерированных по исходному коду программ на языке программирования Kotlin.

По результатам выполненной работы была подана статья на конференцию ESEC/FSE 2019 и сделан доклад на конференции СПИСОК-2019.

В качестве направлений дальнейших исследований в этой области можно выделить задачу поиска аномалий путем анализа различных структур представления, например, промежуточного представления программы для конкретного компилятора (IR), а также исследования кода на языке Javascript, генерируемого транслятором с языка Kotlin. А также применение других подходов к векторизации данных и методов обнаружения аномалий может дать новый результат в исследовании данной области.

[9] https://github.com/PrikhodkoStanislav/code-anomaly-detection

Список литературы

[1] Шкодырев ВП, Ягафаров КИ, Баштовенко ВА, Ильина ЕЭ. Обзор методов обнаружения аномалий в потоках данных // Second Conference on Software Engineering and Information Management (SEIM-2017)(full papers). — 2017. — P. 50.

[2] Richardson Bartley D, Radford Benjamin J, Davis Shawn E et al. Anomaly Detection in Cyber Network Data Using a Cyber Language Approach // arXiv preprint arXiv:1808.10742. — 2018.

[3] Stolfo Salvatore J, Hershkop Shlomo, Bui Linh H et al. Anomaly detection in computer security and an application to file system accesses // International Symposium on Methodologies for Intelligent Systems / Springer. — 2005. — P. 14–28.

[4] Kumar Vipin. Parallel and distributed computing for cybersecurity // IEEE Distributed Systems Online. — 2005. — no. 10. — P. 1.

[5] Spence Clay, Parra Lucas, Sajda Paul. Detection, synthesis and compression in mammographic image analysis with a hierarchical image probability model // Proceedings IEEE Workshop on Mathematical Methods in Biomedical Image Analysis (MMBIA 2001) / IEEE. — 2001. — P. 3–10.

[6] Baur Christoph, Wiestler Benedikt, Albarqouni Shadi, Navab Nassir. Deep autoencoding models for unsupervised anomaly segmentation in brain mr images // International MICCAI Brainlesion Workshop / Springer. — 2018. — P. 161–169.

[7] Aleskerov Emin, Freisleben Bernd, Rao Bharat. Cardwatch: A neural network based database mining system for credit card fraud detection // Proceedings of the IEEE/IAFE 1997 computational intelligence for financial engineering (CIFEr) / IEEE. — 1997. — P. 220–226.

[8] Ahmed Mohiuddin, Mahmood Abdun Naser, Islam Md. Rafiqul. A Survey of Anomaly Detection Techniques in Financial Domain // Future Gener. Comput. Syst. — 2016. — . — Vol. 55, no. C. — P. 278–288. — URL: https://doi.org/10.1016/j.future.2015.01.001.

[9] Fujimaki Ryohei, Yairi Takehisa, Machida Kazuo. An approach to spacecraft anomaly detection problem using kernel feature space // Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining / ACM. — 2005. — P. 401–410.

[10] Hangal Sudheendra, Lam Monica S. Tracking Down Software Bugs Using Automatic Anomaly Detection // Proceedings of the 24th International Conference on Software Engineering. — ICSE '02. — New York, NY, USA : ACM, 2002. — P. 291–301. — URL: http://doi.acm.org/10.1145/581339.581377.

[11] Feng Henry Hanping, Kolesnikov Oleg M, Fogla Prahlad et al. Anomaly detection using call stack information // Security and Privacy, 2003. Proceedings. 2003 Symposium on / IEEE. — 2003. — P. 62–75.

[12] Oizumi Willian N., Garcia Alessandro F., Colanzi Thelma E. et al. On the relationship of code-anomaly agglomerations and architectural problems // Journal of Software Engineering Research and Development. — 2015. — Jul. — Vol. 3, no. 1. — P. 11. — URL: https://doi.org/10.1186/s40411-015-0025-y.

[13] Taylor R. N., Osterweil L. J. Anomaly Detection in Concurrent Software by Static Data Flow Analysis // IEEE Transactions on Software Engineering. — 1980. — May. — Vol. SE-6, no. 3. — P. 265–278.

[14] Nguyen Tung Thanh, Nguyen Hoan Anh, Pham Nam H. et al. Graph-based Mining of Multiple Object Usage Patterns // Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. — ESEC/FSE '09. — New York, NY, USA :

ACM, 2009. — P. 383–392. — URL: http://doi.acm.org/10.1145/1595696.1595767.

[15] Wang Song, Liu Taiyue, Tan Lin. Automatically Learning Semantic Features for Defect Prediction // Proceedings of the 38th International Conference on Software Engineering. — ICSE '16. — New York, NY, USA : ACM, 2016. — P. 297–308. — URL: http://doi.acm.org/10.1145/2884781.2884804.

[16] Hangal Sudheendra, Lam Monica S. Tracking Down Software Bugs Using Automatic Anomaly Detection // Proceedings of the 24th International Conference on Software Engineering. — ICSE '02. — New York, NY, USA : ACM, 2002. — P. 291–301. — URL: http://doi.acm.org/10.1145/581339.581377.

[17] Bryksin Timofey, Petukhov Victor, Smirenko Kirill, Povarov Nikita. Detecting anomalies in Kotlin code // Companion Proceedings for the ISSTA/ECOOP 2018 Workshops / ACM. — 2018. — P. 10–12.

[18] Wang Song, Chollak Devin, Movshovitz-Attias Dana, Tan Lin. Bugram: Bug Detection with N-gram Language Models // Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. — ASE 2016. — New York, NY, USA : ACM, 2016. — P. 708–719. — URL: http://doi.acm.org/10.1145/2970276.2970341.

[19] Hsiao Chun-Hung, Cafarella Michael, Narayanasamy Satish. Using Web Corpus Statistics for Program Analysis // Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. — OOPSLA '14. — New York, NY, USA : ACM, 2014. — P. 49–65. — URL: http://doi.acm.org/10.1145/2660193.2660226.

[20] Nuez-Varela Alberto S., Prez-Gonzalez Hctor G., Martnez-Perez Francisco E., Soubervielle-Montalvo Carlos. Source Code

Metrics // J. Syst. Softw. — 2017. — . — Vol. 128, no. C. — P. 164–197. — URL: https://doi.org/10.1016/j.jss.2017.03.044.

[21] Caliskan-Islam Aylin, Harang Richard, Liu Andrew et al. De-anonymizing programmers via code stylometry // 24th {USENIX} Security Symposium ({USENIX} Security 15). — 2015. — P. 255–270.

[22] Jiang Lingxiao, Misherghi Ghassan, Su Zhendong, Glondu Stephane. Deckard: Scalable and accurate tree-based detection of code clones // Proceedings of the 29th international conference on Software Engineering / IEEE Computer Society. — 2007. — P. 96–105.

[23] Chilowicz Michel, Duris Etienne, Roussel Gilles. Syntax tree fingerprinting for source code similarity detection // 2009 IEEE 17th International Conference on Program Comprehension / IEEE. — 2009. — P. 243–247.

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

[25] Amer Mennatallah, Goldstein Markus, Abdennadher Slim. Enhancing One-class Support Vector Machines for Unsupervised Anomaly Detection // Proceedings of the ACM SIGKDD Workshop on Outlier Detection and Description. — ODD '13. — New York, NY, USA : ACM, 2013. — P. 8–15. — URL: http://doi.acm.org/10.1145/2500853.2500857.

[26] Çelik M., Dadaşer-Çelik F., Dokuz A. Ş. Anomaly detection in temperature data using DBSCAN algorithm // 2011 International Symposium on Innovations in Intelligent Systems and Applications. — 2011. — June. — P. 91–95.

[27] Campello Ricardo JGB, Moulavi Davoud, Sander Jörg. Density-based clustering based on hierarchical density estimates // Pacific-Asia

conference on knowledge discovery and data mining / Springer. — 2013. — P. 160–172.

[28] Münz Gerhard, Li Sa, Carle Georg. Traffic anomaly detection using k-means clustering // GI/ITG Workshop MMBnet. — 2007. — P. 13–14.

[29] Harada Yoshiyuki, Yamagata Yoriyuki, Mizuno Osamu, Choi Eun-Hye. Log-based anomaly detection of CPS using a statistical method // arXiv preprint arXiv:1701.03249. — 2017.

[30] Dau Hoang Anh, Ciesielski Vic, Song Andy. Anomaly detection using replicator neural networks trained on examples of one class // Asia-Pacific Conference on Simulated Evolution and Learning / Springer. — 2014. — P. 311–322.

[31] Prasad YA Siva, Krishna G Rama. Statistical Anomaly Detection Technique for Real Time Datasets // International Journal of Computer Trends and Technology (IJCTT)–volume. — 2013. — Vol. 6.

Приложение А

Пример аномалии класса “Много выражений expect”

```
@file:kotlin.jvm.JvmMultifileClass
@file:kotlin.jvm.JvmName("ArraysKt")

package kotlin.collections

import kotlin.comparisons.*

@kotlin.internal.InlineOnly
public expect inline operator fun <T> Array<out T>.component1(): T

@kotlin.internal.InlineOnly
public expect inline operator fun ByteArray.component1(): Byte

@kotlin.internal.InlineOnly
public expect inline operator fun ShortArray.component1(): Short

@kotlin.internal.InlineOnly
public expect inline operator fun IntArray.component1(): Int

@kotlin.internal.InlineOnly
public expect inline operator fun LongArray.component1(): Long

/* and 578 similar expressions */
```

Рисунок А1 — исходный код одной из найденных аномалий с оценкой 5

Приложение Б

Пример аномалии класса “Много типов аргументов”

```
@file:Suppress("UNUSED_PARAMETER")

class Unit internal constructor()

private val u = Unit()

operator fun <P1, P2, P3, P4, P5, /* ... */,  P20, R> ((
    P1, P2, P3, P4, P5, /* ... */, P20) -> R).invoke(
    p1: P1, `2`: Unit = u, `3`: Unit = u, `4`: Unit = u, `5`: Unit = u, /* ... */, `20`: Unit = u):
    (P2, P3, P4, P5, /* */, P20) -> R =
    { p2: P2, p3: P3, p4: P4, p5: P5, /* ... */, p20: P20 -> this(p1, p2, p3, p4, p5, /* */, p20) }

operator fun <P1, P2, P3, P4, P5, /* ... */, P20, R> ((
    P1, P2, P3, P4, P5, /* ... */, P20) -> R).invoke(
    `1`: Unit = u, p2: P2, `3`: Unit = u, `4`: Unit = u, `5`: Unit = u, /* ... */, `20`: Unit = u):
    (P1, P3, P4, P5, /* ... */, P20) -> R =
    { p1: P1, p3: P3, p4: P4, p5: P5, /* ... */, p20: P20 -> this(p1, p2, p3, p4, p5, /* ... */, p20) }

operator fun <P1, P2, P3, P4, P5, /* ... */, P20, R> ((
    P1, P2, P3, P4, P5, /* ... */, P20) -> R).invoke(
    `1`: Unit = u, `2`: Unit = u, p3: P3, `4`: Unit = u, `5`: Unit = u, /* ... */, `20`: Unit = u):
    (P1, P2, P4, P5, /* ... */, P20) -> R =
    { p1: P1, p2: P2, p4: P4, p5: P5, /* ... */, p20: P20 -> this(p1, p2, p3, p4, p5, /* ... */, p20) }

/* and 17 similar expressions */
```

Рисунок Б1 — исходный код одной из найденных аномалий с оценкой 5