View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Electronic Archive of Kyiv Polytechnic Institute

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ І
СПЕЦІАЛІЗОВАНИХ КОМП'ЮТЕРНИХ СИСТЕМ

«На правах рукопису»
УДК 004.9

«До захисту допущено»
Завідувач кафедри СПСКС

Віталій РОМАНКЕВИЧ

____________2020 р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 123 Комп'ютерна інженерія
(Комп'ютерні системи та компоненти)

на тему: Спосіб виявлення неоптимального використання програмного коду мовою Kotlin ____

____

Виконала: студентка ІІ курсу, групи _КВ-91_мп
Мурдза Оксана Олегівна ______________________________ __________

Науковий керівник Олександр ЩЕРБИНА, доцент кафедри СПСКС
к.т.н., доцент ______________________________ __________

Консультант з нормоконтролю доцент, с.н.с.,к.т.н. Юлія БОЯРІНОВА ______ __________

Рецензент доцент, к.т.н., доцент Марія ОРЛОВА ______________________ __________

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____________

Київ – 2020 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп’ютерних систем

Рівень вищої освіти – другий (магістерський)
за освітньо-професійною програмою
Спеціальність 123 Комп‘ютерна інженерія
Комп‘ютерні системи та компоненти

ЗАТВЕРДЖУЮ
Завідувач кафедри СПСКС
__________ _Віталій РОМАНКЕВИЧ__

«__01_»______12_______2019__р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Мурдза Оксана Олегівна ____________________________________
(прізвище, ім’я, по батькові)

1. Тема дисертації Спосіб виявлення неоптимального використання програмного коду мовою Kotlin ______________________________
____,
науковий керівник дисертації Олександр ШЕРБИНА, доцент кафедри СПСКС к.т.н. доцент____,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «_12__» __11__ 20___ р. № 3298-С
2. Термін подання студентом дисертації 10 грудня 2020 р. ______________
3. Об’єкт дослідження способи пошуку неоптимального коду написаних мовою Kotlin ____
____
4. Предмет дослідження є розробка системи пошуку неоптимального використання програмного коду мовою Kotlin на основі 4 алгоритмів пошуку аномалій__________________________________
____

5. Перелік завдань, які потрібно розробити: аналіз існуючих методів та систем пошуку неоптимального коду виділення основних алгоритмів пошуку аномалій, програмна реалізація системи пошуку неоптимального коду ______

____

____

6. Перелік ілюстративного матеріалу ________________________________

____

____

____

____

7. Перелік публікацій за тематикою проведених досліджень опубліковано 2 наукові праці, а саме тези доповідей на конференціях. ____________________

____

____

8. Дата видачі завдання 5 вересня 2019 р. ______________________________

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Формування мети та цілі роботи	01.11.2019	
2	Дослідження теоретичного матеріалу	01.03.2020	
3	Дослідження існуючих алгоритмів	01.05.2020	
4	Розробка нової системи	01.07.2020	
5	Реалізація нової системи	01.09.2020	
6	Проведення аналізу роботи системи	01.10.2020	
7	Написання дисертації	20.11.2020	
8	Попередній розгляд магістерської дисертації на кафедрі	26.11.2020	

Студент ____________ Оксана МУРДЗА

Науковий керівник дисертації ____________ Олександр ЩЕРБИНА

РЕФЕРАТ

Актуальність теми. На сьогодні активно розвивається область розробки програмного забезпечення. Кожен день створюється величезна кількість рядків програмного коду. Найчастіше при розгляді вихідного коду програм можна виявити окремі частини коду, що за тією чи іншою ознакою виділяються на загальному тлі розглянутого набору даних. Такі приклади нетипового коду в рамках даної роботи називаються кодовими аномаліями або неоптимальним використанням коду.

Kotlin - це досить молода мова програмування зі швидко зростаючою спільнотою користувачів і значною екосистемою різноманітних проектів з відкритим вихідним кодом.

Розробники вже не перший рік ведуть дослідження з пошуку кодових аномалій в програмах мовою Kotlin. У зв'язку з актуальністю вирішення такого завдання в роботі пропонується розширити існуючі рішення і створити систему виявлення нових фрагментів неоптимального коду.

Мета роботи: підвищення ефективності системи виявлення кодових аномалій за рахунок розширення переліку функцій та виявлення нових класів прикладів коду, що виділяються своїм нестандартним змістом в програмах мовою Kotlin.

Об'єктом дослідження є методи векторизації, токенізації для розбиття коду та збору статистичного аналізу неоптимального використання програмного коду (аномалій).

Предметом дослідження є системи пошуку неоптимального використання програмного коду мовою Kotlin на основі токенів.

*Методи дослідження***.** В роботі використовуються методи токенізації, методи верктоgive ризації та кластеризації даних.

Наукова новизна одержаних результатів полягає в наступному.

1. Проаналізовано основні методики пошуку неоптимального використання коду та показано, шо на сьогодні залишається недостатньо дослідженим питання пошук неоптимального коду написаних мовою програмування Kotlin.
2. Запропоновано спосіб пошуку неоптимального коду, який відрізняється від існуючих методами векторизації та пошуку аномалій, та дозволяє знайти більше неоптимального коду
3. Проведено апробацію і отримано набір кодових аномалій, які зібрані у класи та порівняно з існуючими рішеннями.
4. Отримано експертну оцінку користі знайдених аномалій.

Практична цінність одержаних результатів полягає в тому, що запропонований спосіб підвищує ефективність пошуку неоптимального використання коду, що дозволяє отримати більш якісний код за такими параметрами як швидкодія, пам'ять та поломки програми.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювались на:

- XIII науковій конференції молодих вчених «Прикладна математика та комп'ютинг» ПМК-2020;
- VI міжнародна науково-технічна Internet-конференція

Публікації. За тематикою проведених досліджень опубліковано 2 наукові праці, а саме 2 тези доповідей на конференціях.

Структура та обсяг роботи. Магістерська дисертація складається з вступу, трьох розділів, висновків та додатків.

У вступі надано загальну характеристику програмного коду, проблематику пошуку неоптимального використання програмного коду сформульовано мету дослідження, показано практичну цінність роботи.

У першому розділі надано детальне обґрунтування актуальності напрямку досліджень, виконано оцінку поточного стану в даній сфері, представлено теоретичний огляд особливостей пошуку аномалій.

У другому розділі розроблено та описано систему пошуку неоптимального використання програмного коду мовою Kotlin.

У третьому розділі проведено апробацію.

У висновках проаналізовано отримані результати роботи.

Ключові слова: аномалії, векторизація, токенизація, кластеризація, токен, лексема, TF-IDF, Bag-of-words.

ABSTRACT

Actuality of theme. Today the field of software development is actively developing. Every day a huge number of lines of program code are created. Most often, when considering the source code of programs, you can find individual parts of the code, which in one way or another stand out against the general background of the data set. Such examples of atypical code in this work are called code anomalies or suboptimal use of code.

Kotlin is a fairly young programming language with a fast-growing user community and a large ecosystem of various open source projects.

The developers have been conducting research on the search for code anomalies in Kotlin programs for more than a year. Due to the urgency of solving this problem, the paper proposes to expand existing solutions and create a system for detecting new fragments of suboptimal code.

The purpose of the work is improving the efficiency of the code anomaly detection system by expanding the list of functions and identifying new classes of code examples that stand out for their non-standard content in Kotlin programs.

The object of the study is the methods of vectorization, tokenization for code splitting and collection of statistical analysis of suboptimal use of program code (anomalies).

The subject of the research is systems for searching for suboptimal use of program code in Kotlin language based on tokens.

Research methods. The paper uses methods of tokenization, methods of vectorization and clustering of data.

The scientific novelty of the obtained results is as follows.

1. The main methods of searching for suboptimal use of code are analyzed and it is shown that today the question of searching for suboptimal code written in Kotlin programming language remains insufficiently researched.

2. A method for finding a suboptimal code is proposed, which differs from the existing methods of vectorization and anomaly search, and allows to find more suboptimal code

3. Approbation was carried out and a set of code anomalies was obtained, which were collected in classes and compared with existing solutions.

4. An expert assessment of the benefits of the found anomalies was obtained.

The practical novelty of the obtained results is that the proposed method increases the efficiency of search for suboptimal use of code, which allows to obtain better quality code in such parameters as speed, memory and program crashes.

Approbation of work. The main provisions and results of the work were presented and discussed at:

• XIII scientific conference of young scientists "Applied Mathematics and Computing" PMK-2020;

• VI International Scientific and Technical Internet Conference

Publications. Two scientific papers were published on the subject of the conducted researches, namely 2 abstracts of reports at conferences.

Structure and scope of work. The master's dissertation consists of an introduction, three sections, conclusions and appendices.

The introduction provides a general description of the program code, the problem of finding suboptimal use of program code, formulates the purpose of the study, shows the practical value of the work.

In the first section the detailed substantiation of urgency of a direction of researches is given, the estimation of a current condition in the given sphere is executed, the theoretical review of features of search of anomalies is presented.

The second section develops and describes a system for searching for suboptimal use of program code in the Kotlin language.

In the third section the approbation is carried out.

The results of the work are analyzed in the conclusions.

Keywords: anomalies, vectorization, tokenization, clustering, token, token, TF-IDF, Bag-of-words.

ЗМІСТ

ПЕРЕЛІК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ 12
ВСТУП 14
1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ПОШУКУ КОДОВИХ АНОМАЛІЙ В ПРОГРАМАХ ТА ОБГРУНТУВАННЯ ТЕМИ МАГІСТЕРСЬКОЇ ДИСЕРТАЦІЇ 16
1.1 Виявлення аномалій в різних завданнях 16
1.2 Система пошуку кодових неоптимального коду для мови Kotlin 18
1.3. Компілятор мови програмування Kotlin 21
1.4. Методи векторизації даних 23
1.5. Алгоритми виявлення аномалій 28
ВИСНОВКИ ДО РОЗДІЛУ 1 42
2. РЕАЛІЗАЦІЯ СПОСОБУ ПОШУКУ НЕОПТИМАЛЬНОГО КОДУ МОВОЮ KOTLIN ЗА ДОПОМОГОЮ СТАТИСТИЧНОГО АНАЛІЗУ 43
2.1 Токенізація програми 46
2.2. Векторизація даних 49
2.2.1. Векторизація списку типів токенів 50
2.2.2. Векторизація списку лексем 55
2.3. Виявлення неоптимального коду 58
2.4. Кластеризація неоптимального коду 64
ВИСНОВКИ ДО РОЗДІЛУ 2 68
3. АПРОБАЦІЯ РЕАЛІЗОВАНОЇ СИСТЕМИ 69
3.1 Збір даних 69
3.2 Векторизація получених даних 69
3.3 Запуск алгоритмів пошуку неоптимального коду 70
3.4 Кластеризація найденого неоптимального коду 71
3.5 Складання звіту про найдений неоптимальний код 71
3.6 Отримання експертних оцінок 76
3.7. Порівняння результатів зі схожими роботами 78

ВИСНОВКИ ДО РОЗДІЛУ 3 82
ЗАГАЛЬНІ ВИСНОВКИ ПО РОБОТІ. 83
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 85
ДОДАТКИ 91
ДОДАТОК А Приклад неоптимального коду класа “Багато виразів expect” 91
ДОДАТОК Б Приклад неоптимального коду класа “Багато типів аргументів” 91
ДОДАТОК В. Презентація. 91
ДОДАТОК Г. Публікації за темою роботи. 91
ДОДАТОК Д. Лістинг програми. 91
ДОДАТОК Е. Довідка про впровадження. 91

ПЕРЕЛІК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

ПЗ — програмне забезпечення.

API — це сукупність засобів та правил, що вможливлюють взаємодію між окремими складниками програмного забезпечення або між програмним та апаратним забезпечення.

NLP (Natural Language Processing) — обробка природної мови - підрозділ інформатики.

AI — Штучний інтелект.

Токен - об'єкт, що створює з лексеми в процесі лексичного аналізу.

Токенізація — це процес поділу мови на компоненти.

Векторизація — перетворення в числові вектора.

Кластеризація (Кластерний аналіз) — розбиття заданої вибірки об'єктів на підмножини.

Kotlin — мова програмування.

JVM (Java Virtual Machine) — віртуальна машина Java.

JetBrains — компанія з розробки ПЗ.

Code Samples (Patterns) –шаблон, патерн, зразок коду.

Java — мова програмування.

GrouMiner — система патернів коду програм написаних на мовах сімейства JVM.

Javascript — мова програмування.

DIDUCE — система пошуку аномалій (anomaly detection system).

GitHub — веб-сервісів для спільної розробки ПЗ.

PSI (Program Structure Interface) — інтерфейс структури програми

TF-IDF — ститистичний показник

SVM (Support Vector Machine) — система підтримки машини векторизації

HDBSCAN — метод пошуку аномалій

DBSCAN — метод пошуку аномалій

K-means — метод кластиризації

LOF (local outlier factor) — метод пошуку аномалій

Scikit-learn — бібліотека Python

Bag-of-words — метод векторизації

IEEE — група стандартнів

DBN (Deep Belief Network) — глибока нейромережа довіри

ВСТУП

В даний час активно розвивається область розробки програмного забезпечення. Кожен день створюється величезна кількість рядків програмного коду. Мільйони людей по всьому світу займаються написанням коду, його тестуванням, проводять рев'ю і налагодження. Однак результатом діяльності програмістів є програмний продукт, до якого, як правило, пред'являється досить багато вимог, які дозволяють судити про його якість і працездатності. У зв'язку з цим необхідно бути впевненим, що на всіх етапах, на шляху від вихідного коду до програмного продукту в ньому немає дефектів. Одним з найважливіших завдань є тестування компілятором програми на предмет наявності помилок, які можуть негативно відобразитися на роботі програмного продукту.

Природа їх виникнення може бути абсолютно різною. Можливо програма містить проблему, яка криється в логіці розробленого коду, і тоді тягар відповідальності за можливі неполадки лягає на програміста, який написав цей код. Однак можливі випадки, коли код не містить помилок і повністю коректний, але в силу своєї специфіки потрапляє під визначення неоптимального коду (кодові аномалії). Аномалії в своїй суті показують, як не прийнято більшістю розробників на тій чи іншій мові програмування писати програмний код. У зв'язку з цим виявлення неоптимального коду може допомогти розробникам мови програмування в рішенні цілого ряду завдань. Наприклад, неоптимальний код може допомогти виявити недоліки в дизайні мови програмування, можуть вказувати на проблеми продуктивності програм, оптимізацій компілятора, виведення типів, генерації коду, аналізу потоку даних.

Kotlin [1] - це досить молода і одина з найбільш активно розвиваючихся мов програмування с швидко зростаючою спільнотою користувачів і значною

екосистемою різноманітних проектів з відкритим вихідним кодом. Kotlin є високорівнева статично типізована мова програмування, що використовується для розробки на платформі JVM (Java Virtual Machine), також компілюється в мови програмування Javascript, Java і в код інших платформ. Мова програмування розроблений компанією JetBrains [2]. Розробники мови Kotlin регулярно вдосконалюють екосистему мови, в тому числі компілятор. Їх зацікавленість у виявленні неоптимального коду в програмах на мові програмування Kotlin полягає в тому, що кодові аномалії дозволять звернути увагу на нестандартні підходи до застосування мовних конструкцій, дадуть інформацію про те, як варто вдосконалити компілятор, а також неоптимальний код можуть стати тестовими прикладами в процесі тестування компілятора.

У зв'язку з цим завдання виявлення аномалій в програмах на мові Kotlin досить актуальна і важлива як для користувачів, так і для розробників мови програмування. У лабораторії JetBrains Research [3] вже не перший рік ведуться дослідження на тему пошуку неоптимального коду в програмах на мові Kotlin. Вже отримані перші результати з прикладами неоптимального коду, частина з яких вже включена в тести для компілятора мови Kotlin. У зв'язку з актуальністю вирішення такого завдання в цій роботі пропонується провести дослідження, спрямоване на розширення існуючого рішення і створення інструменту для виявлення нових специфічних прикладів неоптимального коду.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ПОШУКУ КОДОВИХ АНОМАЛІЙ В ПРОГРАМАХ ТА ОБГРУНТУВАННЯ ТЕМИ МАГІСТЕРСЬКОЇ ДИСЕРТАЦІЇ

1.1 Виявлення аномалій в різних завданнях

Виявлення аномалій відноситься до задачі пошуку примірників даних, які не відповідають очікуваному уявленню. Також аномалією називають відхилення поведінки системи від очікуваного стану [1]. Завдання виявлення аномалій поставлена в різних областях знань для вирішення широкого кола проблем. Наприклад, вельми актуальна задача виявлення поширення кібератак в мережі [2, 3, 4]. Аномальне значення трафіку в комп'ютерній мережі може означати, що зламаний комп'ютер відправляє конфіденційні дані в неавторизований пункт призначення. Також не менш значущими сферами застосування методів виявлення аномалій є медицина, де відбувається визначення патологій за медичними даними [5, 6], а також фінансова сфера, в якій відбувається визначення підозрілих активностей, фіксація розкрадань персональних даних і грошових коштів [7, 8]. В космічній сфері аномальні показники датчиків космічних кораблів можуть свідчити про несправності будь-якого компонента космічного корабля [9].

У програмній інженерії завдання виявлення аномалій (неоптимального коду) ставиться з метою пошуку помилок [10], виявлення вторгнень [11], пошуку архітектурних недоліків [12], помилок синхронізації в паралельних програмах [13] і т.д. Розробники різних інструментів, спрямованих на пошук аномалій в програмах, вводять свої власні визначення неоптимального коду, тому їх цілі, методи і результати також різняться.

Система GrouMiner [14] призначена для виявлення аномальних взаємодій об'єктів у програмах на мові програмування Java. Підхід, описаний в даній роботі, полягає в моделюванні взаємодії об'єктів шляхом побудови

орієнтованого ациклического графа, вузлами якого є виклики методів і звернення до полів об'єктів, а ребра представляють залежності між ними. В рамках цієї роботи використовується статичний аналіз коду. Побудова графа взаємодії об'єктів відбувається на основі абстрактного синтаксичного дерева програми. Методи виявлення неоптимального коду дозволяють виявити нестандартні виклики методів і нетипові області графа потоку керування.

В роботі [15] описується система прогнозування дефектів в програмному забезпеченні на мові Java. Дефекти передбачаються для двох різних ситуацій: в ситуації, коли дефект міститься всередині одне проекту, і в ситуації міжпроектної дефектів. В обох випадках процес виявлення дефектів складається з декількох етапів: етапу векторизації вихідних даних, навчання моделі класифікації і передбачення аномальність поступаємих об'єктів даних. Даний підхід має на увазі статичний аналіз коду, при якому програма представлена у вигляді абстрактного синтаксичного дерева. Векторизація здійснюється на основі інформації про вершини дерева. У даній роботі витяг семантичних ознак на етапі векторизації здійснюється автоматично з використанням алгоритму DBN (Deep Belief Network), якому і присвячена більша частина роботи. Як алгоритмів класифікації в роботі використовуються методи DTree, наївний байесовский класифікатор, і логістична регресія.

Зовсім інший підхід застосований групою дослідників при реалізації системи DIDUCE [16], де для пошуку неоптимального коду в програмах на мові Java використовується метод динамічного аналізу коду. При запуску програми зберігаються значення всіх виразів програми. Система генерує правила для цих виразів, починаючи від самих строгих, і послаблює їх у міру появи нових значень виразу. Як тільки правила порушуються, це означає, що значення виразу істотно відрізняється від усіх попередніх, і таке значення виразу оголошується аномальним.

Варто відзначити, що методи динамічного аналізу коду спрямовані в першу чергу для виявлення логічних або архітектурних проблем в програмах, тому вони орієнтовані більшою мірою на користувачів мови програмування. Розробникам мови программірованія скоріше більш цікаві методи статичного аналізу програмного коду, тому що вони спрямовані на дослідження коду без прив'язки до середовища виконання програми.

В рамках поставленої задачі пропонується проводити статичний аналіз програмного коду. Статичний аналіз коду полягає в аналізі програмного забезпечення без виконання запуску досліджуваних програм в виконуваної середовищі. Даний підхід є одним із способів аналізу коду і широко застосовується в різних дослідженнях. Одним з переваг такого підходу є відсутність необхідності в підготовці тестового оточення запуску програми на відміну від більш складно організованого динамічного аналізу програми. До того ж статичний аналіз не залежить від сторонніх компонентів інфраструктури та середовища виконання програми, що дозволяє ізольовано розглядати програмний код без прив'язки до його оточенню, а також в перспективі дозволяє знаходити помилки, які можуть себе проявити лише через тривалий термін. Статичний аналіз коду дозволяє виявити помилки і дефекти на ранніх стадіях розробки програмного забезпечення, а саме написання коду і його компіляції, що істотно знижує вартість усунення проблем в програмі і попереджає збитки різного характеру.

1.2 Система пошуку кодових неоптимального коду для мови Kotlin

Завдання виявлення неоптимального коду в програмах на мові Kotlin вже вирішувалася раніше групою дослідників команди JetBrains Research [17]. Була створена система пошуку кодових аномалій, яка дозволяє знаходити

нестандартні приклади програм на мові Kotlin. Архітектура системи зображена на рисунку 1.

Рисунок 1 - Архітектура системи

Процес виявлення неоптимального коду складається з декількох етапів:

• збір набору даних з вихідним кодом програм і збірок проектів з сервісу GitHub з використанням наданого API;

- перетворення зібраних даних в дерева розбору і списки інструкцій JVM;
- витяг числових векторів з отриманих структур;
- застосування методів виявлення неоптимального коду до витягнутим векторах ознак;
- постобработка даних, яка полягає в класифікації виявлених неоптимального коду відповідно до їх типом.

Завдання системи полягає в синтаксичному аналізі програмного коду на мові Kotlin. Це рішення спрямоване на аналіз таких структур представлення програми, як дерева розбору і списку інструкцій JVM.

Дерево розбору є однією з основних структур, що описують внутрішнє представлення програми, що будується на етапі синтаксичного аналізу в процесі роботи компілятора. Інструкції JVM є структурою, одержуваної в результаті роботи компілятора мови Kotlin. Такий вибір обумовлений метою авторів рішення провести пошук нетипових прикладів програмного коду, що виділяються своєю нестандартною структурою.

В системі розглянуто два підходи до векторизації дерева розбору: явним витягом ознак та автокодуванням. Обидва підходи мають ряд переваг і недоліків, проте в сукупності вони дозволяють зробити широке охоплення різних видів неоптимального коду.

В якості алгоритмів виявлення неоптимального коду в системі використовуються локальний рівень викиду, ізолюючий ліс, метод еліпсоїдальної апроксимації даних, нейронний автоенкодер. Всі перераховані алгоритми відносяться до класу алгоритмів навчання без вчителя. Робота перших трьох алгоритмів заснована на кластеризації даних. Сутність автоенкодера полягає в стисненні даних з втратами і їх подальшим відновленням.

Всі знайдені аномалії поділяються на класи схожих за змістом прикладів коду. Класифікація неоптимального коду в системі відбувається вручну шляхом візуального огляду виявлених кодових аномалій. В результаті роботи системи формується набір прикладів коду на мові програмування Kotlin, які оголошені аномальними. Також система має зручний механізмом візуалізації знайдених аномалій, яка дозволяє барвисто уявити результат і отримати зворотній зв'язок від експертів.

Дана система послужила основою для розроблюваного в рамках даної роботи рішення. Для розширення системи виявлення неоптимального коду з метою отримання нових видів нестандартних прикладів коду на мові Kotlin прийнято рішення реалізувати нові підходи до структури представлення програм, векторизації даних і нові методи виявлення неоптимального коду.

1.3. Компілятор мови програмування Kotlin

Основним застосуванням мови програмування Kotlin є розробка додатків для платформи JVM. Kotlin є компільовані в байт-код JVM - список інструкцій віртуальної машини мови програмування Java. На рисунку 2 представлено процус компіляції.

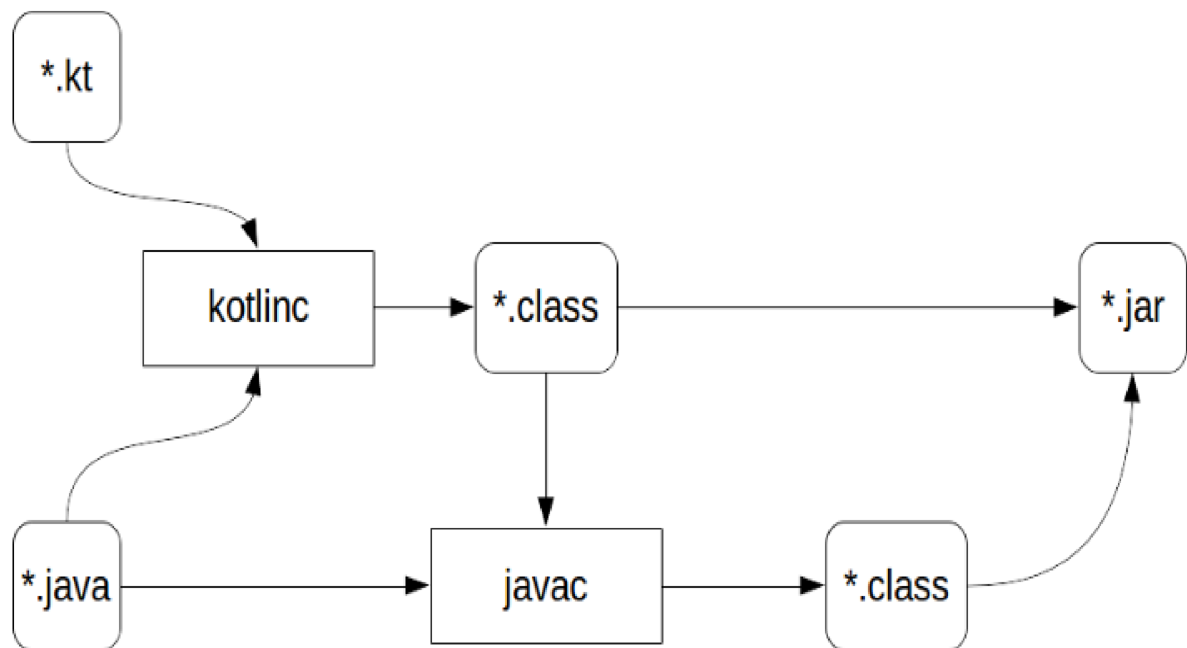


Рисунок 2 – Процес компіляії програми

На вхід компілятора kotlinc надходять вихідні файл, це зображено на рисунку 2, причому не тільки файли kotlin, але і файли java. Це потрібно щоб

можна було вільно посилатися на Java з Kotlin, і навпаки. Сам компілятор прекрасно розуміє вихідні Java, але не займається їх компіляцією, на цьому етапі відбувається тільки компіляція файлів Kotlin. Після отримані * .class файли передаються компілятору javac разом з вихідними файлами * .java. На цьому етапі компілюються всі java файли, після чого стає можливим зібрати разом всі файли в jar (або яким іншим чином).

Компілятор мови Kotlin складається з наступних компонентів:

- лексичний аналізатор, що перетворює послідовність символів вихідного коду програми в послідовність токенів;
- синтаксичний аналізатор, що перетворює набір токенів в дерево розбору, яке в компіляторі мови Kotlin іменується PSI (Program Structure Interface);
- генератор коду інструкцій JVM на основі дерева розбору PSI.

SDK IntelliJ Platform пропонує зручний інструментарій для синтаксичного аналізу програмного коду на мовах, що використовуються в середовищі розробки в рамках цієї платформи - в тому числі на Kotlin.

Компілятор мови дозволяє отримати конкретне синтаксичне дерево коду - інтерфейс структури програм (PSI) [15]. Кожному типу вузла конкретного синтаксичного дерева відповідає визначеному класу, реалізує інтерфейс PsiElement. Підтримується обхід PSI із застосуванням шаблону проектування "відвідувач": для цього потрібно вивчити клас PsiElementVisitor, що надається бібліотекою компілятора Kotlin.

Серед інструментів, що використовують PSI, найбільш релевантним є MetricsReloaded4 - плагін для збору IntelliJ IDEA для підключення синтаксичних метриків Java-кода. Доступно кілька десятків метриків, вичитаних на рівні проекту, файлів, пакетів Java, класу чи методу. У їх кількості такі метрики, як число строкових кодів із коментарями або без, цикломатична та проектувальна сложність, метрики Холстеда, кількість окремих синтаксичних конструкцій.

MetricsReloaded - проект з відкритим вихідним кодом, тобто доступний як список метриків, так і реалізація. Наступно, можна порівняно легко адаптувати Java-метрики для мови Kotlin, при передачі яких використовуються інші елементи PSI.

Важливо зауважити, що до етапу генерації байт-коду JVM компілятор здатний працювати навіть з синтаксично і семантично некоректними конструкціями. У зв'язку з цим варто зазначити, що виявлення помилок на більш ранніх стадіях компіляції програми дозволить скоротити як тимчасові, так і багато інших ресурсомісткі витрати усунення дефектів.

В рамках даної роботи пропонується досліджувати етап лексичного аналізу програми, а саме представлення програми у вигляді списку токенів. Токен є структурою, що включає в себе тип токена і послідовність символів лексеми, який виділяється з вихідного коду програми. У зв'язку з цим токен містить інформацію про вихідний поданні смислової одиниці мови програмування (лексеми) в програмі і про його ідентифікатор, який дозволяє дізнатися, яку смислове навантаження даний токен несе. В рамках поставленої задачі структура списку токенів містить всю необхідну інформацію про подання програми для аналізу його змісту і виявлення нестандартних з точки зору змісту вихідного коду прикладів програм.

1.4. Методи векторизації даних

Алгоритми пошуку неоптимального коду, як і більшість методів машинного навчання, беруть в якості вхідних даних набори числових векторів. У зв'язку з цим необхідно позначити ряд методів, які здатні перетворити дані в вектори числових ознак.

Natural Language Processing (далі - NLP) - обробка природної мови - підрозділ інформатики і AI, присвячений тому, як комп'ютери аналізують

природні мови. NLP дозволяє застосовувати алгоритми машинного навчання для тексту й мови.

Наприклад, ми можемо використовувати NLP, щоб створювати системи на кшталт розпізнавання мови, узагальнення документів, машинного перекладу, виявлення спаму, розпізнавання іменованих сутностей, відповідей на питання, автокомпліта, інтелектуального введення тексту і т.д.

Токенізація (іноді – сегментація) - це процес поділу писемної мови на пропозиції-компоненти. Ідея виглядає досить просто. В англійській і деяких інших мовах ми можемо виокремлювати пропозицію кожен раз, коли знаходимо певний знак пунктуації - точку.

Але навіть в англійському ця задача нетривіальна, так як точка використовується і в скороченнях. Таблиця скорочень може сильно допомогти під час обробки тексту, щоб уникнути невірної розстановки кордонів пропозицій. У більшості випадків для цього використовуються бібліотеки, так що можете особливо не переживати про деталі реалізації.

Отже, як тільки текст перетворився в очищену нормалізовану послідовність слів, запускається процес їх векторизації - перетворення в числові вектора. Для такої трансформації використовуються спеціальні моделі.

У разі розгляду коду в текстовому вигляді або у вигляді списку токенів для векторизації застосовні методи обробки природної мови (NLP), такі як мішок слів (bag-of-words) і мішок n-грам (Bag of n-grams) [18, 19]. Однак існують і інші техніки векторизації, такі як застосування функцій хешування і підрахунку метрик TF-IDF. При поданні коду у вигляді дерева розбору існує велика кількість способів явного і неявного вилучення ознак.

Розглянемо детальніше основні 4 методи векторизації даних для NLP.

Пряме кодування (one-hot encoding) вважається найпростішим способом перетворення токенів в тензори і виконується наступним чином:

- кожен токен являє бінарні вектор (значення 0 або 1);
- одиниця ставиться тому елементу, Який відповідає номеру токена в словнику.

Виглядає так:

{Пес, Кот, ель, на, сів, пень} # Словник
Перший документ
[[1, 0, 0, 0, 0, 0] # Пес
[0, 0, 0, 0, 0, 0] # Кот (мається на увазі)
[0, 0, 0, 0, 0, 0] # ель (мається на увазі)
[0, 0, 0, 1, 0, 0] # на
[0, 0, 0, 0, 1, 0] # сел
[0, 0, 0, 0, 0, 1]] # пень
Другий документ
[[0, 0, 0, 0, 0, 0] # Пес (мається на увазі)
[0, 1, 0, 0, 0, 0] # Кіт
[0, 0, 1, 0, 0, 0] # ель
[0, 0, 0, 1, 0, 0] # на
[0, 0, 0, 0, 1, 0] # сел
[0, 0, 0, 0, 0, 0]] # пень (мається на увазі)

Проблемою прямого кодування є розмірність. Кожна пропозиція складається всього з 4 слів, але в підсумки вийшла велика матриця для кожного документа. Кількість рядків регулюється словником, тому чим більше слів у словнику, тим більше буде матриця.

BAG OF WORDS на відміну від прямого кодування, мішок слів (Bag of words) виділяє вектору весь документ, і кожен елемент кодується 1 по порядку розташування слів в словнику:

{Пес, Кіт, ялина, на, сів, пень} # Словник

Корпус:

[[1, 0, 0, 1, 1, 1] # Перший документ

[0, 1, 1, 1, 1, 0]] # Другий документ

Bag of words вирішує проблему розмірності по одній осі. Кількість рядків визначається кількістю документів. Однак, цей метод не враховує важливість того чи іншого токена, адже одне слово може повторяться по кілька разів. В цьому випадку стане в нагоді альтернативний спосіб, розглянутий далі.

TF-IDF складається з двох компонентів: Термінова частота (частота слов у документах) та обернена частота документа (інверсія частоти документа). Вони вважаються наступним образом:

$$TF_{token_i} = \frac{n_i}{N_i},$$

$$IDF_{token} = \log \frac{p}{P},$$

де n_i - скільки раз зустрічається токен в і-тому документі,

N_i - загальна кількість токенів у і-тому документі,

p - кількість документів, у яких зустрічається токен,

P - загальна кількість документів.

У кінцевому навчанні, TF-IDF - це похідна TF на IDF

$$TF\text{-}IDF = TF \times IDF$$

Підхід явного вилучення ознак [20, 21] хороший тим, що зазвичай отримані ознаки легко інтерпретувати і зрозуміти. Неявні ознаки [22, 23] більш важкі для розуміння, однак такий підхід в окремих випадках здатний визначати складні властивості коду, такі як, наприклад, семантичні залежності.

У результаті отримали для 1-го документа такі важливі слова, як “Пес” та “пень”, для 2-го - “Кіт” та “ель” продемонстровано на таблиці 1.

Таблиця 1 – Аналіз TF-IDF

Токен	TF		IDF	TF-IDF	
	Док. №1	Док. №2		Док. №1	Док. №2
Кот	0	$\frac{1}{4}$	$\log\frac{2}{1}$	0	0.17
Пес	$\frac{1}{4}$	0	$\log\frac{2}{1}$	0.17	0
ель	0	$\frac{1}{4}$	$\log\frac{2}{1}$	0	0.17
на	$\frac{1}{4}$	$\frac{1}{4}$	$\log\frac{2}{2}=0$	0	0
сел	$\frac{1}{4}$	$\frac{1}{4}$	$\log\frac{2}{2}=0$	0	0
пень	$\frac{1}{4}$	0	$\log\frac{2}{1}$	0.17	0

У TF-IDF деякі слова та слова, які зустрічаються у всіх документах, відсутні мало інформації. Крім того, IDF можна розглядати та інші способи, наприклад, у бібліотеці Python Scikit-learn цей параметр гнучко регулюється.

У зв'язку з розглядом подання програми у вигляді структури списку токенів в рамках даної роботи прийнято рішення використовувати підхід до векторизації даних, що використовує методи обробки природної мови (Natural Language Processing) в варіації їх застосування до структури токенів. Такий підхід, як правило, дозволяє досліджувати семантику імен функцій і змінних без урахування структури програми.

1.5. Алгоритми виявлення аномалій

Для аналізу побудованих векторів і виділення аномальних даних необхідно застосувати методи виявлення неоптимального коду. В області машинного навчання під аномалією розуміють відхилення поведінки системи від очікуваного.

Аномалії можна розділити на 3 види [24]:

- точкові аномалії відповідають випадкам, коли окремий екземпляр даних є аномальним по відношенню до решти даними (на рисунку 3 точки A1 і A2, а також область точок A3 є аномальними по відношенню до областей R1 і R2);

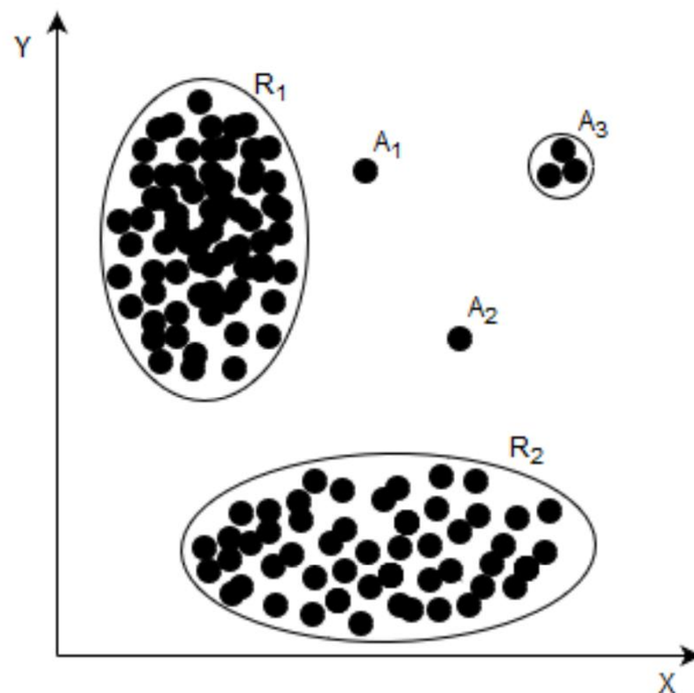


Рисунок 3 - Приклад точкових аномалій

- контекстуальні аномалії відповідають випадкам, коли окремий екземпляр даних є аномальним тільки в певному контексті; аномальна поведінка визначається за допомогою значень контекстних атрибутів, наприклад, часом спостереження (на рисунку 4 в точці А простежується аномалія в контексті з точками N1 - N5);

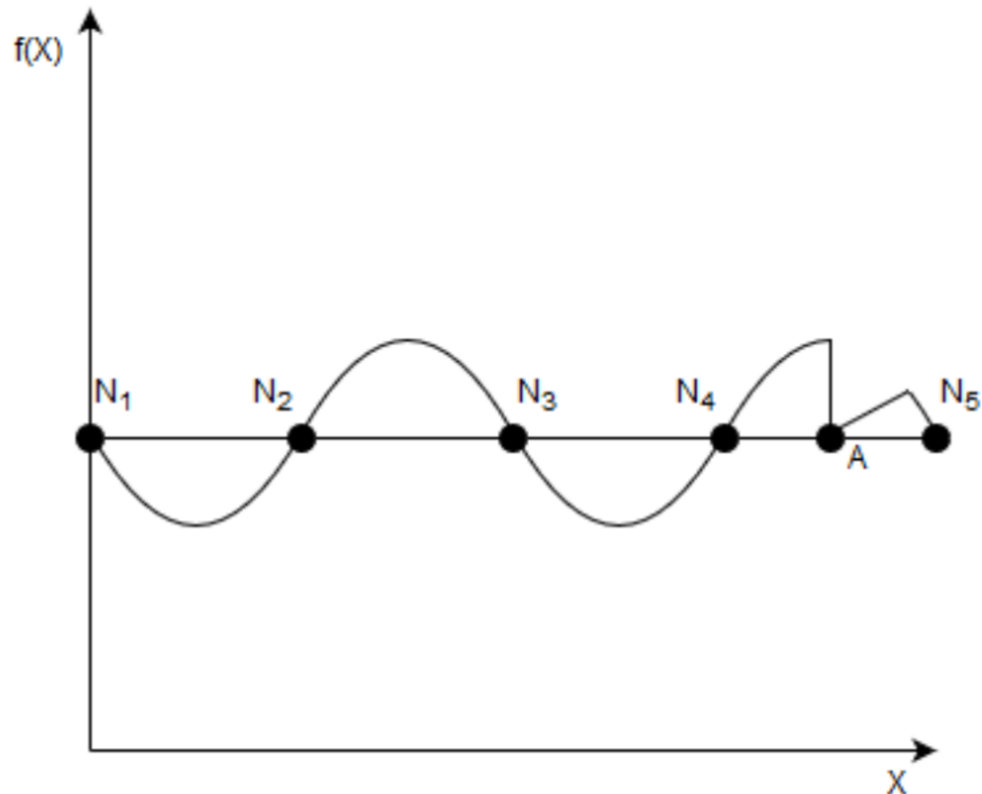


Рисунок 4 - Приклад контекстуальних аномалій

- колективні аномалії відповідають випадкам, коли сукупність екземплярів даних аномальна по відношенню до всього набору даних (на рисунку 5 область А є колективною аномалією).

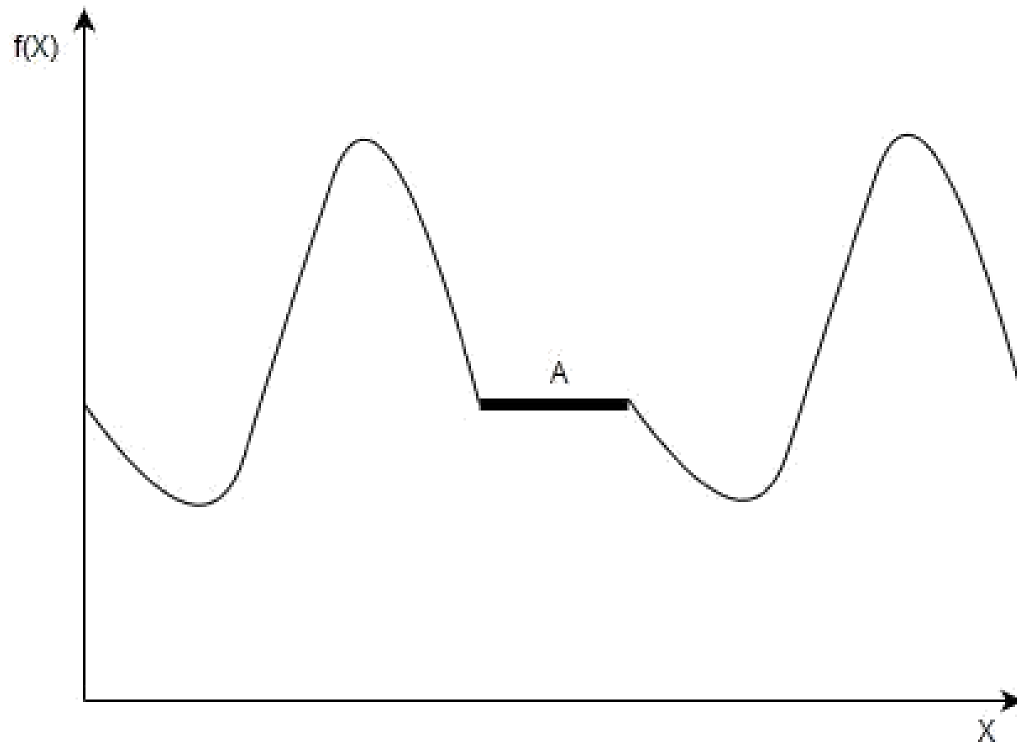


Рисунок 5 – Приклад колективних аномалій

Точкові аномалії найбільш поширений тип викидів. Якщо уявити дані у вигляді точок, то такі аномалії будуть сильно вибиватися із загальної картини.

Ключове завдання їх виявлення - з'ясувати порогове значення відхилення, яке вказує на потенційний викид, що представляє собою окрему велику область для досліджень. Точкові аномалії часто використовуються в системах контролю транзакцій для виявлення шахрайства приклад результату точкових аномалій можна побачити на рисунку 6.

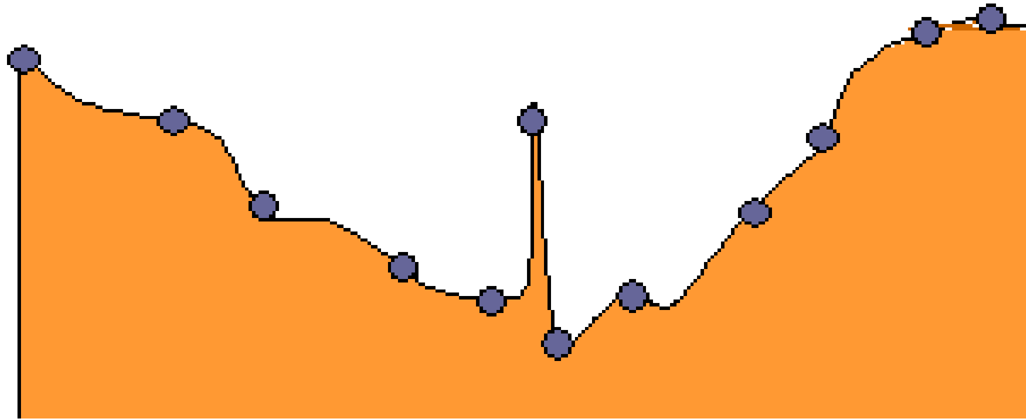


Рисунок 6 - Точкові аномалії

Зі сказаного ви можете справедливо зробити висновок, що маркування точкових аномалій не завжди буде спрацьовувати правильно, адже в задачах аналізу даних можуть бути абсолютно різні умови і аспекти. Іншими словами, аномалії можуть залежати від контексту.

Контекстуальні аномалії. Припустимо, що ми знаходимося в місті Калькутта в Індії, і температура повітря сьогодні становить 32 градуси Цельсія. Нормальна це температура? Без додаткової інформації відповісти на це питання важко: потрібно знати час року, місце розташування, середньодобову температуру за останні 10 років і т. Д. Якщо в Калькутті зараз літо, то така температура буде нормою. Але якщо зима, то потрібно дослідити ситуацію глибше.

Візьмемо інший приклад: ми всі знаємо про масштабні зміни в кліматі, що викликають глобальне потепління. Можна звернутися до останніх новин: «Март на Алясці в цьому році був незвичайно теплим, чого не спостерігалося за всю історію».

Зверніть увагу на фразу «незвично теплий». У випадку з Аляскою мається на увазі 15 градусів Цельсія, але для інших країн така температура не буде аномальною.

Подібні випадки називаються контекстуальних аномаліями, коли відхилення залежить від контекстної інформації, що регулюється контекстними і поведінковими атрибутами. У цьому прикладі контекстний атрибут - місце розташування, а атрибут поведінки - температура.

На рисунку 7, який нижче показаний часовий ряд даних за певний період. Графік був додатково згладжений ядром оцінки щільності, щоб показати кордон тренда. Значення не виходять за межі нормальних, але в них все одно присутні аномальні точки (виділені помаранчевим), що залежать від часу.

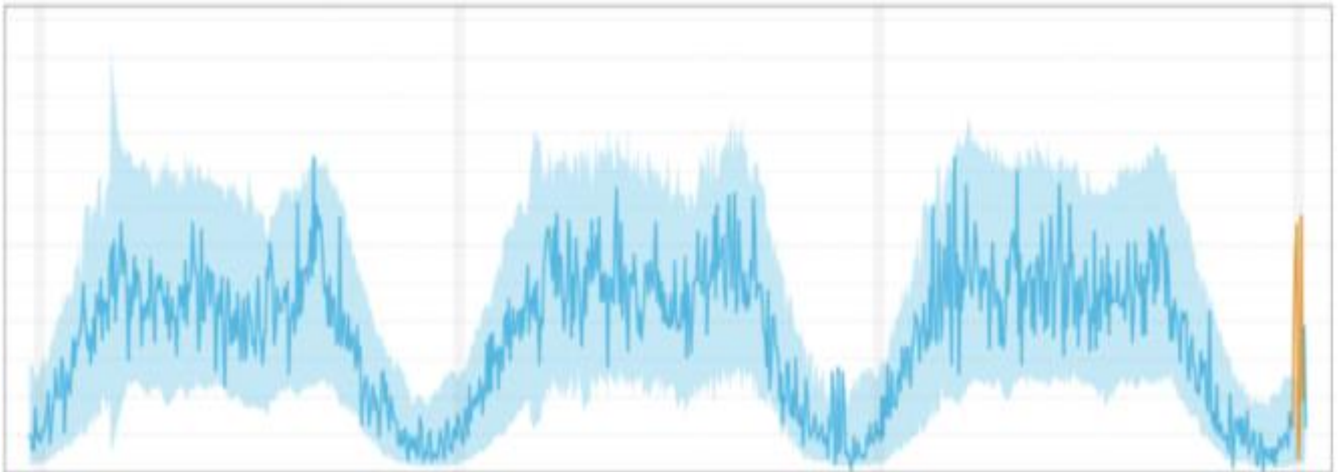


Рисунок 7 - Контекстуальні аномалії

У різних контекстах виявлення аномалій залежить від специфіки даних. Тому в більшості випадків для формалізації цих контекстів слід консультуватися з фахівцями в конкретній предметній області.

На рисунку 8 обведені пунктиром точки утворюють область, яка істотно відрізняється від інших точок.

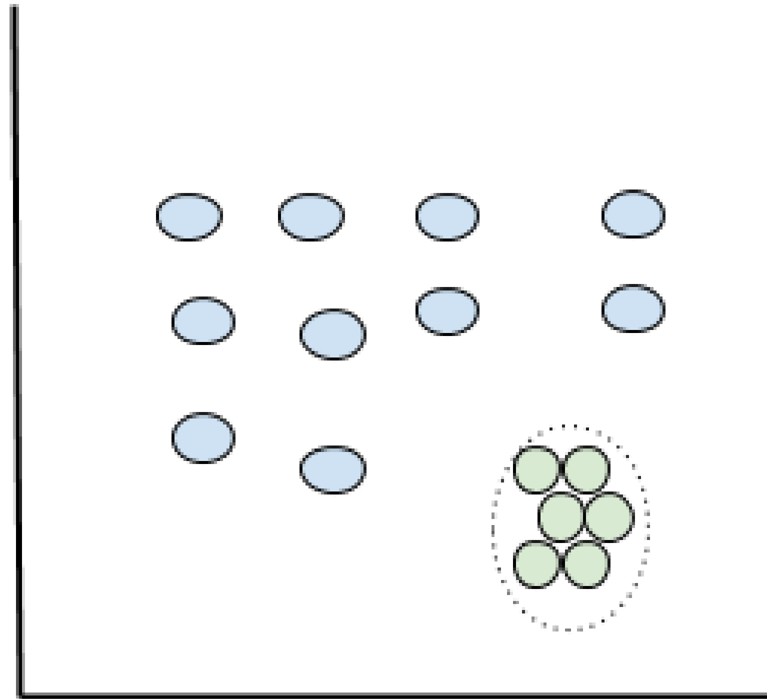


Рисунок 8 - Колективні аномалії

Це - приклад колективних аномалій. Їх основна ідея полягає в тому, що такі згруповані точки не можуть бути аномаліями окремо зображено на рисунку 8. Для прикладу візьмемо щоденні поставки текстильної фабрики. У подібних галузях часто трапляються затримки замовлень. Але якщо в якийсь із днів відбувається занадто багато затримок, може знадобитися додаткове розслідування. Одна відкладена поставка не зіграє ролі, але при аналізі подібної ситуації повинна враховуватися загальна картина.

Колективні аномалії цікаві тим, що ви дивитеся не на окремі точки, а аналізуєте їх поведінку в цілому.

Тепер, коли ми познайомилися з основами аномалій, спробуємо співвіднести їх з контекстом машинного навчання. Давайте з'ясуємо, чому вони важливі і коли на них слід звертати увагу.

Для вирішення поставленого завдання передбачається здійснювати пошук точкових аномалій, тому що об'єкти аналізу, списки токенів,

розглядаються ізольовано один від одного і не містять контекстних атрибутів. Як контекстуальних аномалій можуть розглядатися, наприклад, окремі частини списку токенів в контексті всього списку або окремі маркери в контексті інших токенів. Однак, відповідно з поставленим завданням, будуть розглядатися аномалії, що представляють собою список токенів цілком.

В рамках поставленого завдання виявлення нових класів аномалій всі дані розглядаються як немарковані, тобто без інформації про те, які приклади коду є аномальними і до якого класу аномалій вони належать. А значить для вирішення завдання пошуку аномалій доречно використовувати алгоритми навчання без учителя. Розглянемо існуючі методи навчання без учителя, що дозволяють здійснювати виявлення точкових аномалій.

Однокласовий метод опорних векторів (One-class SVM) [25] - представник методів опорних векторів, який відноситься до алгоритмів класифікації і передбачає наявність лише одного класу. Даний алгоритм дозволяє проводити навчання без учителя.

Суть метода полягає в тому, що набір вихідних векторів відділяється від точки початку координат за допомогою побудови розділяє гиперплоскости в просторі вищої розмірності. За результатами роботи методу передбачається отримання двох груп даних: перша група складається з об'єктів, які відносяться до єдиного класу, і ці дані не є аномальними, а друга група складається з об'єктів, які не вдалося визначити до цього класу, і вони оголошуються аномальними. На додаток до цього метод передбачає отримання числової оцінки аномальність.

Метод опорних векторів зводить навчання класифікатора до оптимізаційної задачі, яка розв’язується евристичними алгоритмами. Для побудування нелінійних класифікаторів використовується розширення простору та функції ядер.

Метод SVM у тестах перемагає інші методи за швидкістю та точністю категоризації. При різному підході до вибору ядер метод може емулювати роботу інших математичних методів. SVM може працювати як нейронна мережа, проте таке використання обмежує його призначення, оскільки метод значно перевершує їх за можливостями.

SVM може бути успішно застосований для керування складними електромеханічними системами, він може забезпечити адаптивність алгоритмів керування, виконувати функції спостерігача, ідентифікатора невідомих параметрів, деякої еталонної моделі, за його допомогою можна керувати складними нелінійними об’єктами, а також об’єктами зі стохастичними параметрами.

Зокрема, метод опорних векторів шукає поділяючу поверхню, максимально віддалену від будь-яких точок даних. Відстань між цією поверхнею і найближчою точкою даних називається зазором класифікатора.

У методі опорних векторів обов'язково мається на увазі, що вирішальна функція цілком визначається (звичайно малою) підмножиною даних, що впливають на положення роздільника. Ці точки називаються опорними векторами, тому що у векторному просторі точку можна розглядати як вектор між початком координат і цією точкою зображено на рисунку 9. Інші точки даних не впливають на вибір поділяючої поверхні.

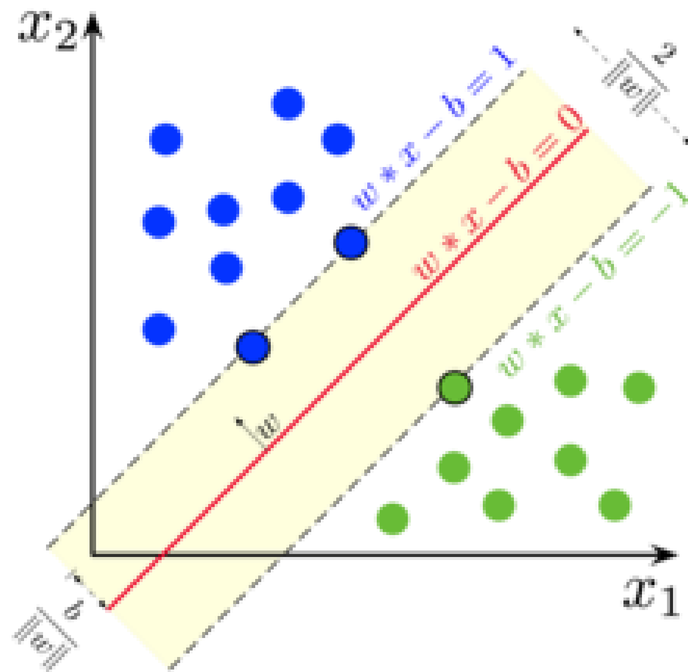
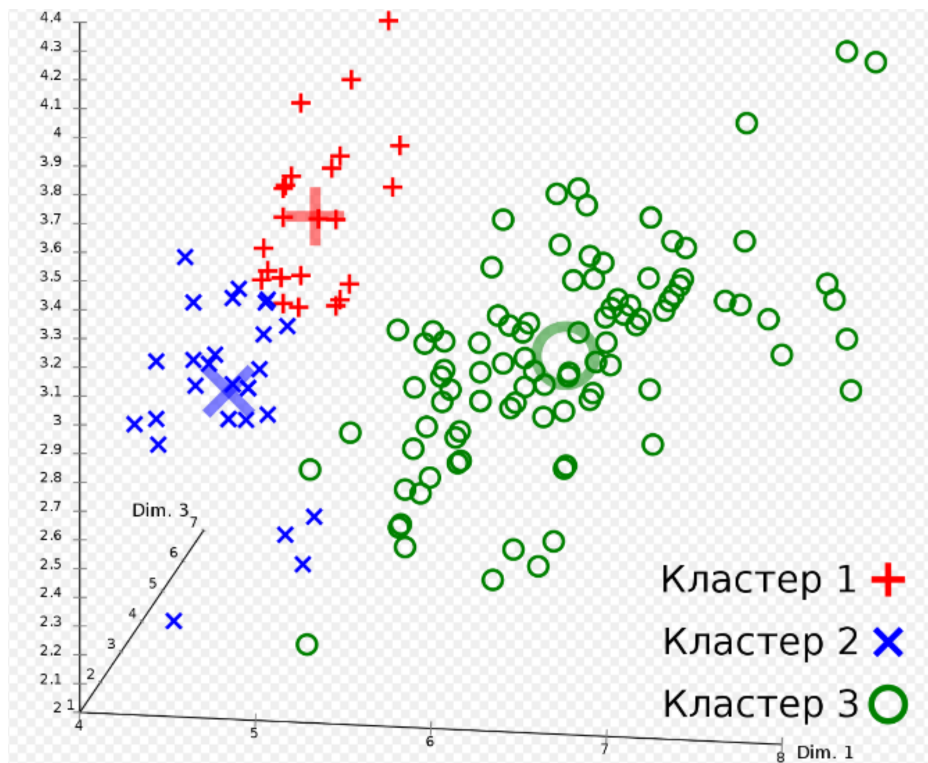


Рисунок 9 - Однокласовий метод опорних векторів

Максимізація зазору виглядає гарною ідеєю, оскільки точки, що лежать поблизу поділяючою поверхні, породжують велику невизначеність; з імовірністю 50% класифікатор може прийняти кожне з двох рішень. Класифікатор з великим зазором знижує невизначеність рішення. Тим самим він створює визначений запас надійності: невелика помилка виміру чи невелика зміна документа не призведе до неправильної класифікації. По своїй конструкції класифікатор SVM вимагає, щоб навколо поділяючої поверхні був широкий зазор. Якщо спробувати помістити між класами широку смугу, то діапазон кутів, при якому це можна зробити, виявиться набагато меншим, чим для гіперплощини. У результаті ємність запам'ятовування моделі зменшується, і можна чекати, що здатність моделі правильно узагальнювати тестові дані збільшується.

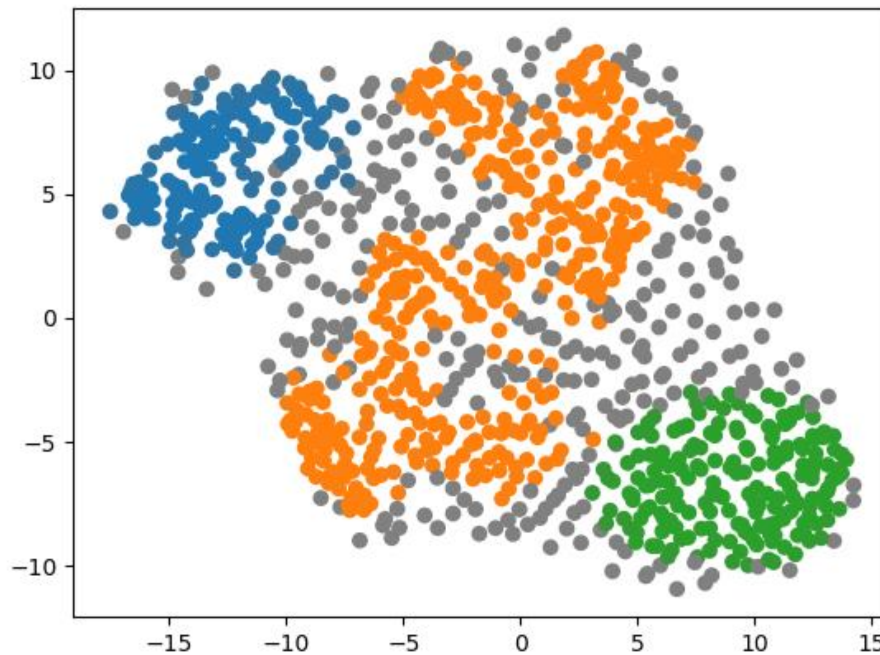
Заснована на щільності просторова кластеризация для додатків з шумами (Density-based spatial clustering of applications with noise, DBSCAN) [26] - алгоритм кластеризації, заснованої на щільності розташування точок. Суть

алгоритму полягає в тому, що в просторі вихідних векторів алгоритм групує разом точки, які тісно розташовані, тобто ті точки, у яких багато близьких сусідів, позначаючи як аномалії точки, які знаходяться самотньо в областях з малою щільністю, тобто найближчі сусіди яких розташовані далеко приклад продемонстровано на рисунку 10. При цьому кількість кластерів не ставить явно, а обчислюється в результаті роботи алгоритму.



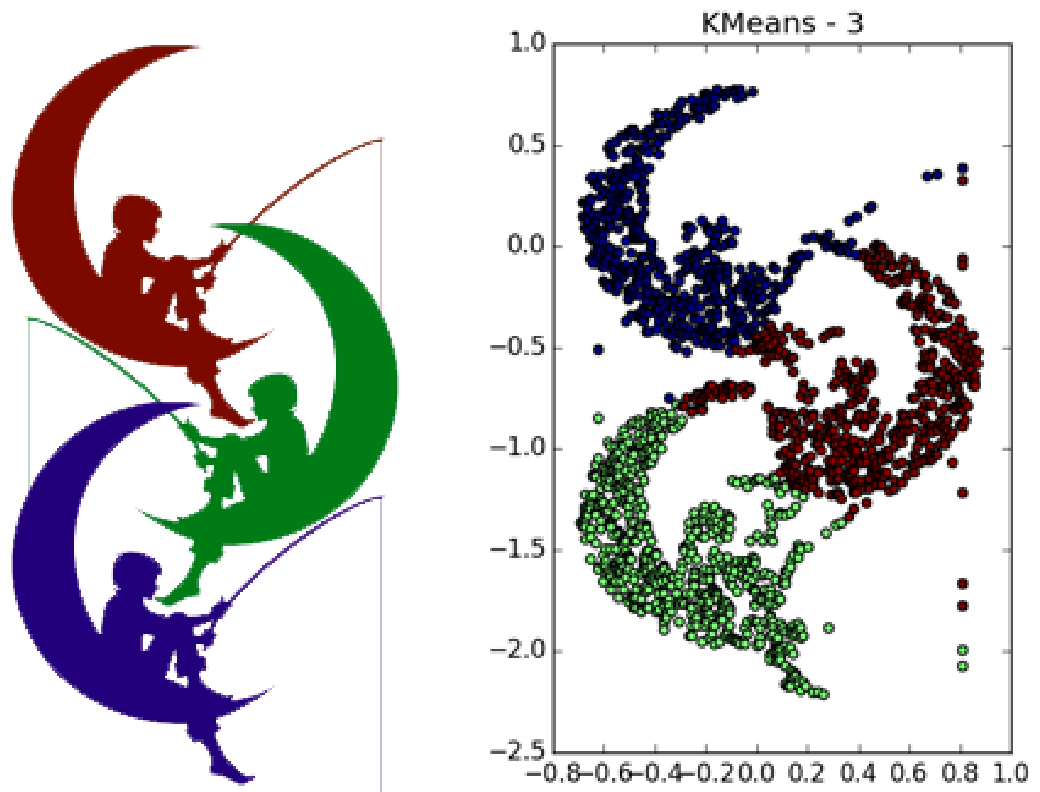
Рисунку 10 – Результат алгоритму DBSCAN

Ієрархічна заснована на щільності просторова кластеризация для додатків з шумами (Hierarchical Density-based spatial clustering of applications with noise, HDBSCAN) [27] - алгоритм кластеризації, що розширює алгоритм DBSCAN шляхом перетворення його в ієрархічний алгоритм кластеризації, який використовує методи вилучення плоскою кластеризації на основі стабільності кластерів продемонстровано на рисунку 11.



Рисунку 11 – Результат алгоритму HDBSCAN

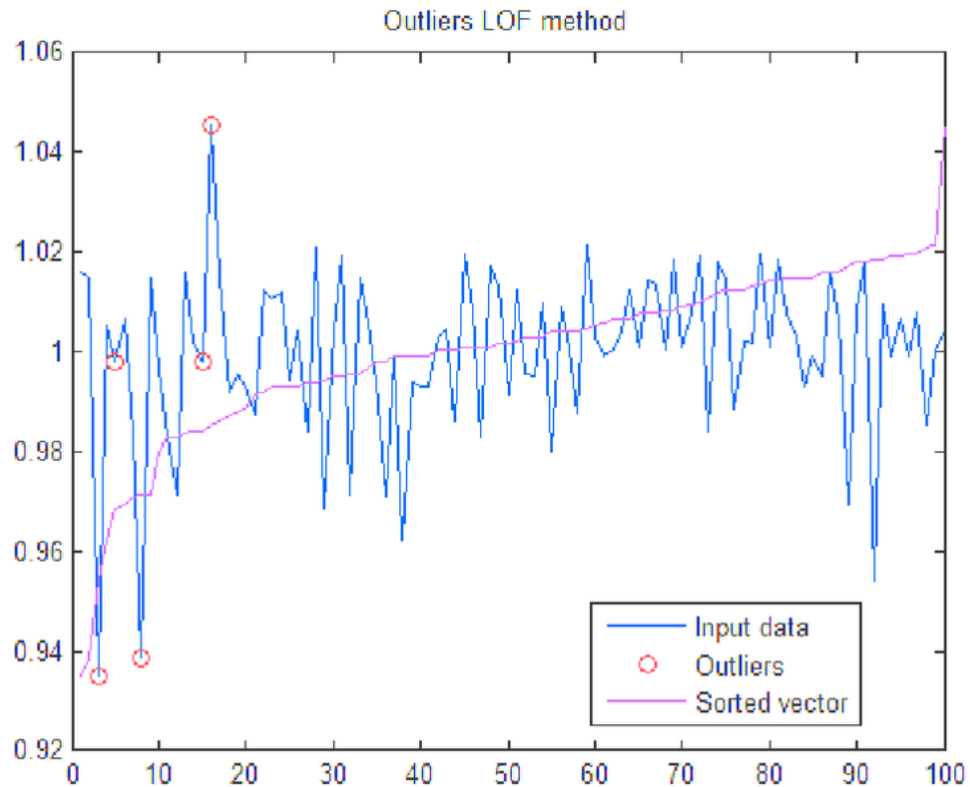
Метод k-середніх (k-means) [28] - алгоритм кластеризації, який складається в розбитті набору елементів на задану кількість кластерів. Метод полягає в ітеративному обчисленні центру мас кластерів з метою мінімізації сумарного квадратичного відхилення точок кластерів від їх центрів. Ітерації зупиняються тоді, коли відстані всередині кластерів перестають змінюватися. Метод кластеризації даних k-means дозволяє згрупувати екземпляри набору даних в кластери найбільш похожих один на одного елементів шляхом мінімізації сумарного квадратичного відхилення точок кластерів від центрів цих кластерів продемонстровано на рисунку 12. Для векторизації вихідного коду застосований метод токенізації програми і підрахунок статистики оцінки важливості токенів TF-IDF. Як параметр метод кластеризації приймає кількість кластерів, на які буде вироблено поділ вихідного набору даних.



Рисунку 12 - Метод k-середніх (k-means)

Даний інструмент дозволяє розділити знайдені аномалії на групи близьких за змістом прикладів вихідного коду, що спрощує процес аналізу отриманих аномалій за рахунок скорочення прикладів, які необхідно розглянути для формування опису знайдених аномалій. Особливо істотна користь даного інструменту при виявленні великої кількості прикладів аномального коду.

Локальний рівень викиду (local outlier factor, LOF) [29] - алгоритм виявлення аномалій, суть якого полягає в оцінці локальної щільності об'єктів і порівняння з локальними плотностями їх сусідів продемонстровано на рисунку 13. Точки, в яких щільність істотно відрізняється від щільності сусідів, оголошуються аномальними.



Рисунку 13 - Локальний рівень викиду

Репликаторні нейронні мережі [30] можуть використовуватися з метою виявлення аномалій в даних. Суть роботи даного алгоритму полягає у відновленні вихідних даних з проміжного представлення, згенерованого автоенкодером. Ступінь аномальності в даному випадку визначається розміром втрат, отриманих під час декодування даних.

Статистичні методи [31] засновані на ідеї зіставлення даних деякого статистичному розподілу. Ступінь аномальності визначається як величина відхилення примірника даних від перевіряється розподілу. Деякі методи даної групи вимагають припущення про розподіл даних, інші можуть обчислювати можливий розподіл даних самостійно.

Всі перераховані методи підходять для вирішення завдання виявлення точкових аномалій в нерозмічену даних. Вони всі ставляться до класу задач

навчання без учителя і всі вони дозволяють оцінити в тому чи іншому вигляді ступінь аномальності об'єкта даних. Однак в даній роботі пропонується зробити акцент на використанні методів локальний рівень викиду, однокласових метод опорних векторів, DBSCAN і HDBSCAN. Такий вибір алгоритмів обумовлений прагненням використовувати в рішенні методи, раніше не вживані в системі виявлення кодових аномалій на мові Kotlin, тому їх використання може дозволити знаходити нові класи аномалій і розширить інструментарій системи пошуку кодових аномалій. Незважаючи на це метод локальний рівень викиду раніше використовувався системою. На нього впав вибір через порівняно невеликих часових витрат, які необхідні для його роботи, і високої якості розпізнавання аномалій, показаного раніше в рамках апробації системи виявлення кодових аномалій.

Найкращим варіантом для системи виявлення кодових аномалій використати методи локального рівня викиду, метод однокласових опорних векторів, DBSCAN і HDBSCAN. Такий вибір алгоритмів обумовлений прагненням використовувати в рішенні методи, раніше не вживані в системі виявлення кодових аномалій мовою Kotlin, тому їх використання може дозволити знаходити нові класи аномалій і розширити інструментарій системи пошуку кодових аномалій.

ВИСНОВКИ ДО РОЗДІЛУ 1

У розділі досліджено галузі, де застосовується виявлення неоптимального коду, а також розглянуто та проаналізовано специфіку системи пошуку некоректного коду, написаного мовою програмування Kotlin. Показано, що основним завданням системи є синтаксичний аналіз програмного коду мово. Kotlin, а саме аналіз таких структур представлення програми, як дерева розбору і списку інструкцій JVM.

Досліджено алгоритми виявлення неоптимального використання коду обрані методи локального рівня викиду, метод однокласових опорних векторів та методи, що базуються на кластеризації, до яких відносяться однокласовий метод опорних векторів, DBSCAN і HDBSCAN, кожний з яких запускається на двох наборах даних: один - для списків типів токенів, інший для списків лексем.

Проаналізовано кластерний аналіз та проаналізовано методи кластеризації. Більш детально було розглянуто k-means метод, що дозволяє згрупувати екземпляри набору даних в кластери найбільш похожих один на одного елементів шляхом мінімізації сумарного квадратичного відхилення точок кластерів від центрів цих кластерів.

Найкращим варіантом для системи виявлення кодових аномалій використати методи локального рівня викиду, метод однокласових опорних векторів, DBSCAN і HDBSCAN. Такий вибір алгоритмів обумовлений прагненням використовувати в рішенні методи, раніше не вживані в системі виявлення кодових аномалій мовою Kotlin, тому їх використання може дозволити знаходити нові класи аномалій і розширити інструментарій системи пошуку кодових аномалій.

2. РЕАЛІЗАЦІЯ СПОСОБУ ПОШУКУ НЕОПТИМАЛЬНОГО КОДУ МОВОЮ KOTLIN ЗА ДОПОМОГОЮ СТАТИСТИЧНОГО АНАЛІЗУ

Схематичне зображення рішення задачі виявлення неоптимального коду зображено на рисунку 14. Розглянемо поетапно рішення даної задачі.

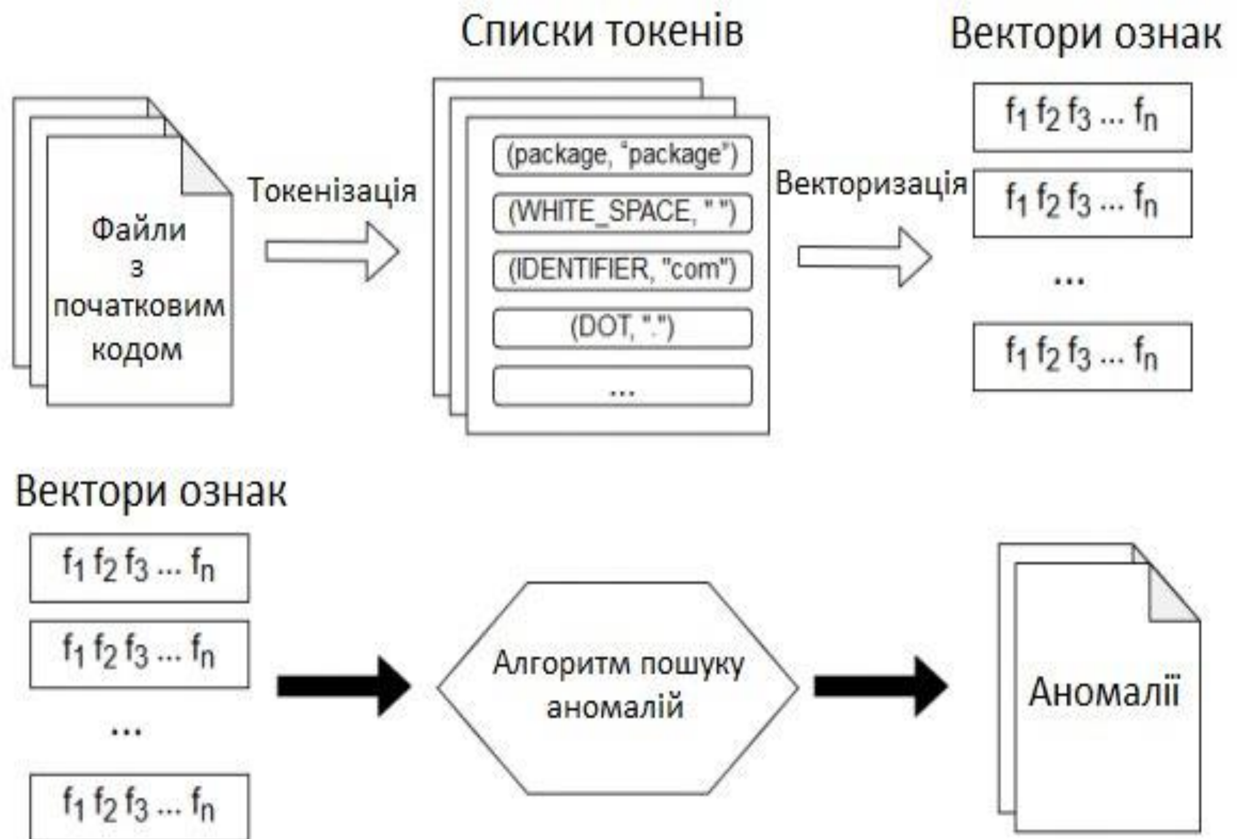


Рисунок 14 - Рішення задачі виявлення неоптимального коду

Для пошуку неоптимального коду необхідно розмістити дуже велику кількість вихідного коду на Kotlin.

По-перше, кодові аномалії зустрічаються рідко, і чим більше вихідного кода є розпорядженні, тим більше аномалій вдається виявити.

По-друге, розмер набора данних напряму впливає на якість работи деяких методів машинного навчання. У зв'язку з цим розумно використати як джерело даних GitHub - найбільший веб-сервіс для розміщення ПО з відкритим вихідним кодом. За оцінкою кінця листопада 2020 року обсяг вихідного коду на мові Kotlin, опублікованого на GitHub, досяг 100 мільйонів строків коду і продовжує зростати [10].

GitHub пропонує зручний API для пошуку та скачування репозиторіїв з вихідним кодом, що має, однак, ряд обмежень. Так, GitHub не надає даних про мовний склад репозиторіїв, а лише автоматично визначає один основний мову. Крім того, на кожен пошуковий запит пропонується не більше, ніж 1000 деталізованих результатів.

Збір набору даних з вихідним кодом програм і збірок проектів відбувається за допомогою сервісу GitHub з використанням наданого API. В програмі задається потрібна кількість сторінок з репозиторію для скачування та задаються параметри за яким скачуються програми написані мовою Kotlin.

```
start_k = 1
finish_k = 2
for k in range(start_k, finish_k):
  req = 'https://api.github.com/search/repositories?q=+language:kotlin&page=%d' % k
  print(req)
  response = requests.get(req)
  json_file = response.json()
  start = 0
  finish = 30
  print(len(json_file["items"]))
  for i in range(start, finish):
    arr = json_file["items"][i]
    id = arr["id"]
    full_name = arr["full_name"]
    name = arr["name"]
    str = "curl -L https://api.github.com/repos/%s/zipball > %s_%s.zip" % (full_name, name, id)
    p = subprocess.Popen(str, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    out, err = p.communicate()
```

Після скачування репозиторіїв з ресурсу GitHub потрібно розархівувати, дістати з папок усі файли з розширенням .kt та впорядковано зберегти.

```
def createSnippetsDir(folder, repo, snipFolder, counter):
    for file in os.listdir(folder):
        path = os.path.join(folder, file)
        path.replace(' ', '_')
        if os.path.isfile(path) and path.endswith(".kt"):
            counter.incFile()
            extractSnippets(path, file, repo, snipFolder, counter)
        elif os.path.isdir(path):
            createSnippetsDir(path, repo, snipFolder, counter)
    return

def extractSnippets(path, file, repo, snipFolder, counter):
    newFileName = file[:file.rindex('.')]
    name = snipFolderName + "\\" + repo + "_" + newFileName
    i = 1
    while os.path.exists(name):
        name2 = name + "_" + str(i)
        if os.path.exists(name2):
            i += 1
        else:
            name = name2
    os.mkdir(name)
    snipId = 1
    f = open(path, 'r')
    state = 0
    fileOutput = open(path + "x", 'w')
    numBrackets = 0
    try:
        for line in f:
            counter.incLine()
            if state == 0:
                if line.find("fun") != -1:
                    counter.incSnippets()
                    state = 1
                    fileOutput = open(name + "\\" + snipFolder + "_" + str(snipId) + ".kt", 'w')
                    fileOutput.write(line)
                    fileOutput.write("\n")
                    for c in line:
                        if c == '{':
                            numBrackets += 1
                        elif c == '}':
                            numBrackets -= 1
```

2.1 Токенізація програми

Для побудови представлення програми у вигляді структури "списка токенів" використовується компілятор мови програмування Kotlin.

Токенізація - процес розбиття текстового документа на окремі слова, які називаються токенами. Приклад токенізації зображено на рисунку 15.

```
# import nltk
# nltk.download('punkt')

from nltk.tokenize import word_tokenize

tokens = nltk.word_tokenize(cleaned_review)

print(cleaned_review)
print(tokens)
```

```
a touching movie it is full of emotions and wonderful acting i could have sat through it a second time
['a', 'touching', 'movie', 'it', 'is', 'full', 'of', 'emotions', 'and', 'wonderful', 'acting', 'i', 'could', 'have', 'sat', 'th
rough', 'it', 'a', 'second', 'time']
```

Рисунок 15 – Приклад токенізації

Крім визначення границь лексем, цей блок також виконує попереднє розподілення морфологічних атрибутів слов, переважна лексема в токенах. Для цього блок використовує інформацію в лексиконі та правила розподілу несловних лексем, а також ряд вспомогальних алгоритмів, включаючи нерозміщення розподілу.

При розподіленні слов визначаються такі торгові марки, як придатність до певної частини речей та набору граматичних атрибутів. Для словникових лексем також визначається словникова стаття, формою якої є лексеми

Для більшості європейських мов словника так чи іначе розділені пробелами та іншими символами-розробниками - наприклад, запятами. Список символів-розділителів задається для кожного мови окремо, як буде детально описано далі.

Є також мови, в яких слова не виділяються явно, а текст представлений із самих сліпкових послідовностей символів (ієрогліфов).

Тип токенізації за певним параметром Сегментація в описі мови. Якщо значення пробілу означає використання пробелів у якісному природному кордоні слов Значення dictionary_lookup означає, що для розбиття на слові використовується перегляд по лексикону за принципом "спробувати знайти слова, прочитані з поточною позицією, потім - наступне слово після них".

В останньому випадку замість вбудованого в движок алгоритму можна використовувати зовнішній токенізатор, оформлений як динамічно завантажувана бібліотека. Ім'я файлу бібліотеки без розширення .DLL або .SO задається параметром SegmentationEngine в описі мови. Наприклад, для японської мови у файлі sg_languages.sol можна переглянути такі строки:

З метою перетворення вихідного коду програми на мові Kotlin в список токенів був склонірован репозиторій з компілятором мови Kotlin, розташований на загальнодоступному ресурсі GitHub [6]. В даному репозиторії в рамках тестової інфраструктури була додана можливість виведення списку токенів в процесі компіляції програми. Шлях до директорії, в якій необхідно запустити генерацію списку токенів по вихідного коду, задається окремим параметром.

У результаті роботи даного модуля поряд з кожним екземпляром вихідного коду на мові Kotlin створюється файл з розширення "txt", в якому у вигляді списку на кожній новій рядку перераховані маркери. Кожен токен є структурою з двох елементів: (тип токена, лексема). В результуючому файлі елементи розділені пропуском.

Частина файлу з шаблонами для токенів:

package ('package')
WHITE_SPACE (' ')
IDENTIFIER ('com')
DOT ('.')
IDENTIFIER ('ground0')
DOT ('.')
IDENTIFIER ('transaction')
WHITE_SPACE ('\n\n')
IDENTIFIER ('import')
WHITE_SPACE (' ')
IDENTIFIER ('android')
DOT ('.')
IDENTIFIER ('support')
DOT ('.')
IDENTIFIER ('test')
DOT ('.')
IDENTIFIER ('InstrumentationRegistry')
WHITE_SPACE ('\n')
IDENTIFIER ('import')
WHITE_SPACE (' ')

Для зручності подальшої обробки даних був розроблений інструмент [7], який на основі отриманого файлу створює два нових файлу: один зі списком типів токена, який має розширення "token", другий зі списком значень лексем, що має розширення "value".

Нище наведений основний код за допомогою якого відбувається токенізація:

```
def feature_extractor_token():
    counter = Counter(0, 0, 0)
    for file in os.listdir(folder):
        path = os.path.join(folder, file)
        #path.replace(' ', '_')
        if os.path.isfile(path) and path.endswith(".txt"):
            counter.incFile()
            extractTokenList(path, file, snipFolder, counter)
        elif os.path.isdir(path):
            createSnippetsDir(path, file, snipFolder, counter)
    print(tokens)
    print(len(tokens))
    print(counter.getFiles())
    return
def extractTokenList(path, file, snipFolder, counter):
    f = open(path, 'r')
    token_types = [line.split(' (')[0] for line in f]
    print(token_types)
    for t in token_types:
        if t not in tokens:
            tokens.append(t)
```

2.2. Векторизація даних

З метою охопити більш широке коло потенційних аномалій, які будуть знайдені за допомогою аналізу списку токенів, прийнято рішення розглянути два підходи до подання програми: список типів токенів і список значень лексем. У кожному з цих підходів пропонується застосовувати різні методи векторизації. Схема перетворення списку токенів в набір векторів представлена на рисунку 16.

Рисунок 16 - Векторизація токенів

2.2.1. Векторизація списку типів токенів

Отже, як тільки текст перетворився в очищену нормалізовану послідовність слів, запускається процес їх векторизації - перетворення в числові вектора. Для такої трансформації використовуються спеціальні моделі.

Для застосування методів виявлення неоптимального коду необхідно зібраний набір списків типів токенів перетворити на вигляд числових векторів. Для вирішення даного завдання застосовується підхід до векторизації текстів, який використовується в області обробки природної мови (NLP). Модель, яка називається "мішок слів" (bag-of-words), містить ідею представлення даних у вигляді "мішка" або безлічі слів. Як і рядок і граматика слів в даній моделі не враховуються. Стосовно даних реалізованої системи словом буде ідентифікатор типу токена. Безліч слів буде складатися з різних типів токенів, які зустрічаються у всьому наборі даних.

Одна з базових моделей векторизації - заповнення вектора значеннями частоти входження слова в файлі. Для реалізації даного методу векторизації необхідно отримати безліч всіх слів. Для цього розроблено окремий модуль, що здійснюють обхід всього званих набора даних і побудова безлічі слів датасета. Потім модуль для кожного файлу будує вектор з підрахунком кількості входжень слів.

Однак, метод підрахунку частоти входження слів в документі або файлі володіє недоліком, пов'язаним із значним впливом слів, які зустрічаються в файлі дуже часто. Рішенням цієї проблеми є метод одноразового кодування (one-hot encoding) або метод логічного векторного кодування. Він полягає в тому, що для кожного файлу будується вектор зі всіляких слів, в якому позначається, чи входить в даний файл слово чи ні. Іншими словами, кожен елемент вектора відображає або наявність або

відсутність слова в файлі. Метод одноразового кодування зменшує проблему дисбалансу розподілу слів, спрощуючи документ до його реалізованих компонентів.

Модель "мішок слів" описує файл автономно, без урахування контекста інших файлів набору даних. Виникає ідея розглянути відносну частоту або рідкість входження слів в файлах проти часто-ти їх входження в інших файлах. Для втілення цієї ідеї існує метод TF-IDF (Term Frequency - Inverse Document Frequency). TF-IDF є статистичну міру, яка використовується для оцінки важливості слова в наборі даних. Міра складається з двох інших заходів: TF і IDF.

Специфіка метрики TF-IDF, на відміну від інших аналогічних, дозволяє виділяти особливості, характерні тільки для одного об'єкта колекції і таким чином виділяють даний об'єкт серед всіх інших елементів.

Підхід явного вилучення ознак хороший тим, що зазвичай отримані ознаки легко інтерпретувати і зрозуміти. Неявні ознаки більш важкі для розуміння, однак такий підхід в окремих випадках здатний визначати складні властивості коду, такі як, наприклад, семантичні залежності. Даний підхід спрямований на витяг фактів, що описують особливості конкретного об'єкта на противагу загальним характеристикам предметної області.

В основі запропонованого алгоритму лежить статистична міра TF-IDF [8], що дозволяє оцінити важливість слова в контексті документа, який є частиною деякої колекції документів. Важливість слова для документа пропорційна частоті вживання цього слова в даному документі, і обернено пропорційна частоті вживання слова у всіх інших документах з колекції.

TF являє собою відношення числа входжень слова до загальної кількості слів у файлі. Обчислюється за формулою:

$$TF(t;d) = \frac{n_t}{\sum_k n_k}; \quad (1)$$

де n_t є число вхождення слова t в файл, в знаменнику — загальне число слів в файлі.

IDF — це інверсія частоти, з якою слово зустрічається в файлі.

Обчислюється за формулою:

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

де | D | - загальне число файлів, $|\{d_i \in D | t \in d_i\}|$ знаменник - число файлів, в яких зустрічається слово t.

Міра TF-IDF є твором двох заходів і обчислюється за формулою:

$$TF - IDF(t, d, D) = TF(t, d) \times IDF(t, D).$$

Для кожного слова у файлі відбувається підрахунок метрик TF, IDF і TF-IDF. Всі три значення заносяться в результуючий вектор. У разі відсутності слова в файлі, відповідні значення в векторі заповнюються нулями.

Таким чином, для кожного типу токена в кожному файлі обчислюються 5 метрик: частота входження, присутність в файлі, метрики TF, IDF, TF-IDF. В результаті виходить вектор складається з 5 n елементів, де n - кількість різних типів токенів у всьому наборі даних.

Існують реалізації описаних способів векторизації в бібліотеках Scikit-learn і Gensym, однак вони припускають завантаження всього набору даних в оперативну пам'ять для обробки, що ні представляється можливим при достатньо великому обсязі даних. У звязку з цим був розроблений

незалежний інструмент векторизації, реалізовані на мові програмування Python з використанням стандартних бібліотек.

Робота системи полягає в обчисленні метрик шляхом послідовного обходу всього набору даних. Спершу відбувається обхід всіх списків токенів з метою формування загального безлічі, що складається з усіх типів токенів, що містяться в векторизованому наборі даних. Потім в процесі повторного проходу всього набору токенів здійснюється підрахунок метрик для побудови вектора. Всі побудовані вектора записуються в результуючий файл, який має розширення "tsv". Фрагмент файлу представлений нижче:

```
1   7.0   7.0   35.0   3.0   0.8823529411764706 0.5422927866210341
2  22.0  22.0  483.0  3.0   0.9574468085106383 0.7038812419196969
3  11.0  11.0  265.0  3.0   0.92   0.6387266666709874
4  14.0  14.0  209.0  3.0   0.9354838709677419 0.6488834147626433
5  4.0   4.0   23.0   3.0   0.8181818181818182 0.491481860290005
6  3.0   3.0   19.0   3.0   0.7777777777777778 0.49306614322200487
7   8.0   8.0   69.0   3.0   0.8947368421052632 0.5441166860514736
8  12.0  12.0  115.0  3.0   0.9259259259259259 0.6174525288874826
9  28.0  28.0  289.0  3.0   0.9661016949152542 0.7400471820418287
10 16.0  16.0  87.0   3.0   0.9428571428571428 0.6547868864416139
11 17.0  17.0  134.0  3.0   0.9459459459\459459 0.6801337210442505
12 11.0  11.0  147.0  3.0   0.92   0.6398379121943362
```

Приклад коду, за яким обраховується кількість входжень токена у файлі:

```
if "IDENTIFIER" in psiLine:
    featureNumberOfIdentifiers += 1.0
if "fun" in psiLine:
    featureNumberOfFuns += 1.0
if "LPAR" in psiLine:
```

```
    featureNumberOfLPAR += 1.0
if "RPAR" in psiLine:
    featureNumberOfRPAR += 1.0
if "COLON" in psiLine:
    featureNumberOfCOLON += 1.0
```

Приклад токенізації вихідного коду програми зображений на рисунку 17.

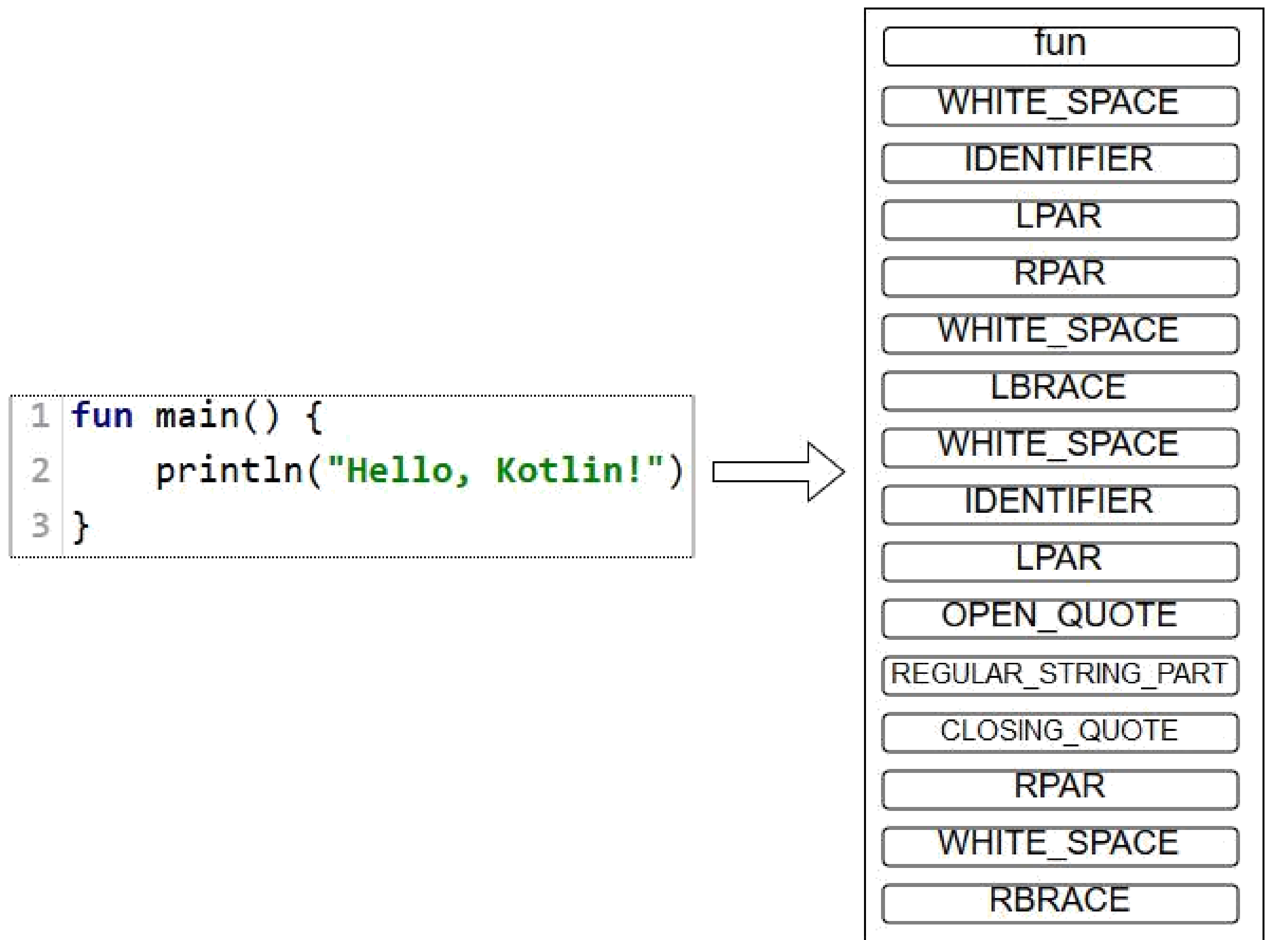


Рисунок 17 - Приклад токенізації програми

Для кожного типу токена в кожному файлі обчислюються 5 метрик: частота входження, присутність в файлі, метрики TF, IDF, TF-IDF. В результаті виходить вектор складається з 5 n елементів, де n - кількість різних типів токенів у всьому наборі даних. Приклад векторизації типів токенов показаний в таблиці 2.

Таблиця 2 - Приклад векторизації типів токенів

№	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

2.2.2. Векторизація списку лексем

Для векторизації списку лексем пропонується використовувати схожі ідеї з тими, які застосовувалися для векторизації списку типів токенів. Однак, кількість лексем у досить великому наборі даних буде вкрай великим. У зв'язку з цим пропонується будувати вектори, відповідні

сумарним і усередненим статистикам по лексемах для кожного типу токена. Як числових статистик, які характеризують значення лексем, пропонується вважати довжину лексеми і її хеш-код.

Існує кілька підходів до обчислення усереднених значень статистик. Будемо використовувати середнє арифметичне і логарифм середнього арифметичного для нормалізації значень. Такий вибір функцій усереднення зроблений через необхідність уніфікувати великий набір чисел, які досить великі за своїм значенням.

Хеш-код лексеми стандартної бібліотеки мови програмування Python містить 19 знаків, що робить число дуже великим і громіздким. Застосування функції логарифмування дозволяє скоротити число до дійсного числа з двозначною цілою частиною. Нижче представлений код, що відповідає ща векторизацію лексем:

```
def extract_info_2(path):
    global id_num
    try:
        f = open(path, "r")
        len_token = 0
        len_value = 0
        token_file_path = path.replace(end_name, end_token_name)
        try:
            f_t = open(token_file_path, "r")
            for line in f_t:
                tokens = line.split(" ")
                len_token += len(tokens)
        except Exception as e:
            f_b_v.write("=====\n" + str(e) + "\n")
            f_b_v.write(token_file_path + "\n=====\n")
            return
        value_len_count = defaultdict(int)
        value_hash_count = defaultdict(int)
        vec = []
        sum = 0
        sum_h = 0
        for line in f:
            values = re.findall("'.+?'", line)
            len_value += len(values)
            if len_value != len_token:
                s_f_v.write(path + "\n")
```

```
            return
        for cou in range(0, len_value):
            value_len_count[tokens[cou]] += len(values[cou])
            value_hash_count[tokens[cou]] += hash(values[cou])
            sum += len(values[cou])
            sum_h += hash(values[cou])
    vec.append(sum)
```

Таким чином, для кожного типу токена буде пораховано 7 статистик: присутність в файлі, сумарні довжина і значення хеш-коду, середнє арифметичне довжини і модуля хеш-коду і логарифм середнього арифметичного довжини і модуля хеш-коду. Всього вектор для кожного файлу буде містити 7 n елементів, де n - кількість різних типів токенів у всьому наборі даних.

```
    for t_t in token_types:
        vlc = value_len_count[t_t]
        if vlc > 0:
            vec.append(1)
        else:
            vec.append(0)
        vec.append(vlc)
        if sum > 0:
            vec.append(vlc / sum)
        else:
            vec.append(0)
        if sum > 0 and vlc > 0:
            vec.append(math.log(vlc / sum))
        else:
            vec.append(0)
        vhc = value_hash_count[t_t]
        vec.append(vhc)
        if sum_h > 0:
            vec.append(vhc / sum_h)
        else:
            vec.append(0)
        if sum_h > 0 and vhc > 0:
            vec.append(math.log(vhc / sum_h))
        else:
            vec.append(0)
    tsv_writer.writerow([id_num] + [path] + vec)
    id_num += 1
except Exception as e:
    f_b_v.write(str(e) + "\n")
    f_b_v.write("=====\n" + path + "\n=====\n")
```

```
for file in os.listdir(folder_for_value_vector):
    path = folder_for_value_vector + "/" + file
    try:
        if os.path.isfile(path) and path.endswith(end_name):
            extract_info_2(path)
        elif os.path.isdir(path):
            visit_folder_2(path)
    except Exception as e:
        f_b_v.write(str(e) + "\n")
        f_b_v.write(path + "\n")
```

Приклад перетворення вихідного коду програми в список лексем зображений на рисунку 18. Приклад векторизації списку лексем показаний в таблиці 2. Умовними позначеннями показані статистики, які обчислюються на основі побудованих лексем.

У графі "B" зазначено значення, що позначає чи присутній даний тип токена в файлі чи ні. У графі "SL" вказана сумарна довжина значень даного типу, "AL" - середня довжина значень даного типу, "LL" - логарифм середньої довжини даного типу. У графі "SH" розміщується сумарне значення хеш-кодів лексем даного типу, "AH" - середнє значення хеш-кодів даного типу, "LH" - логарифм середнього значення хеш-кодів даного типу.

2.3. Виявлення неоптимального коду

Для виявлення неоптимального коду обрані методи, засновані на кластеризації. Кластеризація або кластерний аналіз відповідає за розбиття заданої вибірки об'єктів (ситуацій) на підмножини, які називаються кластерами, так, щоб кожен кластер складався з схожих об'єктів, а об'єкти різних кластерів істотно відрізнялися. Завдання кластеризації відноситься до статистичної обробки, а також до широкого класу завдань навчання без вчителя.

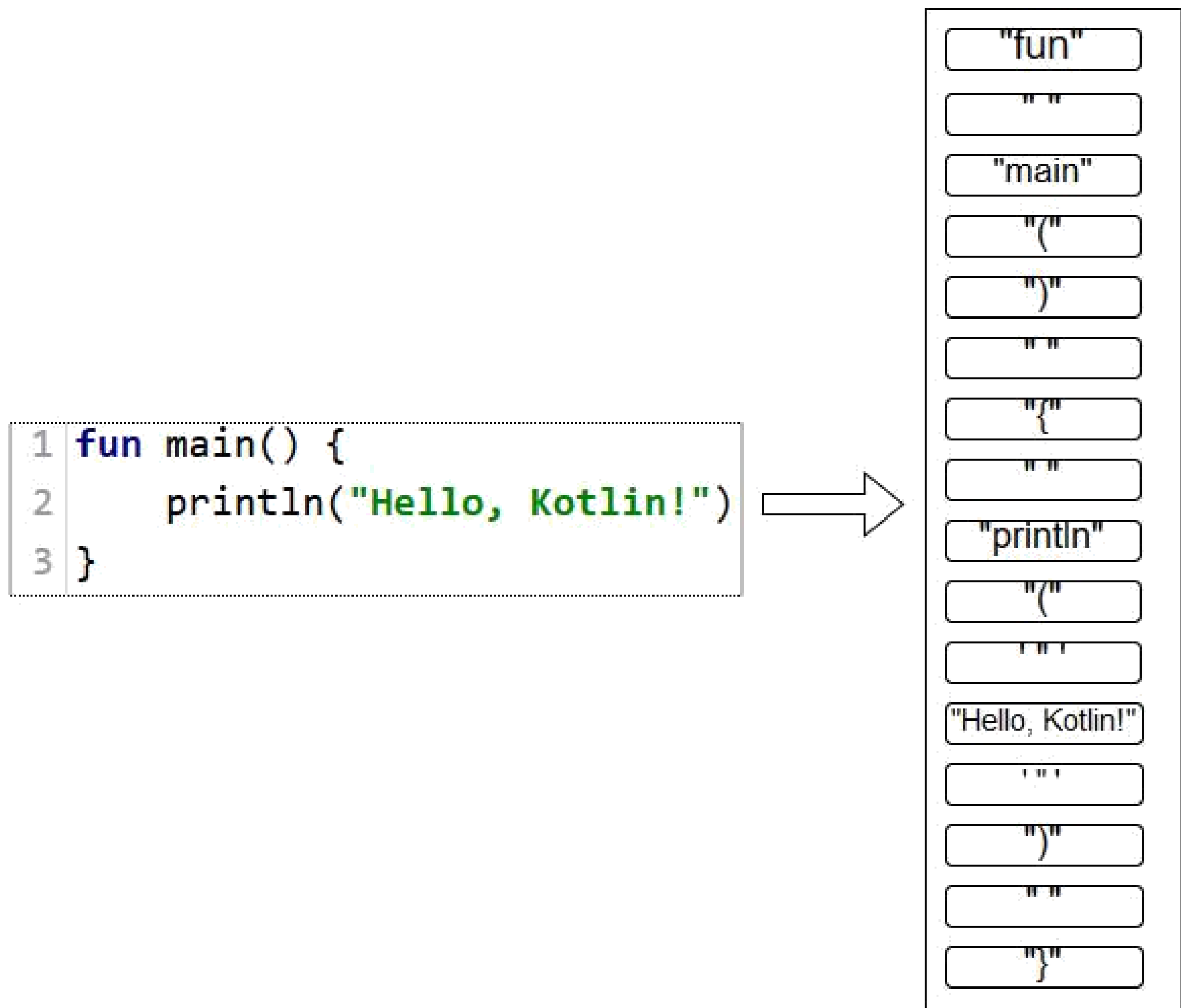


Рисунок 18 - Приклад лексемізації програми

Для виявлення неоптимального коду використовуються такі методи: локальний рівень викиду, однокласових метод опорних векторів, DBSCAN і HDBSCAN. Кожен з перечислених алгоритмів запускається на двох наборах даних: один для списків типів токенів, інший для списків лексем.

За результатами роботи метода однокласових опорних векторів передбачається отримання двох груп даних: перша група складається з об'єктів, які відносяться до єдиного класу, і ці дані не є аномальними, а друга група складається з об'єктів, які не вдалося визначити до цього класу, і вони оголошуються аномальними.

Суть алгоритму DBSCAN полягає в тому, що в просторі вихідних векторів алгоритм групує разом точки, які тісно розташовані, тобто ті точки, у яких багато близьких сусідів, позначаючи як аномалії точки, які знаходяться самотньо в областях з малою щільністю, тобто найближчі сусіди яких розташовані далеко. При цьому кількість кластерів не ставить явно, а обчислюється в результаті роботи алгоритму.

HDBSCAN - алгоритм кластеризації, що розширює алгоритм DBSCAN шляхом перетворення його в ієрархічний алгоритм кластеризації, який використовує методи вилучення плоскою кластеризації на основі стабільності кластерів.

Локальний рівень викиду - алгоритм виявлення аномалій, суть якого полягає в оцінці локальної щільності об'єктів і порівняння з локальними плотностями їх сусідів. Точки, в яких щільність істотно відрізняється від щільності сусідів, оголошуються аномальними. В результаті пошуку локальним рівенем викиду також було згенеровано графік за результатами вихідних даних зоюражено на рисунку 19.

В результаті роботи кожного методу виявлення аномалій утворюється два набору прикладів програмного коду, оголошеного аномальним.

Як реалізації методів виявлення аномалій була обрана стабільна реалізація алгоритмів, що входить до складу бібліотеки Scikit-learn.

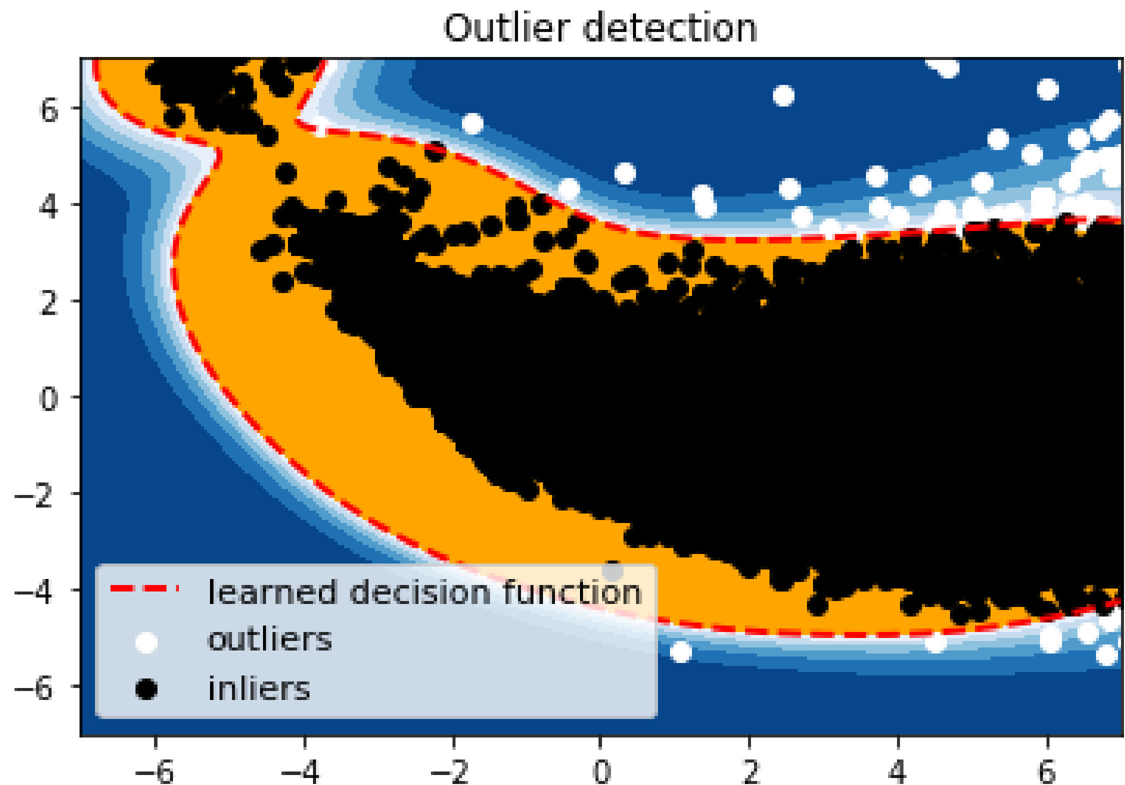


Рисунок 19 - Результати пошуку локальним рівенем викиду

Нижче продемонстровано основний код який відповідає за пошук неоптимального коду:

```
X = PCA(n_components=2).fit_transform(params)
num = X.shape[0]
OUTLIER_FRACTION = 0.01

clf = svm.OneClassSVM(kernel="rbf")
clf.fit(X)

dist_to_border = clf.decision_function(X).ravel()
threshold = stats.scoreatpercentile(dist_to_border,
        100 * OUTLIER_FRACTION)
is_inlier = dist_to_border > threshold

xx, yy = np.meshgrid(np.linspace(-7, 7, 500), np.linspace(-7, 7, 500))
n_inliers = int((1. - OUTLIER_FRACTION) * num)
n_outliers = int(OUTLIER_FRACTION * num)
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.title("Outlier detection")
plt.contourf(xx, yy, Z, levels=np.linspace(Z.min(), threshold, 7),
```

```
                    cmap=plt.cm.Blues_r)
a = plt.contour(xx, yy, Z, levels=[threshold],
                      linewidths=2, colors='red')
plt.contourf(xx, yy, Z, levels=[threshold, Z.max()],
                    colors='orange')
b = plt.scatter(X[is_inlier == 0, 0], X[is_inlier == 0, 1], c='white')
c = plt.scatter(X[is_inlier == 1, 0], X[is_inlier == 1, 1], c='black')
plt.axis('tight')
plt.legend([a.collections[0], b, c],
         ['learned decision function', 'outliers', 'inliers'],
         prop=matplotlib.font_manager.FontProperties(size=11))
plt.xlim((-7, 7))
plt.ylim((-7, 7))
plt.show()
```

Кількість лексем у досить великому наборі даних буде вкрай великим. У зв'язку з цим пропонується будувати вектори, відповідні сумарним і усередненим статистикам по лексемах для кожного типу токена. Як числових статистик, які характеризують значення лексем, пропонується вважати довжину лексеми і її хеш-код.

Приклад лексемізації програми продемонстровано нижче на рисунку 20. Для того щоб обчислювати вирази, необхідно вміти розбивати їх на окремі складові. Наприклад, вираз A * B - (W + 10) складається з таких елементів: A, *, B, -, (, W, +, 10 і). Кожен з них представляє єдину неподільну частину виразу. У загальному випадку необхідна функція, яка повертає один за одним всі елементи вираження. Ця функція також повинна вміти пропускати прогалини і символи табуляції і визначати кінець виразу.

Кожен елемент вираження називається лексемою (token). Тому функція, яка повертає чергову лексему, часто називається get_token (). У цій функції використовується глобальний покажчик на рядок з розбираємо виразом. У показаної тут версії функції get_token () цей глобальний покажчик називається prog. Мінлива prog описана глобально, оскільки вона повинна зберігати своє значення між викликами функції get_token () і бути доступною іншим функціям. Крім значення підлягає поверненню лексеми,

необхідно знати її тип. Для аналізатора, що розробляється в цьому розділі, знадобляться тільки три типи: змінна, число і роздільник. Їм відповідають константи VARIABLE, NUMBER і DELIMITER, (DELIMITER використовується як для операторів, так і для дужок.) Нижче наведено текст функції get_token () разом з необхідними глобальними описами, константами і допоміжної функцією:

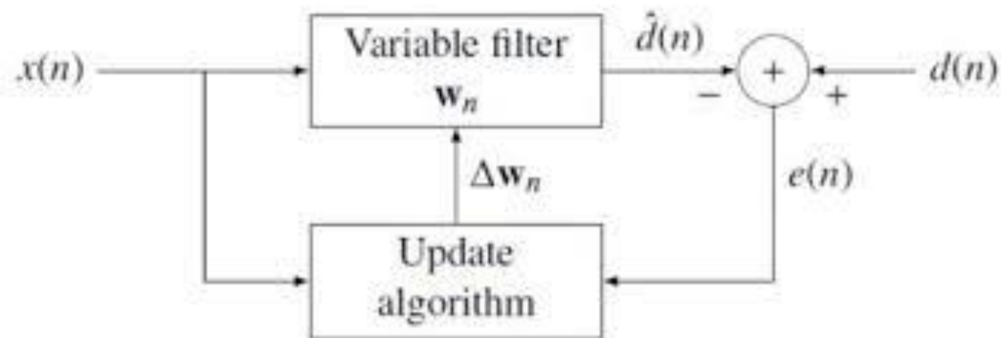


Рисунок 20 – Приклад лексимізації програми

Для пошуку неоптимального коду за допомогою лексем будемо використовувати середнє арифметичне і логарифм середнього арифметичного для нормалізації значень. Такий вибір функцій усереднення зроблений через необхідність уніфікувати великий набір чисел, які досить великі за своїм значенням.

Хеш-код лексеми стандартної бібліотеки мови програмування Python містить 19 знаків, що робить число дуже великим і громіздким. Застосування функції логарифмування дозволяє скоротити число до дійсного числа з двозначною цілою частиною. Нижче представлено код обробки лексем.

```
def extract_info_2(path):
    global id_num
    try:
        f = open(path, "r")
        len_token = 0
```

```
len_value = 0
token_file_path = path.replace(end_name, end_token_name)
try:
    f_t = open(token_file_path, "r")
    for line in f_t:
        tokens = line.split(" ")
        len_token += len(tokens)
except Exception as e:
    f_b_v.write("=====\n" + str(e) + "\n")
    f_b_v.write(token_file_path + "\n=====\n")
    return
value_len_count = defaultdict(int)
value_hash_count = defaultdict(int)
vec = []
sum = 0
sum_h = 0
for line in f:
    values = re.findall("'.+?'", line)
    len_value += len(values)
    if len_value != len_token:
        s_f_v.write(path + "\n")
        return
    for cou in range(0, len_value):
        value_len_count[tokens[cou]] += len(values[cou])
        value_hash_count[tokens[cou]] += hash(values[cou])
        sum += len(values[cou])
        sum_h += hash(values[cou])
vec.append(sum)
```

Приклад векторизації списку лексем показаний в таблиці 3.

2.4. Кластеризація неоптимального коду

Кластерний аналіз – це метод класифікаційного аналізу; его основное назначение - розбиття багатьох спільнот досліджуваних об'єктів та визнаків на однорідних у певному сенсі групи, або кластери. Це багатомерний статистичний метод, тому передбачається, що вихідні дані можуть бути значущим обсягом, тобто істотно більшим може бути як кількість об'єктів досліджень (спостережень), так і признаків, що характеризують ці об'єкти.

Таблиця 3 - Приклад векторизації списку лексем

№	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

Основна ідея використання кластеризації для виявлення неоптимального коду полягає в тому, щоб вивчити нормальний режим у вже наявних даних (навчити), а потім скористатися цією інформацією, щоб вказати, є одна з точок аномальної чи ні, коли пропонуються нові дані (тест).

Загальні кроки (після загальної попередньої обробки):

1. Вибрати найкращу модель відповідно до вашими данними.

2. Підготувати модель до даних про навчання, цей крок може варіруватись в залежності від рівня залежності від вибраних моделей, на цьому етапі необхідно виконати кілька налаштувань гіперпараметрів.
3. Як тільки нові дані отримані, порівняйте їх з результатами моделей та визначте, що це нормальна точність або неоптимальний код, здатність виконати цю класифікацію сильно залежить від моделей (якщо я домовився про це, як це зробити для рекламних моделей), деякі основані на розстоянії, в той час як інші використовуються ймовірності.
4. Якщо з плином часу відбувається яка-небудь еволюція даних, то через деякий час модель слід переобучити, щоб вивчити нове введення (у протилежному випадку це нове введення завжди може бути класифіковано як аномальне).

Звісно, кластеризація не ідеальна для всіх проблем, пов'язаних з виявленням неоптимального коду (як і будь-якого іншого методу, ви знаєте, вільний ланч), але не об'єднання цієї техніки з виведення інших інтелектуальних функцій може допомогти вам вирішити безліч проблем; наприклад, що відбувається, коли у вас є тимчасовий ряд, і проблема в тому, що значення збільшується занадто швидко, але все ще в звичайному діапазоні значень? Додавання виробничих в алгоритм кластеризації може допомогти вам знайти цю аномалію.

Для полегшення процесу розбору знайдених аномалій розроблений інструмент [8], що дозволяє автоматизовано розділити зібраний набір прикладів вихідного коду на групи. В основі роботи інструмента вибрано метод кластеризації даних k-means, який дозволяє згрупувати екземпляри набору даних в кластери найбільш похожих один на одного елементів

шляхом мінімізації сумарного квадратичного відхилення точок кластерів від центрів цих кластерів.

Для векторизації вихідного коду застосований метод токенізації програми і підрахунок статистики оцінки важливості токенів TF-IDF. Як параметр метод кластеризації приймає кількість кластерів, на які буде вироблено поділ вихідного набору даних. Даний інструмент дозволяє розділити знайдені аномалії на групи близьких за змістом прикладів вихідного коду, що спрощує процес аналізу отриманих аномалій за рахунок скорочення прикладів, які необхідно розглянути для формування опису знайдених аномалій.

Особливо істотна користь даного інструменту при виявленні великої кількості прикладів аномального коду. Нижче представлена частина коду, яка відповідає за кластеризацію даних за допомогою k-means методу.

```
df=pd.read_csv(features_path, header=0, sep='\t', encoding = "ISO-8859-1")

feature_vectors = df.values[:,2:]

paths_to_sources = df.values[:,1:2]
filenames_list = []
for t_t in paths_to_sources:
  for i in t_t:
    filenames_list.append(str(i))

kmeans = KMeans(n_clusters=23).fit(feature_vectors)
labels = kmeans.labels_

k = 0
for i in labels:
    token_filename = filenames_list[k].split('/', 6)[5]
    filename = token_filename.replace(".txt.token", ".kt")
    source_path = filenames_list[k].replace(".txt.token", ".kt")
    folder_name = folder + str(i)
    if not os.path.exists(folder_name):
        os.mkdir(folder_name)
    shutil.copyfile(source_path, folder_name + "/" + filename)
    k += 1
```

ВИСНОВКИ ДО РОЗДІЛУ 2

Реалізовано спосіб виявлення неоптимального коду за допомогою системи, що використовує методи локального рівня викиду, метод однокласових опорних векторів, DBSCAN і HDBSCAN. Такий вибір алгоритмів обумовлений прагненням використовувати в рішенні методи, раніше не вживані в системі виявлення неоптимального коду мовою Kotlin, тому їх використання може дозволити знаходити нові класи неоптимального коду і розширити інструментарій системи пошуку неоптимального коду. Незважаючи на це, метод локального рівня викиду раніше використовувався системою, але його було обрано через порівняно невеликі часові витрати, які необхідні для його роботи і високої якості розпізнавання неоптимального коду.

Для виявлення неоптимального використання коду обрані методи, що базуються на кластеризації. Це методи: однокласовий метод опорних векторів, DBSCAN і HDBSCAN. Кожен з перерахованих алгоритмів запускається на двох наборах даних: один - для списків типів токенів, інший для списків лексем. В методі виявлення неоптимального використання коду було обрано реалізацію алгоритмів, що входить до складу бібліотеки Scikit-learn., що є досить точною.

Для кластеризації даних використано метод кластеризації даних k-means дозволяє згрупувати екземпляри набору даних в кластери найбільш похожих один на одного елементів шляхом мінімізації сумарного квадратичного відхилення точок кластерів від центрів цих кластерів. Для векторизації вихідного коду застосований метод токенізації програми і підрахунок статистики оцінки важливості токенів TF-IDF. Як параметр метод кластеризації приймає кількість кластерів, на які буде вироблено поділ вихідного набору даних.

3. АПРОБАЦІЯ РЕАЛІЗОВАНОЇ СИСТЕМИ

Тестування даної системи буде відбуватись на великому наборі програмного коду. Буде продемонстровано порівння з вже існуючими рішеннями а також представлено результати фахівцям з великим досвідом роботи та їх думку.

3.1 Збір даних

Для проведення апробації першочерговим завданням стає збір набору вихідного коду програм, в якому буде відбуватися виявлення неоптимального коду. За допомогою модуля збірки репозиторіїв системи пошуку неоптимального коду вдалося отримати 96101 репозиторій з ресурсу GitHub. Кожен такий репозиторій містить код на мові програмування Kotlin. Всього було зібрано 1.8 млн файлів з вихідним кодом на Kotlin. На основі кожного файлу був побудований список токенів, який потім був перетворений в два списки: список типів токенів і список лексем. Разом вийшло два набори даних: один складається з 1.8 млн. списків типів токенів, інший з 1.8 млн списків лексем.

3.2 Векторизація получених даних

Перед векторизацией отриманих даних було зібрано список, що складається з різних типів токенів, присутніх в зібраному наборі даних. Всього вийшло 130 різних типів токенів. Найбільш часто вживаними типами токенів виявилися WHITE_SPACE, IDENTIFIER, DOT, LPAR, package.

Потім до кожного з наборів даних був застосований метод векторизації. На основі списку типів токенів вийшов набір векторів

розмірності 5 · n = 5 · 130 = 650. Для списку лексем набір векторів отримав розмірність 7 · n = 7 · 130 = 910.

3.3 Запуск алгоритмів пошуку неоптимального коду

На отриманих наборах векторів були запущені алгоритми виявлення неоптимального коду: локальний рівень викиду, однокласових метод опорних векторів, DBSCAN і HDBSCAN. В результаті декількох раундів експериментів шляхом візуальної оцінки підвибірки потенціального неоптимального коду розміром близько 50 були обрані наступні параметри для запуску алгоритмів.

Локальний рівень викиду: n_neighbors = 20 (кількість сусідів, щодо яких вважається фактор локальних викидів), contamination = 0.0001 (частка даних, оголошується неоптимальним кодом).

Однокласових метод опорних векторів: kernel = " rbf " (тип ядра, який використовується в алгоритмі). В результаті роботи алгоритму була виділена частка аномалій, складова 0.0001 від усього набору даних.

DBSCAN: eps = 0.5 (максимальна відстань між двома об'єктами, які будуть розглядатися як елементи одного кластера).

HDBSCAN: min_cluster_size = 15 (найменша кількість виділених кластерів).

Параметри виділення частки аномальних точок для всіх алгоритмів були обрані таким чином, щоб в результаті роботи алгоритму було виділено не більше 200 екземплярів даних, оголошених аномаліями. Це зроблено через те, що система виявлення неоптимального коду спрямована на виділення недалекого людиною числа аномальних прикладів коду.

3.4 Кластеризація найденого неоптимального коду

За допомогою розробленого інструментарію вдалося зібрати набір з 1237 унікальних прикладів аномального коду. Всі знайдені аномалії були розділені на 23 групи за допомогою інструменту автоматизованої кластеризації вихідного коду. Такий вибір кількості груп обумовлений тим, що експертна оцінка результатів вельми ресурсномісткий процес, тому необхідно отримати досить невелику вибірку найбільш показових прикладів неоптимального коду.

3.5 Складання звіту про найдений неоптимальний код

У процесі формування звіту для розробників ПЗ мовою програмування Kotlin з метою отримання експертної оцінки корисності знайдених аномалій був проведений відбір 47 найбільш показових прикладів серед виявленого неоптимального коду. З кожної групи в залежності від її чисельності було вибрано по 2 - 3 приклади вихідного коду програм. Кожна група прикладів неоптимального коду була супроводжена коротким описом, що характеризує відмінні риси даного класу.

Розглянемо деякі класи неоптимального коду детальніше та наведемо прклади.

Перевірки на null це перевірка посилання за яким можна дістати змінну. У випадку коли повертається null це означає що такого об'єкту не існує. Розберемо на прикладі нижче наведеного некороктного коду.

```
val i = 2
if (i == null) {
    DO ()
}
```

Такі перевірки не потрібно так як значення змінної вже задано тому вона ніколи не буде null. Мова програмування Kotlin є nullsave і у більшості випадках цього не потрібно робити.

Багато перечислень (enum) – це ще один вид некоректного коду, який веде за собою не читабельний код, який не відповідає стандартам. У таких випадках краще записувати дані у файл і потім працювати з ним. Такі дані зберігаються в стеці, який є досі обмежений в пам’яті. Неправильне використання enum класів та даниxх може привести до помилки stackoverflow

Нижче на рисунку 21 наведено приклад некоректного коду

```
// Status code information
enum class StatusCode(val statusCode: Int, val message: String) {
    STATUS_200( statusCode: 200, message: "Success"),
    STATUS_227( statusCode: 227, message: "Incorrect invite code"),
    STATUS_228( statusCode: 228, message: "Invite timestamp finished"),
    STATUS_229( statusCode: 229,  message: "Email address already in use"),
    STATUS_230( statusCode: 230,  message: "Password did not conform with password policy"),
    STATUS_231( statusCode: 231,  message: "User account already exists"),
    STATUS_232( statusCode: 232, message: "Couldn't update email address for user with id"),
    STATUS_233( statusCode: 233, message: "Too many request"),
    STATUS_234( statusCode: 234, message: "User not found exception"),
    STATUS_235( statusCode: 235, message: "Too many attempts with incorrect password"),
    STATUS_236( statusCode: 236, message: "Incorrect username or password"),
    STATUS_237( statusCode: 237,  message: "Incorrect email address"),
    STATUS_238( statusCode: 238, message: "Invalid parameter exception")
}
```

Рисунок 21 – Приклад неоптимального коду enum виразів

Багато умовних виразів також не є хорошим стилем написання коду. У таких випадках рекомендується найти один або декілька об’єктів за даними яких можна опріділити наступний код що буде виконуватись. До

умовних операторів відносяться такі конструкції як: if, if-else, if-else-if, when, а також їх вкладені конструкції.

Інтервали. У більшості середовищах розробки правильно розставити інтервали можна за допомогою «гарячих клавіш». У більшості випадках це не впливає на роботу програми та на процес компіляції, але не можна забувати про конвертацію кода написаного мовою Kotlin у JavaScript та у Java. У такому випадку це може вплинути не тільки на час конвертації, але й може некоректний код видати в результаті.

Використання шістнадцятерічних чисел можуть бути некоректним кодом. Це звязано з тим, що на такі дані виділяється на багато більше пам'яті що веде до неправильного використання ресурсів, особливо в великих кількостях.

Арифметичні вирази також є некоректним кодом. Це пов'язано з тим, що при кожному запуску програми потрібно виконувати однакову операцію. Це веде за собою затрати пам'яті та швидкості. Прикладом може слугувати наступний вираз:

```
val MAX_AGE = 60 * 60 * 24
```

Багато типів аргументів також впливає на роботу компілятора. Це пов'язано з тим, що дані, які потрібні для роботи методу або відокремленої частини коду, що виконується у даний момент часу, зберігаються у стеці, а також посилання за якими можна дістати об'єкти з heap. Ще одним недоліком є не читабельність коду, до якої може призвести такий код. Приклад надлишковість типів аргументів у метода:

```
fun fileGeneration (templateData: List <List <String>>, fieldNameIndex: Int, csvData: CsvRequest, validationRegexIndex: Int, requiredIndex: Int, evaluationIndex: Int): Pair <List <List <IdResolverRequest>>, List <File>> { }
```

Код представлений нижче це один рядок який показує сигнатуру методу. У таких випадках краще розбити метод на декілька або хоча б створити об'єкт, який буде включати всі ці дані для полегшення читабельності.

Динамічні вирази є дуже затратні за пам'яттю так, як на них виділяється пам'ять автоматично і вподальшому збільшує за потребою. У таких виразах недоліком також є те, що при кожній потребі вираз афтоматично буде оброблятись навіть якщо він не змінився.

Багато виразів expect. За кожним виразом expect стоїть перевірка стандартна за заданими параметрами або реалізована власноруч. У випадку коли таких виразів багато можна помістити у спеціальний блок try-catch.

Довгі строки, строкові інтерполірування та багатострокові строки:

```
EMPLOYER_DDIA_PRODUCT_CONTRIBUTORY_RATES("employer_ddia_product_contributory_rates", null),
EMPLOYER_DDIA_PRODUCT_VOLUNTARY_RATES("employer_ddia_product_voluntary_rates", null),
DDIA_DENTAL_RATE_STRUCTURE_AGE_BANDED_CONTRIBUTORY(
   "ddia_dental_rate_structure_age_banded_contributory",
   "EMPLOYER_DDIA_PRODUCT_CONTRIBUTORY_RATES"
),
DDIA_DENTAL_RATE_STRUCTURE_TIERED_CONTRIBUTORY(
   "ddia_dental_rate_structure_tiered_contributory",
   "EMPLOYER_DDIA_PRODUCT_CONTRIBUTORY_RATES"
),
DDIA_DENTAL_RATE_STRUCTURE_AGE_BANDED_VOLUNTARY(
   "ddia_dental_rate_structure_age_banded_voluntary",
   "EMPLOYER_DDIA_PRODUCT_VOLUNTARY_RATES"
),
```

Будь-яка частина коду використовується повторно, змінюється та переписується зазвичай іншим програмістом. Такий код стає меншчитабельним, шо також не відповідає конвенції.

Багато присвоєнь зазичай викликана не правильною побудовою архітектури проекту та малим досвідом розробників зображено на рисунку 22.

```
val lines = csvBuilder.bufferedReader().readLines()
val header = lines[0].split( ...delimiters: "\t")
val fieldNameIndex = header.indexOf("field_name")
val evaluationIndex = header.indexOf("evaluation")
val validationRegexIndex = header.indexOf("validation_regex")
val requiredIndex = header.indexOf("is_required")
val templateData = lines
        .asSequence()
        .drop( n: 1)
        .map { it.split("\t".toRegex()) }
        .toList()
val domain: Domain = Domain.getDomainById(csvData)
```

Рисунок 22 – Приклад багато присвоєнь

У таких випадках краще всього створити певний об’єкт і звертатися до його полів.

Великі масиви данних також призводять до неоптимального використання коду. Вони роблять кот не читабельним, а також ведуть до великих затрат пам’яті, що може призвести до одніє з найбільших помилок після якої програма може перестати працювати: ErrorOutOfMemmory, а при обробці набору даних до ExceptionStackOverFlow.

3.6 Отримання експертних оцінок

Сформований звіт з прикладами аномального коду на мові Kotlin був переданий для оцінки двом незалежним експертам, які є розробниками мови програмування Kotlin. Оцінка проводилася незалежно на підмножині всього набору неоптимального коду, включеного до звіту, за п'ятибальною шкалою. На основі оцінок, виставлених експертами конкретних прикладів аномального коду, була сформована усереднена оцінка для кожного класу знайденого неоптимального коду. Результати експертної оцінки представлені в таблиці 4.

У графі "Опис класу" зазначена коротка характеристика оцінюваного класу неоптимального коду. У графі "T" знаком "+" позначені ті групи неоптимального коду, елементи яких були виявлені в результаті аналізу типів токенів, в графі "L" знаком "+" позначені групи, елементи яких були виявлені в процесі аналізу лексем. Графи "Q1" і "Q2" містять усереднені оцінки відповідних експертів для входять до складу групи неоптимального коду. Знак "-" вказує на те, що жоден екземпляр відповідного класу не потрапив в оцінювану вибірку для відповідного експерта. У графі "Q" представлена середня оцінка для класу неоптимального коду, що розраховується як середнє арифметичне значення оцінок експертів.

За результатами експертної оцінки можна зробити висновок, що досить невелика кількість класів і прикладів аномалій зацікавило експертів, які проводять оцінку звіту з неоптимальним кодом. Лише 2 класу з 23 отримали підсумкові оцінки 4 і 5. Оцінку 3 і вище вдалося отримати 7 з 23 отриманих класів.

Таблиця 4 - Експертна оцінка класів неоптимального коду

№	**Опис класа**	**T**	**L**	**Q_1**	**Q_2**	**Q**
1	Багато виразів expect	+	+	5	–	5
2	Багато перечислень (enum)	+		4	–	4
3	Багато типів аргументів	+		5	2	3.5
4	Багато класів	+		4	3	3.5
5	Арифметичні вирази	+		3	3	3
6	Динамічні вирази		+	3	–	3
7	Довгі строки		+	3	–	3
8	Багато довгих циклів	+		4	1	2.5
9	Функції з властивостями	+		4	1	2.5
10	Великі масиви данних	+		2	3	2.5
11	Строкові інтерполювання	+	+	3	1	2
12	Строкові підстановки		+	3	1	2
13	Багато виразів when	+		3	1	2
14	Багато присвоїнь	+		2	2	2
15	Заміна імен типів		+	2	2	2
16	Багато умовних виразів	+		2	–	2
17	Шістнадцятерічні числа		+	2	–	2
18	Вирази assert	+	+	1	1	1

19	Інтервали	+	+	1	1	1
20	Іменовані аргументи	+		1	1	1
21	Багатострокові строки		+	1	1	1
22	Перевірки на null		+	1	1	1
23	Нестандартне кодування		+	–	1	1

Варто відзначити, що оцінки експертів досить серйозно різняться. якщо розглядати оцінки лише першого експерта, щоб результати становили більш позитивні. 2 клас отримали наивисшую оцінку, 6 з 23 класів отримав оцінки 4 і 5, а 12 класів, щоб отримати більше половини, отримав оцінки 3 і вище. Що стосується оцінок другого експерта, то вони виявилися суттєво нижче оцінок першого, і не більше трьох балів. Якщо узагальнити результати, можна зробити вивід, що аномалії, отримані на основі аналізу типових токенів, виявилися більш цікавими для експертів, що не отримали аномалії, отримані при аналізі лексем. Частина знайдених аномалій отримала досить високі оцінки від експертів.

3.7. Порівняння результатів зі схожими роботами

Знайдено було роботу та досліджено на тему "Дослідження аномалій роботи компілятора мови Kotlin", в рамках якої проводилося дослідження пошуку умовних кодових аномалій на основі байт-коду, що генерується компілятором мови Kotlin для платформи JVM. В результаті був також сформований звіт з набором аномалій, знайдених за допомогою аналізу JVM байт-коду, і переданий експертам для оцінки корисності аномалій. Експертами виступили розробники мови програмування Kotlin, які

оцінювали аномалії на основі аналізу токенів. В результаті роботи був отриманий набір з 18 класів аномалій.

Раніше були отримані оцінки для робіт, спрямованих на пошук неоптимального коду в програмах на мові Kotlin по дереву розбору і байт-коду, засновані на явному витяганні ознак (робота "Детектор аномалій в програмах на мові Kotlin", в результаті якої було виділено 23 класу аномалій) і автокодуванням (робота "Виявлення проблем продуктивності в програмах на мові програмування Kotlin з використанням статичного аналізу коду ", в результаті якої було виявлено 29 класів неоптимального коду) в процесі векторизації даних. Однак, оцінки результатів даних робіт були виставлені іншими експертами, не тими, які оцінювали аномалії, знайдені на основі аналізу токенів і байт-коду.

Порівняльна таблиця результатів досліджень представлена в таблиці 4. Оцінки описуваної роботи представлені в графі Q1. У графі Q2 перераховані оцінки дослідження в рамках пошуку неоптимального коду на основі байт-коду. У графі Q3 вказані оцінки роботи з пошуку неоптимального коду по дереву розбору і байт-коду, засновані на векторизації автокодуванням. У графі Q4 перераховані оцінки роботи виявлення неоптимального коду на основі явного виділення ознак по дереву розбора. Оцінки перераховані для класів, які були отримані в рамках даної роботи.

Таблиця 5 – Порівняння результатів

№	Описання класа	Q_1	Q_2	Q_3	Q_4
1	Багато виразів expect	5	–	–	–
2	Багато перечислень (enum)	4	–	4	–
3	Багато типів аргументів	3.5	–	–	–
4	Багато класів	3.5	–	–	–
5	Арифметичні вирази	3	1	–	–
6	Динамічні вирази	3	–	–	–
7	Довгі строки	3	–	–	–
8	Багато довгих циклів	2.5	2	2	2
9	Функції з властивостями	2.5	–	–	–
10	Великі масиви даних	2.5	–	–	–
11	Строкове інтерполірування	2	–	–	–
12	Строкові підстановки	2	–	–	–
13	Багато виразів when	2	–	–	–
14	Багато присвоєнь	2	–	–	–
15	Заміна імен типів	2	–	–	–
16	Багато умовних виразів	2	–	–	–
17	Шістнадцяткові числа	2	–	–	–
18	Вирази assert	1	–	–	–

19	Інтервали	1	–	–	–
20	Іменовані аргументи	1	–	–	–
21	Багаторядкові строки	1	–	–	–
22	Провірки на null	1	–	–	–
23	Нестандартне кодування	1	–	–	–

Як видно з результатів порівняння, більшість виявлених класів аномалій є новими і не були виделені та досліджені раніше. Серед технічних класів, які вдалось також знайти в результаті інших досліджень, прослідковується вдосконалення експертних оцінок. Це говорить про те, що виявлення неоптимального коду на основі аналізу токенів є успішно застосованим до задачі пошуку неоптимального коду в програмах написаних мовою програмування Kotlin.

ВИСНОВКИ ДО РОЗДІЛУ 3

У розділі проведено апробації даної системи. Всього було зібрано 1.8 млн файлів з вихідним кодом на Kotlin. На основі кожного файлу був побудований список токенів, який потім був перетворений в два списки: список типів токенів і список лексем. Разом вийшло два набори даних: один складається з 1.8 млн. списків типів токенів, інший з 1.8 млн списків лексем.

У розділі було описано класи знайденого неотимального коду шляхом пошуку неоптимального коду за допомогою векторизації типів токенів та лексем.

Було проведено відбір 47 найбільш показових прикладів серед виявлених кодових аномалій. з кожної групи в залежності від її чисельності було вибрано по 2 - 3 приклади вихідного коду програм. Кожна група прикладів неоптимального коду була супроводжена коротким описом, що характеризує відмінні риси даного класу.

У розділі розписано класи неоптимального коду та до яких наслідків призводить такий код. Представлення рішення уникання більшості знайдених аномлій

Проводилась цінка незалежно на підмножині всього набору неоптимального коду, включеного до звіту, за п'ятибальною шкалою. На основі оцінок, виставлених експертами конкретних прикладів аномального коду, була сформована усереднена оцінка для кожного класу знайдених аномалій.

У результаті перевірених експериментів отримано набір нових класів неоптимального коду, що було одним з першочергових завдань дослідження. Аномалії, отримавши оцінки 4 і 5, було надіслано розробниками мовних програм програмування Kotlin при розгляді з метою включення в тести для компілятора мови Kotlin.

ЗАГАЛЬНІ ВИСНОВКИ ПО РОБОТІ.

Під час роботи були отримані наступні результати:

• проаналізовано алгоритми токенізації, векторизації та пошуку неоптимального коду;

• досліджено систему пошуку неоптимального коду від розробників JetBrains Research

• розроблена та реалізована система виявлення кодових аномалій на основі токенів, що включає модуль векторизації токенів, модуль пошуку неоптимального коду та модуль автоматизованої кластеризації виявлених неоптимального коду;

• проведено апробацію розробленої системи наборів даних, що містить 1,8 млн. Файлів із ресурсом GitHub з вихідним кодом на мові Kotlin, в результаті якої отримано набір прикладів аномального коду;

• зібрано та проаналізовані основні знайдені аномалії

• представлено 23 класи аномалій, багато з них було розгянуто вперше в цій роботі.

• порівняно результати з іншими способами пошуку неоптимального коду

• описано знайдені класи неоптимального коду, пояснено наслідки такого коду та варіанти заміни такого коду

На основі отриманих результатів можна зробити висновок про те, що запропонований підхід дозволяє успішно здійснити пошук кодових аномалій на основі списку токенів, генерованих за вихідним коду програм на мові програмування Kotlin. Дана система дозволяє знайти більше некоректного коду за рахунок використаня інших алгоритмів векторизації, токенізації та пошуку неоптимального коду.

У якості розробок подальших досліджень у цій галузі можна задати задачу пошуку неоптимального коду за допомогою аналізу різних структур представлення, наприклад, промежуточного представлення програм для конкретного компілятора (IR), а також досліджень кода на мові Javascript, генеруваного транслятором з мови Kotlin. Також застосування інших підходів до векторизації даних та методів виявлення неоптимального коду може дати новий результат у дослідженні цієї області.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

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. Graphbased 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., MartnezPerez 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. Deanonymizing 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 kmeans 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 // AsiaPacific 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.