

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий інститут прикладного системного аналізу
Кафедра системного проектування**

«На правах рукопису»
УДК _____

До захисту допущено:
Завідувач кафедри
_____ Вадим МУХІН
«__» _____ 2022р.

**Магістерська дисертація
на здобуття ступеня магістра
за освітньо-науковою програмою
“Інтелектуальні сервіс-орієнтовані розподілені обчислювання”
зі спеціальності 122 "Комп'ютерні науки"
на тему: «Виявлення плагіату програмного коду в студентських
роботах»**

Виконала:

студентка II курсу, групи ДА-01мн
Мироненко Єлизавета Андріївна _____

Науковий керівник:

доцент, к.т.н., с.н.с
Кисельов Геннадій Дмитрович _____

Рецензент:

Професор, д.т.н., професор,
Бідюк Петро Іванович _____

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

Студентка _____,

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут прикладного системного аналізу
Кафедра системного проектування

Рівень вищої освіти – другий (магістерський)

Спеціальність – 122 "Комп'ютерні науки"

Освітньо-професійна програма – "Інтелектуальні сервіс-орієнтовані розподілені обчислювання"

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Вадим МУХІН

02 листопада 2021 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Мироненко Єлизаветі Андріївні

1. Тема дисертації «Виявлення плагіату програмного коду в студентських роботах», науковий керівник дисертації Кисельов Геннадій Дмитрович доцент, к.т.н.,с.н.с, затверджені наказом по університету від _____ р. № _____

2. Термін подання студентом дисертації – 10 червня 2022 р.

3. Об'єкт дослідження – система для перевірки наявності запозичень в програмному коді

4. Предмет дослідження – методи та підходи виявлення плагіату програмного коду

5. Перелік завдань, які потрібно розробити

1. Провести загальний аналіз предметної області – дослідити поняття плагіату, види плагіату, проаналізувати сучасні дослідження щодо частоти випадків академічного шахрайства серед студентів.

2. Дослідити існуючі інструменти для перевірки програмного коду на подібність – виділити їх переваги та недоліки.
 3. Провести порівняльний аналіз існуючих методів перевірки на плагіат.
 4. Зпроектувати та реалізувати власну систему для перевірки програмного коду на наявність запозичень.
 5. Аналіз отриманих результатів.
6. Орієнтовний перелік графічного (ілюстративного) матеріалу - презентація
7. Орієнтовний перелік публікацій
8. Дата видачі завдання – 04 квітня 2022р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Отримання завдання	04.04.2022	
2	Збір інформації, дослідження існуючих публікацій	11.04.2022	
3	Розробка плану дослідження	18.04.2022	
4	Дослідження існуючих методів пошуку плагіату програмного коду	20.04.2022	
5	Аналіз результатів	03.05.2022	
6	Проектування системи пошуку плагіату програмного коду	04.05.2022	
7	Програмна реалізація системи пошуку запозичень програмного коду	11.05.2022	
8	Оформлення дипломної роботи	28.05.2022	
9	Отримання допуску до захисту та подача роботи в ДЕК	10.06.2022	

Студентка

Є.А. Мироненко

Науковий керівник

Г.Д. Кисельов

РЕФЕРАТ НА МАГІСТЕРСЬКУ ДИСЕРТАЦІЮ

виконану на тему: «Виявлення плагіату програмного коду в студентських роботах»

студенткою Мироненко Єлизаветою Андріївною

Актуальність. З розвитком інформаційних технологій та мережі Інтернет проблема порушення принципів академічної доброчесності набула досить великих масштабів, окрім того, ситуація значно погіршилась з вимушеним переходом на дистанційне навчання. Ситуацію додатково погіршує ще й те, що різні медіа ресурси пропонують послуги рерайтингу, копірайтингу та ін. Через це студенти можуть і не до кінця усвідомлювати поняття плагіату та не розуміти, що вони займаються академічним шахрайством. Згідно зі статистики на 2020 рік, кількість плагіату в роботах студентів, що навчаються в університетах за напрямком «Комп'ютерні науки», складала до 80%[32]. Ці статистичні дані, безсумнівно, підкреслюють що проблема академічної недоброчесності на сьогодні є дуже серйозною та роботи студентів необхідно перевіряти більш ретельно з використанням ефективних методів для перевірки на плагіат.

Мета і задачі дослідження. Метою даного дослідження є розробка ефективного методу перевірки програмного коду на плагіат. Основними завданнями даної роботи є дослідження існуючих статей, публікацій та робіт на тему виявлення плагіату програмного коду, практична перевірка та проведення порівняльного аналізу існуючих рішень з метою подальшого проектування та реалізації більш ефективного та стійкого до вразливостей методу перевірки програмного коду на наявність запозичень.

Об'єкт дослідження. Технології та алгоритми, які використовуються для вирішення задачі виявлення випадків плагіату в сирцевому програмному коді.

Предмет дослідження. Підходи та алгоритми, що дозволяють розробити ефективну систему для перевірки студентських робіт на плагіат, яка буде захищена від такої вразливості як порушення хешування.

Наукова новизна отриманих результатів. Полягає в розробці ефективного алгоритму перевірки програмного коду на плагіат з використанням дерев абстрактного синтаксису та нейронних мереж, що у комбінації надаватиме менше можливостей для атаки колізією хешування.

Практичне значення отриманих результатів. Імплементация розробленого алгоритму може бути використана у вищих навчальних закладах з метою перевірки завдань з програмування на наявність запозичень, що дозволить більш ефективно перевіряти роботи, економити час викладача на перевірку, отримати реальний зріз знань студентів та в перспективі підвищити залишковий рівень знань студента з певних дисциплін.

Загальний обсяг роботи – 100 сторінок, 42 рисунки, 24 таблиці, 35 посилань.

Ключові слова: PLP, Transformers, MOSS, Mossad, виявлення плагіату програмного коду, AST, Code2Vec, TreeBERT.

ABSTRACT FOR MASTER'S DISSERTATION

**performed on the topic: "Student's Assignments Code Infringement
Detection"**

by student Yelyzaveta Myronenko

Relevance of the topic theme. With constant development of information technology and the Internet, the problem of violating the principles of academic integrity has become rather serious. In addition, the situation has significantly deteriorated with the forced transition to distance learning. The situation is further aggravated by the fact that various media resources offer rewriting, copywriting and other services. As a result, students may not be fully aware of the concept of plagiarism and may not understand that they their actions are nothing else than an academic fraud. According to statistics for 2020, the amount of plagiarism in the work of students that are studying at universities in the "Computer Science" field was almost 80%[32]. These statistics undoubtedly emphasize that the problem of academic dishonesty is very serious today and that students' work needs to be inspected more effectively using effective plagiarism testing methods.

The purpose and objectives of the work. The aim of this study is to develop an effective method for checking software code for plagiarism. The main objectives of this work are: studying of existing articles, publications, works related to the detection of plagiarism in software code, practical testing and comparative analysis of existing solutions in order to design and implement a more efficient and stable method for source code plagiarism detection.

Object of study. Technologies and algorithms used to solve the problem of detecting cases of plagiarism in program code.

Subject of study. Approaches and algorithms that allow to develop an effective system for testing student work for plagiarism, which will be protected from such vulnerability as hashing violation.

Scientific novelty of the obtained results is to develop an effective source code plagiarism detection algorithm, which is protected from hashing violation problem, using abstract syntax trees and neural networks.

The practical significance of the obtained results. The implementation of the developed algorithm can be used in universities to check programming assignments for plagiarism, which will help to inspect students' works more effectively, save teacher's time that is usually spent on work checks, get a real understanding about students' knowledge and increase the residual level of student knowledge in certain disciplines.

The total volume of work is 100 pages, 42 pictures, 24 tables, 35 links.

Keywords: PLP, Transformers, MOSS, Mossad, source code plagiarism detection, AST, Code2Vec, TreeBERT.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ	10
ВСТУП	11
1 ПОНЯТТЯ ПЛАГІАТУ ПРОГРАМНОГО КОДУ, ПРОБЛЕМА НАЯВНОСТІ ПЛАГІАТУ В СТУДЕНТСЬКИХ РОБОТАХ.....	13
1.1 Визначення поняття «плагіат» та види плагіату	13
1.2 Проблема плагіату в студентських роботах	15
1.3 Класифікація типів плагіату вихідного програмного коду	17
1.3.1 Класифікація за Беллоном	18
1.3.2 Класифікація за Роєм.....	19
1.3.3 Класифікація за Зельцером.....	20
1.4 Методи пошуку фрагментів плагіату програмного коду	20
1.4.1 Строкові алгоритми	21
1.4.2 Алгоритми з використанням лексичного аналізу	21
1.4.3 Алгоритми з застосуванням синтаксичних дерев	21
1.4.4 Алгоритми на основі графів залежностей.....	22
1.4.5 Алгоритми на основі метрик	22
1.4.6 Комбіновані підходи	23
1.5 Огляд існуючих сервісів для перевірки програмного коду на плагіат	23
1.5.1 MOSS.....	24
1.5.2 Codequiry	25
1.5.3 JPlag.....	26
1.5.4 Copyleaks	26
1.6 Висновки до розділу	27
2 АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ ПЕРЕВІРКИ КОДУ НА ПОДІБНІСТЬ	29
2.1 Порівняльний аналіз різних систем для перевірки коду на подібність	29
2.2 Загальний огляд MOSS.....	35

2.3 Розгляд принципів роботи MOSS	37
2.4. Огляд Mossad.....	40
2.5 Висновки до розділу	43
3 ПРОЕКТУВАННЯ АВТОРСЬКОЇ СИСТЕМИ ПЕРЕВІРКИ	
ПРОГРАМНОГО КОДУ НА ПЛАГІАТ	45
3.1. Запропонована блок-схема системи	46
3.2 Дерева абстрактного синтаксису.....	48
3.3 Code2Vec.....	53
3.4 Transformers	58
3.5 Вибір методу для перевірки на наявність запозичень у програмному кодi.....	63
3.6 Проектування системи	69
3.7 Приклад роботи методу для виявлення наявності плагіату	72
3.8. Висновки до розділу	75
4 РОЗРОБКА СТАРТАП ПРОЕКТУ	77
4.1 Ідея проекту	77
4.2 Технологічний аудит ідеї проекту	78
4.3 Аналіз ринкових можливостей.....	79
4.4 Розробка ринкової стратегії проекту	84
4.5 Розробка маркетингової програми.....	86
4.6 Висновки до розділу	89
ВИСНОВКИ	91
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	95

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

NLP – Natural Language Processing (обробка природної мови)

MOSS – Measure of Software Similarity

AST – Abstract Syntax Tree (дерево абстрактного синтаксису)

ML – Machine Learning (машинне навчання)

Code2Vec – Code to Vector

Code2Seq – Code to Sequence

Seq2seq – Sequence-to-sequence

FNN – Feedforward neural network (нейронна мережа з прямим зв'язком)

PLP – Program Language Processing

ВСТУП

Безсумнівно, плагіат існував вже давно, але свого небаченого рівня поширення академічне шахрайство досягло саме з розвитком та популяризацією мережі Інтернет та суміжних інформаційних технологій. Цей факт не є якимось раптовим «відкриттям», оскільки Інтернет робить широко доступними мільярди статей, книжок, джерел коду, фотографій, звуків і відео, а це означає, що їх можна просто скопіювати та поділитися. Окрім того, ще однією причиною розмиття терміну «плагіат» є досить часте використання різними сучасними медіа таких термінів як рейтинг, репостинг, копіпастинг та ін.

Проблема сучасного плагіаризму виникає через плеяду факторів, основними з яких є :

- Не повне розуміння того, що являється плагіатом;
- Незацікавленість завданням;
- Погане планування часу та/або його відсутність;
- Віра, що викладач не помітить та не знайде плагіат в роботі;
- Бажання отримати гарну оцінку, приклавши для цього мінімум зусиль.

На даний момент, автоматичне виявлення плагіату та пошук вихідних документів розробляється і досліджується в якості можливого стримуючого фактору. Звісно, люди вміють виявляти плагіат в своїх областях знань, наприклад, автор коду з легкістю впізнає свою роботу, але необхідні значні зусилля та час щоб дізнатися про всі можливі джерела, які необхідно перевірити та представити переконливі докази проти студента, що порушив правила академічної доброчесності. Ручний аналіз тексту на плагіат стає непрактичним у великих масштабах, що привертає значну увагу до автоматичного виявлення плагіату.

Метою даної роботи є розробка ефективного методу пошуку плагіату програмного коду, який можна було би застосувати під час перевірки завдань або лабораторних робіт студентів з програмування.

Дана тема є актуальною, бо приблизно 70% студентів хоч раз під час свого навчання копіювали роботу свого колеги та подавали її на перевірку як свою [30]. Окрім того, станом на 2017 рік, згідно з дослідженням Transparency International середній рівень академічного шахрайства в Україні становить приблизно 35% [11]. В результаті, через досить часту практику клонування чужих робіт, випускник вищого навчального закладу має значно нижчий середній показник знань, що в подальшому впливає і на репутацію вищого навчального закладу, а також може фінансово наражати потенційного роботодавця такого випускника збитками.

1 ПОНЯТТЯ ПЛАГІАТУ ПРОГРАМНОГО КОДУ, ПРОБЛЕМА НАЯВНОСТІ ПЛАГІАТУ В СТУДЕНТСЬКИХ РОБОТАХ

1.1 Визначення поняття «плагіат» та види плагіату

Плагіат визначається як представлення оригінальної роботи іншого автора або ідей як власних, шляхом включення їх у свою роботу без повного визначення, тобто, цитування та посилання на джерела [1]. Даного визначення стосуються як всі опубліковані, так і неопубліковані матеріали, причому неважливо у якому форматі – електронному, друкованому чи рукописному.

Словник Меріам-Вебстер [2] надає наступні значення слову «плагіат»:

- Презентація ідей, слів інших авторів як своїх власних;
- Використання чужих робіт без цитування;
- Крадіжка робіт інших людей;
- Представлення ідеї як нової та оригінальної, повністю або частково використовуючи матеріали з існуючих джерел.

В загальному, в сфері інформаційних технологій плагіату поділяється на дві великі групи: плагіат тексту та програмного коду. В свою чергу, плагіат тексту теж поділяється на дві групи (інтелектуальний та буквальный), а також три великі підтипи можна виокремити і у плагіаті програмного коду. Детальну загальну класифікацію видів плагіату представлено на рисунку 1.1.

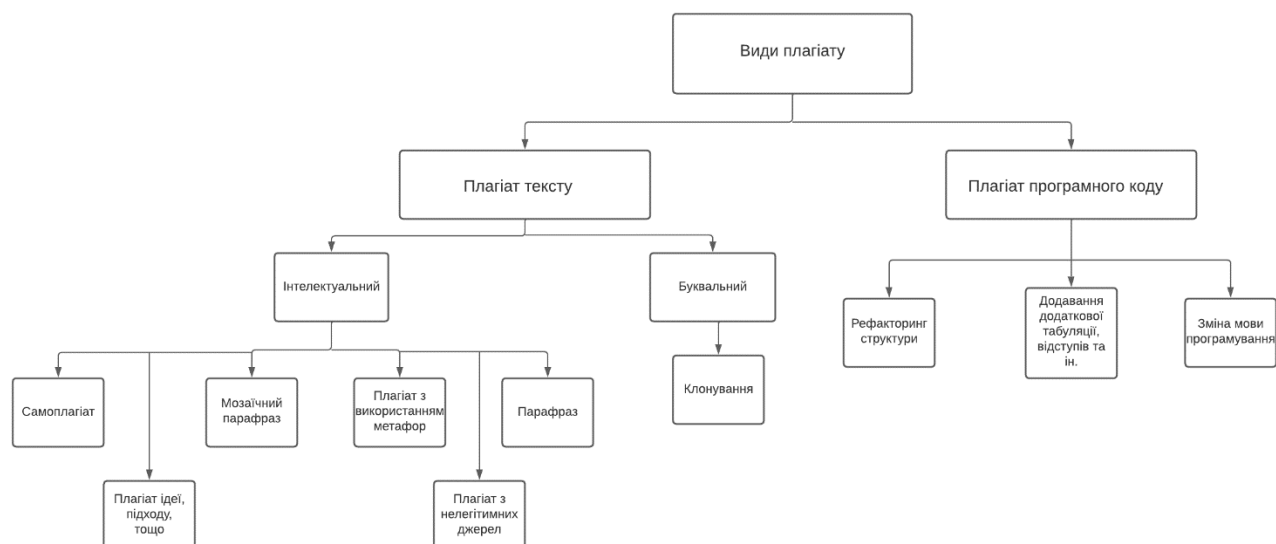


Рисунок 1.1 - Загальна класифікація видів плагіату

Тепер дамо коротку загальну характеристику кожному зазначеному на схемі типу плагіату.

Почнемо з плагіату тексту, а саме буквального плагіату. До цього типу плагіату відносяться наступні категорії:

- Клонування - видання скопійованої в інших людей без внесення будь-яких змін роботи за власну без цитувань та зазначення оригінального джерела.

Далі розглянемо підвиди інтелектуального плагіату:

- Парафраз - копіювання чужої роботи та представлення її у новий спосіб, тобто, шляхом зміни побудови речення, зміни порядку слів та підбору синонімів без зазначення оригінального джерела.
- Мозаїчний парафраз - поєднання декількох робіт інших дослідників та змінення їх представлення за допомогою, наприклад, використання іншого граматичного стилю.

- Плагіат з використанням метафор - для більш кращої та чіткої передачі ідей інших науковців плагіатор використовує метафори.
- Плагіат ідеї, підходу до задачі або рішення - в даному виді плагіату ідея, підхід до вирішення деякої задачі або її вирішення беруться з інших документів та в результаті видаються за власну.
- Самоплагіат - у своїй роботі автор включає попередньо опубліковані матеріали статей.
- Плагіат з нелегітимних джерел - автор у дослідницькій роботі виносить посилання які не є автентичними.

Тепер розглянемо види, що належать до плагіату програмного коду:

- Рефакторинг структури - розробник змінює порядок функцій та/або змінює синтаксис без зазначення оригінального джерела програми.
- Додання додаткової табуляції, відступів та коментарів та подальше видання роботи за власну.
- Зміна мови програмування - програміст переписує код іншою мовою програмування без позначення авторів оригінального програмного коду.

Варто зазначити, що вся подальша робота буде стосуватися виключно плагіату програмного коду та його пошук в роботах студентів.

1.2 Проблема плагіату в студентських роботах

Поява плагіату в роботах студентів, на жаль, на сьогодні є досить поширеним явищем. Окрім того, дуже часто студенти не вважають що копіювання роботи іншої людини – неетична практика та є прямим порушенням правил академічної доброчесності. Наприклад, скопіювати статтю з Інтернету здається студенту швидше, ніж розбиратися в задачі

самостійно, особливо коли студент вже завантажений великою кількістю завдань та лабораторних робіт. Також ситуацію додатково погіршує факт сприйняття студентами застосування тактик порушення доброчесності через призму оптимізації власного часу як стандартну та розповсюджену практику – саме це є одним з найбільш вагомих факторів порушення правил академічної доброчесності у середовищі студентства. Тобто, студент усвідомлює той факт, що деяка кількість його одногрупників вдається до нечесних академічних практик та отримує позитивний результат (у широкому сенсі – високі академічні бали, при низьких витратах власного часу). Такий приклад поведінки, що пропагує плагіаризм як доволі ефективний метод для подолання академічного модуля заохочує в свою чергу студента скористатися цією ж тактикою за для отримання високої оцінки та зменшення часу, який необхідний для якісного вивчення матеріалу[11,31]. В результаті, ефект від сприйняття поведінки інших перевершує за величиною ефекти індивідуального ставлення до списування та до ймовірності можливого покарання. Але, згідно з дослідженнями [7], [8] та [9], студенти схильні переоцінювати кількість випадків копіювання серед інших студентів, що і є фундаментом для раціоналізації свого власного звернення до зазначених практик. Також варто відзначити ще одне дослідження з теми академічної недоброчесності в якій досліджувалась думка студентів та було проведено соціологічне опитування для викладачів - статтю «Dishonesty in Academics and Business: A Cross-Cultural Evaluation of Student Attitudes»[10].

Згідно робіт [8] та [9] можна виділити наступні факти:

- Близько 85% опитуваних ставляться нейтрально до плагіату робіт та списувань на іспитах;
- 70% першокурсників підтримують м'яке покарання або його відсутність;

- Приблизно 52% студентів виступають за усне покарання;
- Один з десяти студентів підтримує досить жорсткі міри боротьби з плагіатом: недопуск до іспиту, незадовільна оцінка тощо.

Також виділимо ключові статистичні дані з опитування [10]:

- Майже всі викладачі (більше 97%) проявляють досить «спокійне» ставлення до випадків плагіату;
- Більше ніж 60% викладачів обмежуються усним попередженням як реакція на списування та плагіат;
- Лише 5% викладачів дозволяють користуватися конспектами, підручниками тощо на контрольних та іспитах (дана методика вважається однією з найбільш ефективних попередження та зниження кількості випадків академічної недоброчесності).

В якості підсумку, можна зробити висновок що важливою складовою, необхідною для зменшення кількості випадків плагіату в студентських роботах, є більш жорстка реакція викладачів на ці випадки. У випадку більш ретельної перевірки роботи на плагіат та більш жорсткої реакції на академічне шахрайство з боку викладачів студенти менш схильні вдаватися до списування, що значно покращить середні показники залишкових знань серед студентів після вивчення курсу та допоможуть сформувати дисципліну серед учнів.

1.3 Класифікація типів плагіату вихідного програмного коду

Різні автори дещо по-різному класифікують види плагіату програмного коду. Розглянемо деякі з них, такі як класифікації за Беллоном, Роєм та Зельцером [29]. Варто зазначити, що всі описані нижче класифікації не враховують випадки форматування вихідного коду.

1.3.1 Класифікація за Беллоном

Згідно з Беллоном, можна виділити три типи плагіату програмного коду.

- Тип 1. Ідентичні фрагменти програмного коду.

```

1
2 //оригінальний метод
3 public int firstMethod(int firstVar, int secondVar) {
4     if(firstVar==0) {
5         return secondVar;
6     }
7     return firstMethod(secondVar, firstVar/secondVar);
8 }
9
10 //плагіат оригінального методу
11 public int firstMethod(int firstVar, int secondVar) {
12     if (firstVar == 0)
13         return secondVar;
14     return firstMethod(secondVar, firstVar / secondVar);
15 }

```

Рисунок 1.2 - Приклад ідентичних фрагментів коду

- Тип 2. Частини коду типу 1, які містять зміну назв, типів, значень констант та змінних.

```

//оригінальний метод
public int firstMethod(int firstVar, int secondVar) {
    if(firstVar==0) {
        return secondVar;
    }
    return firstMethod(secondVar, firstVar/secondVar);
}

//плагіат оригінального методу
public long firstMethod(long var1, long var2) {
    if (var1 == 0L)
        return var2;
    return firstMethod(var2, var1 / var2);
}

```

Рисунок 1.3 - Приклад 2-го типу плагіату програмного коду

- Тип 3. Фрагменти коду типу 2 з деякими модифікаціями (наприклад, додавання або видалення деяких операторів).

```

//оригінальний метод
✓ public int firstMethod(int firstVar, int secondVar) {
✓     if(firstVar == 0) {
        return secondVar;
    }
    //доданий оператор
✓     if (secondVar == 0) {
        return firstVar;
    }
    return firstMethod(secondVar, firstVar/secondVar);
}

//плагіат оригінального методу
✓ public long firstMethod(long var1, long var2) {
✓     if (var1 == 0L)
        return var2;
    return firstMethod(var2, var1 / var2);
}

```

Рисунок 1.4 - Приклад 3-го типу плагіату

1.3.2 Класифікація за Роєм

В загальному, дана класифікація ґрунтується на описаних вище типах класифікації Беллона. Для типів 1-3 класифікації за Беллоном вводиться узагальнюючий надтип «синтаксичної близькості». Окрім цього, згідно Рою, існує четвертий вид – семантичний тип клонів. Отже, дана класифікація виглядає наступним чином:

- Синтаксично близькі фрагменти:
 - типи 1-3 за класифікацією Беллона.
 - семантично еквівалентний код, представлений за допомогою різних синтаксичних конструкцій.

```

//оригінальний метод
public int firstMethod(int firstVar, int secondVar) {
    if(firstVar==0) {
        return secondVar;
    }
    return firstMethod(secondVar, firstVar/secondVar);
}

//плагіат оригінального методу
public int firstMethod(int firstVar, int secondVar) {
    while (secondVar != 0) {
        swap(firstVar, secondVar);
        secondVar /= firstVar;
    }

    return firstVar;
}

```

Рисунок 1.5 - Приклад семантично еквівалентного коду, але представленого за допомогою різних синтаксичних конструкцій

1.3.3 Класифікація за Зельцером

За цією класифікацією виділяються наступні види клонування програмного коду:

- Точні клони – еквівалент першого типу описаних вище класифікацій;
- Клони 2-го типу – змінні та/або константи в тілі дублікату можуть бути замінені на вирази, що обчислюються;
- Клони 3-го типу – окремі вирази у тілі клону можуть бути замінені на інші вирази.

1.4 Методи пошуку фрагментів плагіату програмного коду

Станом на сьогодні, можна виділити декілька груп методів пошуку клонів програмного коду:

- Строкові алгоритми;
- Алгоритми з застосуванням лексичного аналізу;

- Алгоритми з застосуванням синтаксичних дерев;
- Алгоритми на основі графів залежностей;
- Алгоритми на основі метрик;
- Гібридні підходи.

Розглянемо детальніше зазначені групи методів.

1.4.1 Строкові алгоритми

Основна концепція даної групи алгоритмів доволі проста: з фрагменту коду, що аналізується, виключаються зайві пробіли та коментарі (дана задача досить легко вирішується за допомогою використання регулярних виразів), після чого використовуються алгоритми хешування або суфіксного дерева для пошуку повторюваних підрядків певної довжини. Зазначені алгоритми можуть знайти лише повністю ідентичні фрагменти коду (клони типу 1, згідно з класифікацією видів плагіату програмного коду, яка була описана раніше), тому на практиці широкого використання не мають[29]. Але, в якості переваги даного підходу можна підкреслити його незалежність від мови аналізованої програми.

1.4.2 Алгоритми з використанням лексичного аналізу

Алгоритми пошуку плагіату на основі лексичного аналізу працюють схоже з рядковими, але в них програмний код перед пошуком клонів перетворюється на послідовність токенів з урахуванням мови програмування; при цьому токени, що відповідають за різні ідентифікатори і константи, можуть різнитися (для пошуку клонів типу I), так і збігатися (для пошуку клонів типу II). Відомими прикладами аналізаторів дублювання коду, що працюють за такими принципами, є Dup та CCFinder; також існує утиліта JPlag [16], завданням якої є пошук запозичень між програмами мовою Java.

1.4.3 Алгоритми з застосуванням синтаксичних дерев

Існує також підхід до пошуку клонів програмного коду за допомогою побудови абстрактних синтаксичних дерев (AST) та пошуку в них подібних

піддерев. Дана група алгоритмів корисна при пошуку дублювання коду, але може виявитися недостатньо ефективним при пошуку навмисних запозичень у разі їх змін, наприклад, при перестановці рядків або операндів місцями. Перевагою підходу можна назвати його незалежність від мови програмування на етапі роботи з AST, що може дозволити аналізувати взаємну ідентичність програм різними мовами.

1.4.4 Алгоритми на основі графів залежностей

Підходи на основі статичного аналізу (графів потоку управління та потоку даних) та профілювання надають складний у реалізації, але потенційно потужний інструмент для аналізу дублювання коду. Інструменти, що використовують профілювання, можуть бути досить складними у використанні в промислових системах з причин складності самого програмного забезпечення та необхідності введення деяких даних для його роботи при профілюванні; використання ж таких інструментів при автоматичній перевірці рішень задач з програмування значною мірою спрощується через використання готових вхідних даних для перевірки коректності рішень - таким чином, при інтеграції в систему перевірки рішень аналізатора унікальності коду, організаторам змагальних та контрольних заходів не потрібно додавати будь-які додаткові вхідні дані для перевірки рішень на унікальність [29]. Необхідно відзначити, що використання графів залежностей для аналізу запозичень безпосередньо може дати велику кількість помилкових спрацьовувань на простих та середніх за складністю реалізації алгоритмах, особливо класичних та відомих. Аналогічно підходу з AST на етапі аналізу графів не залежить від мови програмування.

1.4.5 Алгоритми на основі метрик

Також деякі дослідники застосовували завдання пошуку дублювання програмного коду шляхом підрахунку різних метрик вихідного коду: кількості рядків, кількості викликів функцій, кількості рівнів вкладеності

блоків, кількості аргументів функції, що використовуються, і/або локальних змінних, і т.д.

1.4.6 Комбіновані підходи

Розглянуті вище підходи часто комбінуються, багато алгоритмів поєднують у собі етапи лексичного аналізу чи побудови графової моделі та підрахунку метрик. Одним із відносно нових підходів до фінальної обробки отриманих описаними вище алгоритмами даних є застосування нейронних мереж: блоки вихідного коду оброблялися за допомогою нейронної мережі з метою пошуку повторюваних шаблонів; також в роботі [\[34\]](#) було застосовано навчання нейронної мережі розпізнавання автора коду за його «стилем програмування» та подальша перевірка авторства програмного коду цією нейронною мережею; але такий підхід може бути слабо застосовним до класичних олімпіад з програмування та контролю завдань з коротких навчальних курсів, при контролі яких у організатора швидше за все не буде достатньої кількості даних для навчання нейромережі.

1.5 Огляд існуючих сервісів для перевірки програмного коду на плагіат

Перед тим, як розглядати існуючі сервіси, варто зазначити, що процедура перевірки програмного коду на плагіат відрізняється від перевірки тексту. Згідно з вже розглянутими класифікаціями схожості програмного коду, можна виділити наступні критерії «схожості»:

- Ідентичні фрагменти програмного коду;
- Зміна назв, типів значень, змінних та констант;
- Додавання та видалення окремих операторів;
- Семантично еквівалентний код—представлення за допомогою різних семантичних конструкцій.

В загальному, на сьогодні можна виділити декілька популярних сервісів перевірки програмного коду на плагіат, такі як:

- 1 MOSS;
- 2 Codequiry;
- 3 JPlag;
- 4 Copyleaks.

Розглянемо кожен з них.

1.5.1 MOSS

MOSS (Measure of Software Similarity) – це система, що була розроблена в 1994 році для визначення ступеню плагіату програмного коду. Основним застосуванням даного інструменту на даний момент є виявлення плагіату під час вирішення задач на заняттях з програмування. Алгоритм за яким працює дана система є доволі ефективними в порівнянні з іншими алгоритмами та методами визначення порушення академічної доброчесності [21].

Проте слід зауважити, що MOSS слід вважати лише фреймворком (у широкому розумінні цього терміну) для перевірки вихідного коду на плагіат, оскільки одним із обмежуючих факторів є те, що система без модифікацій може виділити лише подібність двох фрагментів програмного коду, а рішення чи даний випадок є плагіатом приймає безпосередньо перевіряючий. Тим не менш, використання цього інструменту допомагає значно зменшити навантаження на викладачів, шляхом застосування напівавтоматичної перевірки, бо система лишень виділяє фрагменти програмного коду, на які необхідно звернути увагу. Після того як кортеж документів та файлів надсилається на перевірку, MOSS створює колекцію HTML-сторінок, що демонструють частини файлів, в яких виявлено збіги.

На сьогодні MOSS підтримує такі мови програмування як:

- C, C#, C++;
- Java;
- Python;
- JavaScript;
- Basic;
- FORTRAN;
- ML;
- Haskell;
- Lisp;
- Scheme;
- Pascal;
- Modula2;
- Ada.

Отже, як вже було зазначено, приймати рішення виключно за індикативними знаходженнями MOSS є неправильним, тому що демонструються лише фрагменти програмного коду, які потребують подальшого аналізу та можуть бути “false positive”.

Умовами користування сервісом є реєстрація та використання поточного ідентифікатора користувача. Великою перевагою даної системи є безкоштовні реєстрація та користування.

1.5.2 Codequiry

Перевагою системи Codequiry, на відміну від інших програм перевірки програмного коду на плагіат, є те, що даний інструмент не ігнорує такі деталі як коментарі та перевірку назв змінних. Codequiry дозволяє відносно швидко ідентифікувати будь-яку повторювану інформацію у рамках задачі аналізу тексту коду [18]. Але, варто зазначити, що функціонал перевірки програмного на плагіат коду у вищезазначеній системі є платним.

Основними особливостями Codequiry є:

- 1 Глибокий аналіз коду, в тому числі, коментарів та змінних;
- 2 Бібліотека системи налічує понад 20 мільярдів рядків загальнодоступного коду для порівняння;
- 3 Проста у використанні;
- 4 Підтримка більшості мов програмування;
- 5 Звіт у вигляді кругової діаграми джерел, що були використані в коді.

1.5.3 JPlag

JPlag - популярна система для перевірки вихідного коду на плагіат. В якості переваг цього інструменту можна виділити те, що його використання безкоштовне, але все одно користувачу попередньо необхідно пройти процедуру реєстрації.

Цю систему розробив Guido Malpohl у 1996 році. Згідно [16], основними особливостями системи є :

- 1 Підтримка таких мов програмування як Java, C, C++, C#, тощо;
- 2 Дозволяє порівняння документів, що знаходяться в бібліотеці, але не проводить пошук співпадінь в Інтернеті;
- 3 Надання результатів в HTML-форматі;
- 4 Побудова гістограми подібності.

1.5.4 Copyleaks

Copyleaks на момент написання цього звіту є одним з популярних сервісів перевірки програмного коду на плагіат. Основними особливостями даної системи є:

- 1 Простота використання;
- 2 Відносно висока точність;
- 3 Підтримка більше ніж 20 різних мов програмування;
- 4 Можливість роботи з різними форматами файлів;

5 Створення звіту подібності для кожного окремого запиту перевірки.

Але, на жаль, детального аналізу даного сервісу не було проведено через високу плату за кожну перевірку, відсутня безоплатна демо версія.

1.6 Висновки до розділу

В даному розділі було введено загальне поняття плагіату, описано його види. Значна увага була приділена класифікації видів плагіату програмного коду, адже це є безпосередньою темою даного дослідження. Також було розглянуто існуючі групи методів, що використовуються для пошуку випадків академічного шахрайства у програмному коді в роботі студента. Окрім цього, було підкреслено важливість та актуальність теми роботи шляхом наведення існуючих досліджень щодо відношення студентів та викладачів до випадків плагіату.

Отже, в якості підсумку можна виділити наступні ключові моменти даного розділу:

Плагіат є дуже великою проблемою у сучасному академічному світі, що визначається як процес використання чужих ідей та робіт без цитування, або ж факт видачі цих робіт, як свої. Можна виділити два великих семантичних підвиди плагіату: плагіат тексту та плагіат коду.

Існує декілька видів плагіату програмного коду, основі з них: рефакторинг, додавання додаткової табуляції, та переніс на іншу мову програмування, без посилання на оригінал. Використання цих прийомів окремо, чи у комбінації винесено як наступні класифікації: за Беллоном, Роєм та Зельцером.

Основними групами методів пошуку випадків академічного шахрайства в програмному коді є:

- Строкові алгоритми;
- Алгоритми з використанням лексичного аналізу;
- Методи на основі графів залежностей;
- Алгоритми на основі метрик;
- Гібридні підходи.

На сьогодні, існують декілька вже розроблених інструментів, які здатні виявляти запозичення у програмному коді, що можуть бути класифіковані як плагіат, за критеріями, що були описані у цьому розділі. Найбільш поширеними за використанням є MOSS, JPlag, Codequiry та Copyleaks, що підтримують найпоширеніші мови програмування та мають достатньо велику кількість користувачів, через що вони були обрані для подальшого конкретного аналізу.

2 АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ ПЕРЕВІРКИ КОДУ НА ПОДІБНІСТЬ

В даному розділі буде детально розглянуто один з найбільш поширених інструментів для перевірки програмного коду на подібність, а саме Measure of Software Similarity (MOSS) та подібні йому за функцією системи з відкритим та закритим сирцевим кодом.

2.1 Порівняльний аналіз різних систем для перевірки коду на подібність

В цьому пункті буде проведено порівняльний аналіз існуючих систем для перевірки знаходження співпадінь в програмному коді.

Для аналізу було обрано декілька популярних систем – MOSS, JPlag, Codequiry та Plaggie. Обрані системи було проаналізовано за такими технічними можливостями та функціями: проста та зручність використання, можливість виключення шаблонного коду, масштабованість, демонстрація результатів, можливість використання у якості веб-сервісу, наявність коду системи у відкритому доступі, кількість доступних мов програмування. Опишемо більш детально характеристики за якими проводився аналіз.

1. **Простота та зручність використання** – дана характеристика важлива, адже користувачу повинно бути легко користуватися інструментом та викладач не повинен витратити велику кількість часу на його налаштування.

2. **Можливість виключення шаблонного коду** – можливість виключити з перевірки певні шаблони коду, які з великою ймовірністю будуть однаковими, наприклад, є якась частина програмного коду за завданням яка може бути однаковою в усіх студентів.

3. **Масштабованість** – чи можливо налаштувати систему таким чином, щоб її можна було використати і для інших мов програмування.

4. **Демонстрація результатів** – надзвичайно важливий критерій порівняння, адже якісна презентація результатів перевірку викладачу значно скоротить час, який може бути необхідний для додаткової перевірки. Якісний звіт про перевірку має відповідати наступним критеріям:

- У звіті підкреслено фрагменти коду, що співпадають;
- Можливий одночасний порівняльний перегляд файлів, що перевірялися;
- Звіт містить інформацію про кількість рядків, в яких було виявлено збіг, кількість параметрів системи, що були активні на момент перевірки, графік результатів перевірки.

5. **Можливість використання у якості веб-сервісу** – чи можливо використовувати систему в якості веб-сервісу.

6. **Наявність коду системи у відкритому доступі** – чи є система так званою opensource системою, тобто, чи є можливість для користувача завантажити собі код системи.

7. **Кількість доступних мов програмування** – скільки мов програмування підтримує система для перевірки програмного коду на плагіат.

Для зручності результати аналізу було зведено в таблицю 2.1.

№ п/п	Характеристика порівняння	MOSS	JPlag	Codequiry	Plaggie
1	Зручність використання	Так	Так	Так (перевірено безкоштовний функціонал)	Ні
2	Можливість виключення шаблонного коду	Так	Так	Так	Так
3	Масштабованість	Ні	Ні	Ні	Ні
4	Демонстрація результатів	Так, відповідає всім вимогам	Так, відповідає всім вимогам	Так (перевірено безкоштовний функціонал)	Так, відповідає всім вимогам
5	Можливість використання у якості веб-сервісу	Так	Так	Ні	Так
6	Наявність коду системи у відкритому доступі	Відсутній, Присутні декілька проектів, що написані за алгоритмом MOSS	Відсутній	Відсутній	Наявний
7	Кількість доступних мов програмування	23	6	Більше 20	1

Таблиця 2.1. Результати порівняльного аналізу систем

З таблиці видно, що за технічними характеристиками MOSS є найкращою альтернативою – окрім того, що у даної системи всі характеристики порівняння на високому рівні, MOSS підтримує найбільшу кількість мов програмування. На жаль, не вдалося повністю протестувати систему Codequiry, бо її користування можливо лише за наявності платної (доволі коштовної) підписки, на даний момент пробні безкоштовні версії відсутні.

Тепер порівняємо дві з найпоширеніших системи для пошуку запозичень, а саме MOSS та JPlag, за такою важливою характеристикою як точність перевірки.

Для проведення перевірки було використано фрагменти програмного коду, які відповідають чотирьом типам згідно класифікації подібності програмного коду. Далі наведемо приклади, які порівнювалися під час аналізу.

Перевірка 1 – ідентичний програмний код:

```
rs > Yelyzaveta_Onyshchen > Desktop > example-1.py
def binary_search(arr, low, high, x):
    if high >= low:
        mid = (high + low) // 2

        if arr[mid] == x:
            return mid

        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)

        else:
            return binary_search(arr, mid + 1, high, x)

    else:
        return -1
```

```
rs > Yelyzaveta_Onyshchen > Desktop > example-1.1.py
def binary_search(arr, low, high, x):
    if high >= low:
        mid = (high + low) // 2

        if arr[mid] == x:
            return mid

        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)

        else:
            return binary_search(arr, mid + 1, high, x)

    else:
        return -1
```

Рисунок 2.1.1 Ідентичні фрагменти програмного коду

Як можна побачити з рисунку 2.1.1, перший та другий фрагмент коду абсолютно ідентичні.

Перевірка 2 – зміна назв, типів значень, змінних та/або констант:

```

C: > Users > Yelyzaveta_Onyshchen > Desktop > example-1.py
1 def binary_search(arr, low, high, x):
2     if high >= low:
3         mid = (high + low) // 2
4
5         if arr[mid] == x:
6             return mid
7
8         elif arr[mid] > x:
9             return binary_search(arr, low, mid - 1, x)
10
11        else:
12            return binary_search(arr, mid + 1, high, x)
13
14    else:
15        return -1

C: > Users > Yelyzaveta_Onyshchen > Desktop > example-2.py
1 def bs(array, first, last, x):
2     if last >= first:
3         middle_element = (last + first) // 2
4
5         if array[middle_element] == x:
6             return middle_element
7
8         elif array[middle_element] > x:
9             return bs(array, first, middle_element - 1, x)
10
11        else:
12            return bs(array, middle_element + 1, last, x)
13
14    else:
15        return -1

```

Рисунок 2.1.2. Фрагменти програмного коду, які демонструють другий тип подібності

З наведеного прикладу видно, що програмні коди відрізняються лише назвами змінних.

Перевірка 3 – додавання та видалення окремих операторів:

```

1 int gcd(int l, int r) {
2
3     if(l==0) return r;
4
5     if(r==0) return l;
6
7     return gcd(r, l%r);
8 }

1 int gcd(long a, long b)
2 {
3     if (b == 0L) { return a;
4     }
5     return gcd(b, a % b);
}

```

Рисунок 2.1.3. Фрагменти коду, які демонструють третій тип подібності

Як можна побачити з рисунку 2.1.3, наведені приклади програмного коду відрізняються тим, що у другому фрагменті було видалено оператор, також у фрагменті справа було змінено назви змінних та їх типи.

Перевірка 4 – семантично еквівалентний код:

На рисунку 2.1.4 наведено два фрагменти програмного коду, які є семантично еквівалентними. Обидва фрагмента демонструють функцію бінарного пошуку.

```

C: > Users > Yelyzaveta_Onyshchen > Desktop > example-3.py
1 def binarySearch(array, x, low, high):
2
3     while low <= high:
4
5         mid = low + (high - low)//2
6
7         if array[mid] == x:
8             return mid
9
10        elif array[mid] < x:
11            low = mid + 1
12
13        else:
14            high = mid - 1
15
16    return -1

```

```

C: > Users > Yelyzaveta_Onyshchen > Desktop > example-1.py
1 def binary_search(arr, low, high, x):
2     if high >= low:
3         mid = (high + low) // 2
4
5         if arr[mid] == x:
6             return mid
7
8         elif arr[mid] > x:
9             return binary_search(arr, low, mid - 1, x)
10
11        else:
12            return binary_search(arr, mid + 1, high, x)
13
14    else:
15        return -1

```

Рисунок 2.1.4. Приклад семантично еквівалентного коду

Загальні результати порівняння зведені у таблицю 2.1.

Таблиця 2.1. Загальні результати порівняння інструментів MOSS та JPlag за точністю

Тип подібності коду	MOSS	JPlag
Ідентичні фрагменти коду	100 %	100 %
Зміна назв змінних, їх типів	100 %	100 %
Додавання або видалення окремих операторів	92 %	90 %
Семантично еквівалентний код	64 %	55 %

Як видно з результатів порівняльного аналізу, під час аналізу фрагментів програмного коду першого та другого типів обидві системи виявили 100 % подібність. Що стосується третього та четвертого типу схожості, то система MOSS виявилася більш ефективною та точною. Але, варто зазначити, що не зважаючи на більш низький показник точності, результати отримані після перевірки системою JPlag все ще містять досить

високий показник подібності та такі результати є приводом для більш детальної перевірки.

2.2 Загальний огляд MOSS

На сьогодні, система MOSS є одним з найбільш поширених та популярних для кількісного визначення наявності запозичень в програмному коді, адже алгоритм роботи системи показав себе більш ефективно в порівнянні з іншими аналогічними інструментами для перевірки на плагіат. Але, як і у всіх інших випадках використання таких систем, викладач, який проводить перевірку, безпосередньо приймає рішення про те, чи є робота, що аналізується клонуванням та порушенням правил академічної доброчесності, тому що система MOSS може продемонструвати лише схожість певних частин програмного коду.

Серед основних переваг даного інструменту можна виділити [21]:

1. Підтримка більш ніж двадцяти сучасних мов програмування.
2. Система показує лише співпадіння у великих фрагментах коду, завдяки чому схожість індивідуальних токенів (наприклад, типів змінних) опускається.
3. Можливість знайти плагіат без прив'язки до його розташування у файлі з кодом.
4. Ігнорування коментарів та пробілів.
5. Перевірка, яку надає MOSS стійка до спроб обфускації вхідного програмного коду.
6. Широке застосування (велике ком'юніті)

Для проведення перевірки необхідно зареєструватися та завантажити файли для перевірки. Коли система закінчує перевірку, то надає посилання, за яким можна перейти для перегляду результатів у браузері. Варто зазначити, що результати перевірки будуть доступні

протягом двох тижнів з моменту проведення аналізу. На рисунку 2.2 можна побачити приклад відповіді системи після перевірки.

Moss Results

Mon Dec 28 15:54:31 PST 2020

Options -l java -m 10

Midterm Exam

[[How to Read the Results](#) | [Tips](#) | [FAQ](#) | [Contact](#) | [Submission Scripts](#) | [Credits](#)]

File 1	File 2	Lines Matched
./Midterm Exam/I.java (96%)	./Midterm Exam/I.java (96%)	23
./Midterm Exam/G.java (96%)	./Midterm Exam/I.java (96%)	23
./Midterm Exam/G.java (96%)	./Midterm Exam/I.java (96%)	23
./Midterm Exam/F.java (96%)	./Midterm Exam/I.java (96%)	26
./Midterm Exam/F.java (96%)	./Midterm Exam/I.java (96%)	26
./Midterm Exam/F.java (96%)	./Midterm Exam/G.java (96%)	26
./Midterm Exam/B.java (89%)	./Midterm Exam/I.java (92%)	31
./Midterm Exam/B.java (89%)	./Midterm Exam/I.java (92%)	31
./Midterm Exam/B.java (89%)	./Midterm Exam/G.java (92%)	31
./Midterm Exam/B.java (89%)	./Midterm Exam/E.java (92%)	31
./Midterm Exam/H.java (97%)	./Midterm Exam/L.java (97%)	31
./Midterm Exam/C.java (93%)	./Midterm Exam/D.java (93%)	24
./Midterm Exam/D.java (60%)	./Midterm Exam/E.java (53%)	32
./Midterm Exam/C.java (60%)	./Midterm Exam/E.java (53%)	32
./Midterm Exam/E.java (43%)	./Midterm Exam/L.java (42%)	30
./Midterm Exam/E.java (43%)	./Midterm Exam/H.java (42%)	30
./Midterm Exam/D.java (48%)	./Midterm Exam/L.java (42%)	23
./Midterm Exam/D.java (48%)	./Midterm Exam/H.java (42%)	13
./Midterm Exam/C.java (48%)	./Midterm Exam/L.java (42%)	23
./Midterm Exam/C.java (48%)	./Midterm Exam/H.java (42%)	17
./Midterm Exam/A.java (46%)	./Midterm Exam/L.java (34%)	20
./Midterm Exam/A.java (46%)	./Midterm Exam/H.java (34%)	9
./Midterm Exam/A.java (46%)	./Midterm Exam/E.java (35%)	26
./Midterm Exam/A.java (46%)	./Midterm Exam/D.java (40%)	10
./Midterm Exam/A.java (46%)	./Midterm Exam/C.java (40%)	14
./Midterm Exam/D.java (29%)	./Midterm Exam/K.java (28%)	6
./Midterm Exam/C.java (29%)	./Midterm Exam/K.java (28%)	6
./Midterm Exam/K.java (26%)	./Midterm Exam/L.java (23%)	2
./Midterm Exam/H.java (23%)	./Midterm Exam/K.java (26%)	2
./Midterm Exam/E.java (24%)	./Midterm Exam/K.java (26%)	5

Any errors encountered during this query are listed below.

Рисунок 2.2 - Приклад надання результатів системою MOSS

Також є можливість більш детального перегляду результатів, яка продемонстрована на рисунку 2.3.

/Applications/MOSS/Midterm_Exam/submissions/F.java (96%)		/Applications/MOSS/Midterm_Exam/submissions/J.java (96%)	
2-27		3-24	

<pre> /Applications/MOSS/Midterm_Exam/submissions/F.java import java.util.Scanner; public class MultiplicationTable { public static void main(String[] args) { // the Scanner class to read the user's input. Scanner input = new Scanner(System.in); // the while loop to read user's input int num = 0; while (num <= 1 num >= 12){ System.out.println("Please enter a number between 1 and 12;"); num = input.nextInt(); // its not in the range, it return to top if (num > 1 num < 12){ continue; } // if it's in the range, it will stop and print bottom else if (num <= 1 num >= 12){ break; } } // for loop to produce and display the multiplication table. System.out.println("The multiplication table of " + num + "is"); for (int i = 0; i <13; i++) { System.out.println(num + " x " + i + " = " + num * i); } } } </pre>	<pre> /Applications/MOSS/Midterm_Exam/submissions/J.java import java.util.Scanner; public class MultiplicationTable { public static void main(String[] args) { Scanner input = new Scanner(System.in); int multi = 0; while (multi <= 1 multi >= 12){ System.out.println("Please enter a number between 1 and 12;"); multi = input.nextInt(); if (multi > 1 multi < 12){ continue; } else if (multi <= 1 multi >= 12){ break; } } System.out.println("The multiplication table of " + multi + "is"); for (int i = 0; i <13; i++) { System.out.println(multi + " x " + i + " = " + multi * i); } } } </pre>
--	--

Рисунок 2.3 - Приклад надання результатів системою MOSS

На рисунку 2.3 можна побачити що MOSS оцінив що надані на перевірку файли схожі на 96% та вказав номери строк, де виявлено співпадіння. Якщо звернути увагу безпосередньо на виділений детектором плагіату код, то можна побачити, що логіка повністю співпадає та система це виявила, не дивлячись на те, що в одному з файлів була спроба додати коментарі та змінити назви змінних.

2.3 Розгляд принципів роботи MOSS

В даному підпункті буде розглянуто послідовність дій яка виконується підготовки програмного коду до перевірки.

Згідно з роботи [20], першим етапом MOSS проводить процедуру нормалізації програмного коду. Це означає, що видаляються всі коментарі та пробіли, що не мають семантичного значення. Також на цьому етапі проходить «боротьба» з таким типом плагіату як перейменування змінних в коді – надання всім змінним однакового значення.

На наступному етапі відбувається підготовка вхідних даних до оцінки подібності. Дану процедуру можна поділити на два етапи:

1. Поділення на n-грами;
2. Створення fingerprints.

Опишемо ці два етапи більш детально.

На етапі поділення на n-грами, вже нормалізований програмний код, поділяється на послідовність n-грам, що перетинаються. В даному випадку n – поріг шуму, що надзвичайно важливий, адже значення цього параметру напряду впливає на точність розпізнавання співпадінь – система не зможе знаходити співпадіння в підстроках довжина яких менше за n . Приклад цього перетворення можна побачити на рисунках 2.3 та 2.4.

```
int hello = 0;
return hello;
```

Рисунок 2.3. Приклад вихідного коду

```
TYP_INT ID EQ NUM SEMI
RET ID SEMI
```

Рисунок 2.4 - Приклад результатів нормалізації коду з рисунку 2.3.

Наступним кроком з отриманих n-грам генеруються хеші. Таким чином, вхідні дані представляються у вигляді послідовності хешів. Приклад можна побачити на рисунку 2.5. Візьмемо до уваги, що $n = 3$ в цьому прикладі.

```
30 15 56 83 71 10
```

Рисунок 2.5 - Приклад представлення нормалізованих вхідних даних у вигляді послідовності хешів

Після цього йде етап створення fingerprints за допомогою алгоритму «winnowing»[20,21], за яким в результаті обираються набори хешів для того щоб презентувати всі вхідні дані. Розглянемо принцип роботи даного алгоритму на нижчезазначеному прикладі.

Нехай є певний текст (див. рисунок 2.6) та було проведено процедуру нормалізації, що була описана вище, та в результаті було отримано послідовність хешів (див.рисунок 2.7).

A do run run run, a do run run

Рисунок 2.6. Приклад вхідного тексту

77 74 42 17 98 50 17 98 8 88 67 39 77 74 42
17 98

Рисунок 2.7 - Нормалізований вхідний текст, репрезентований у вигляді хешів

Отже, маємо послідовність хешів $h_1 \dots h_n$, якщо $n > t - k$, то хоча б одне значення з h_i має бути обрано для гарантії виділення всіх співпадінь хоча б довжини t . Розглянемо наступний підхід для імплементації вищеописаної припущення:

1. Нехай розмір вікна дорівнює $w = t - k + 1$
2. В послідовності хешів $h_1 \dots h_n$. кожна позиція $1 \leq i \leq n - w + 1$ визначає вікно хешів $h_1 \dots h_{i+w-1}$.
3. Обирається один хеш з кожного вікна для того щоб створити fingerprint. Необхідно обирати хеш з мінімальним значенням. У випадку, якщо у вікні декілька мінімальних значень, то береться до уваги значення, що максимально праворуч.

Приклад створення вікон довжиною 4 наведено на рисунку 2.8.

(77, 74, 42, 17)	(74, 42, 17, 98)
(42, 17, 98, 50)	(17, 98, 50, 17)
(98, 50, 17, 98)	(50, 17, 98, 8)
(17, 98, 8, 88)	(98, 8, 88, 67)
(8, 88, 67, 39)	(88, 67, 39 , 77)
(67, 39, 77, 74)	(39, 77, 74, 42)
(77, 74, 42, 17)	(74, 42, 17, 98)

Рисунок 2.8 - Приклад вікон хешів

Обрані fingerprints з рисунку 2.8 наведено на рисунку 2.9.

17 17 8 39 17

Рисунок 2.9 - Обрані fingerprints

Тепер, після пояснення роботи winnowing алгоритму, повернемося до системи MOSS. Після виділення fingerprints, система відстежує позицію кожного fingerprint, таким чином що програмний код зі збігами можна потім отримати для звітності. Також варто додати, що користувач завдяки цьому може вказати певні шаблони, які варто ігнорувати у випадку знаходження співпадінь.

2.4. Огляд Mossad

Окрім великої кількості переваг, система для виявлення плагіату програмного коду MOSS має вразливість, яку було доведено у одній з робіт Массачусетського університету – «Mossad: Defeating Software Plagiarism Detection» [22]. В статті описується створення системи Mossad, яка здатна згенерувати більш ніж 12 варіантів семантично-еквівалентного коду та MOSS не буде рахувати згенеровані варіації за плагіат.

Згідно з авторами [22], основна ідея даної системи побудована на використанні такого недоліку MOSS як гіпотеза порушення хешування. Тобто, за цією гіпотезою, якщо у вікно хешів довжини n додати токен це призведе до неспроможності системи виявити плагіат.

Продemonструємо цей недолік на прикладі. На рисунках 2.10 – 2.12 зображено приклад нормалізації вхідного коду з додаванням рядка, що не має семантичного значення.

```
int hello = 0;
bool nothing = true;
return hello;
```

Рисунок 2.10 - Приклад вхідного коду

```
TYP_INT ID EQ NUM SEMI
TYP_BOOL ID EQ BOOL SEMI
RET ID SEMI
```

Рисунок 2.11 - Нормалізація фрагменту коду

```
30 15 56 12 45 39 97 62 80 71 9
```

Рисунок 2.12 - Хеші токенів, зображених на рисунку 2.12.

Якщо ми порівняємо результати з рисунків 2.12 та 2.5, то побачимо що, наприклад, на рисунку 2.12 значення 83 не має через додавання семантично неважливого рядка. Отже, саме цю вразливість і використовує система Mossad для зменшення відсотку співпадіння фрагментів програмного коду. Також, варто зазначити, що Mossad завжди генерує такі фрагменти коду, які є непідозрілими, тобто відсоток схожості буде менше 25%.

В загальному, Mossad працює наступним чином:

Початковим кроком роботи цієї системи є компіляція. Програмний код компілюється в проміжну форму, створюється копія програмного коду та створюється пул (pool), що містить в собі кожну строку коду, що необхідно для подальшої генерації можливих варіантів програмного коду, що здатні «обійти» перевірку.

Етап генерації складається з двох етапів – вибору та внесення змін. На кроці вибору обирається рядок коду з пулу та проводяться прості семантичні перевірки, наприклад, перевірка на кінець рядка та що рядок не впливає на семантику. Після того обраний рядок проходить процедуру внесення змін. Під час проходження цього етапу проводяться перевірки на те, що оригінальна семантика коду була збережена. Сама суть процедури полягає в тому, що обраний рядок буде у випадковому порядку додано до тимчасового варіанту програмного коду та буде проведена низка перевірок. У випадку незбереження початкової семантики програмного коду, варіант буде анульовано та процедура почнеться спочатку з етапу вибору.

Після етапу генерації проходить перевірка поточних результатів MOSS. Іншими словами, Mossad запускає перевірку MOSS для оригінального та генерованого файлу і якщо результат менше за 25% то вважається, що завдання виконано, якщо ні – отримана варіація коду додається до пулу та попередні два етапи повторюються доти, поки не буде досягнуто необхідний результат.

Наведемо приклад застосування Mossad до програмного коду.

На рисунку 2.13 наведено оригінал коду.

```
int NchooseK(int n, int k){  
    if(k==0)return 1;  
    if(n==k)return 1;  
    else  
        return NchooseK(n-1, k-1) +  
               NchooseK(n-1, k);  
}
```

Рисунок 2.13 - Оригінал програмного коду

На рисунку 2.14 наведено приклад коду після модифікацій, внесених системою Mossad.

```

int NchooseK(int n, int k){
    if(k==0) return 1;
    int NchooseK(int n, int k);
    if(n==k) return 1;
    else
        return NchooseK(n-1, k-1) +
               NchooseK(n-1, k);
    int i = 0;
}

```

Рисунок 2.14 - Варіант коду, отриманий за допомогою Mossad

2.5 Висновки до розділу

В даному розділі було проведено порівняльний аналіз найбільш популярних систем аналізу програмного коду на наявність запозичень, а саме MOSS, JPlag, Codequiry та Plaggie. У випадку Codequiry, на жаль, детальний аналіз провести не вдалось, адже робота з цією системою можлива лише за наявності платної підписки, trial версія на даний момент відсутня. Що стосується інших систем – аналіз було успішно проведено та найкращою альтернативою було обрано MOSS, оскільки, MOSS є одним з лідерів за використанням у галузі, та de facto стандартом для багатьох вузів для перевірки сирцевого коду на плагіат. MOSS виділяє серед конкурентів факто того, що система підтримує 23 мови програмування, та є дуже швидкодієюю та точною системою. Додатковою перевагою є чітко описаний в академічних статтях алгоритм та принципи роботи.

MOSS має досить велику кількість переваг серед яких:

- Підтримка великої кількості мов програмування;
- Співпадіння виявляються лише у фрагментах коду певної довжини;
- Система знаходить запозичення навіть при їх переміщенні.

- Всі коментарі та пробіли, що не мають семантичного значення, ігноруються;
- Стійкість до обфускації.

В загальному, всю процедуру роботи MOSS можна розділити на наступні етапи: нормалізації, генерування хешів, створення fingerprints (відбитків) за допомогою алгоритму «winnowing».

Проте система MOSS має значний недолік, який було продемонстровано у роботі [22]. Згідно з авторів, MOSS вразлива до гіпотези порушення хешування - система не здатна знайти плагіат, якщо у вікно хешів додати $1+n$ токенів. Саме цей принцип і покладено в основу системи MOSSAD. Дана система здатна генерувати щонайменше 12 варіантів програмного коду, який подається на вхід системи, ці варіанти будуть семантично еквівалентні до оригіналу та MOSS не виявить плагіату при порівнянні оригіналу та одного із згенерованих системою представлень коду. Окрім того, варто додати, що якщо порівняти всі згенеровані системою пропозиції представлення оригінального коду, то максимальний відсоток схожості буде дорівнювати 25%. Даний відсоток вважається за поріг так званих «непідозрілих» результатів.

Отже можна зробити висновок, що система MOSS вразлива до видів атак, що застосовують генетичні алгоритми модифікації коду, як то MOSSAD. Через що виникає інженерна необхідність розробити та реалізувати підхід, що після його апробації та порівняння точності виявлення плагіату сирцевого коду можна запропонувати як ефективну та більш нову альтернативу існуючим методам, що і буде розглядатись у наступному розділі.

3 ПРОЕКТУВАННЯ АВТОРСЬКОЇ СИСТЕМИ ПЕРЕВІРКИ ПРОГРАМНОГО КОДУ НА ПЛАГІАТ

В даному розділі буде описано рішення, що пропонується та проектування відповідної системи.

В загальному, застосування нейронних мереж для обробки текстів (NLP) докорінно змінило ландшафт досліджень у цій темі, що абсолютно не дивно, враховуючи швидкість та точність у деяких задачах, які надають такі види мереж, особливо багат шарові прецептрони та нейронні мережі на основі механізму уваги. Цілком природньо, що після того, як людство почало успішно застосовувати такі види нейронних мереж для вирішення деяких задач Natural Language Programming, а саме сумаризації та переказу, знаходження подібності та генерації тексту почалися експерименти із застосування технології до сфери програмного коду. Враховуючи, що з рештою мови програмування засновані, здебільшого на семантичних принципах «натуральних мов» (а саме у переважній більшості англійської), було встановлено що задачі обробки програмного коду (Programming Language Processing, PLP) доволі добре вирішуються за допомогою застосування до сирцевого коду високорівневих мов (у даній роботі не будуть розглядатися езотеричні мови). Типовими задачами PLP, що не дивно, є такі самі, що і для NLP, проте саме в контексті PLP додатково виділяється окремий вид задач, як то документація коду.

Принципи застосування нейромереж трансформерів для вирішення PLP задач, здебільшого є таким самим, як і для NLP – застосування bag of words та Masked Language Model (застосування техніки маскуванню слова у послідовності, за для встановлення контекстних зв'язків між синтаксичними одиницями у цій послідовності), проте наївне застосування такої техніки для PLP обходить стороною синтаксичну структуру коду, що є структурованою та несе у собі саму суть масиву тексту, що обробляється.

За для того, щоб не губити таке важливе джерело інформації було адаптовано до використання у сучасних нейромережах техніку застосування Abstract Synthax Tree(AST) [13]. Звісно, що використання AST для репрезентації строгої структуризації коду та зв'язків від батьківських вузлів (node) до дітей (child node) має як свої переваги, так і недоліки, особливо враховуючи, що при лінеаризації (векторизації, що необхідна для того, щоб нейронні мережі могли обробити бажаний фрагмент коду) буду відбуватись суттєве ускладнення задачі (в контексті обробки), наприклад, для трансформерів збільшення кількості токенів має наслідки у вигляді чотирьократного збільшення складності при збільшенні довжини уривку (sequence) [14]. Так само як і застосування LSTM (tree-LSTM) не є панацеєю, оскільки для цього типу мереж необхідно буде пройти все дерево для того.

Проте не зважаючи на overhead, вирішення таких типів задач все ж є дуже цікавим та необхідним у сучасному світі, оскільки таке застосування технологій значно пришвидшує темпи розробки програмного продукту, шляхом допомоги розробнику, наприклад тією самою автоматизованою документацією, вирішенням деяких NL-2-PL (Neural Language to Programming Language) задач, як то пошук функцій за текстовим запитом натуральною мовою та деяких інших. Серед сучасних досліджень на цю тему можна виділити наступні технології: Code5, CodeBert, TreeBert, Code2Vec, Code2Seq, CuBERT. Більшість з цих досліджень буде тим чи іншим чином застосовувати AST задля репрезентації структури, а у випадку Code2Vec та Code2Seq дослідження зосереджені на створення векторних моделей репрезентації термів мов програмування.

3.1. Запропонована блок-схема системи

Ідея полягає в тому, щоб відмовитись, від хоч і доволі швидкого, але вразливого до атаки колізією хешів алгоритму MOSS на користь більш

сучасних та менш вразливих алгоритмів нейронних мереж. Для того щоб реалізувати можливість алгоритмів нейронних мереж обробити та порівняти фрагменти оригінального та підозрілого фрагменти сирцевого коду необхідно векторизувати обидва фрагменти. Враховуючи що будь-який програмний код є формалізованим та у будь-якій мові програмування (навіть езотеричних) присутня формалізована структура та стилістика написання, існує можливість перетворювати код на вектори за допомогою різноманітних технік bag of words (наприклад, Word2Vec) не є цілком раціональним, оскільки може бути втрачена семантика та зв'язність коду. Задля цього можна і необхідно для початку підготувати код (попередня обробка, або ж preprocessing) – прибрати несуттєві для компілятора елементи тексту з коду, «узагальнити» назви елементів коду, після чого можна або одразу перейти до формування AST за допомогою різноманітних модулів у різних мовах або ж у випадку складнощів – для початку скомпілювати код, а потім перейти до етапу генерування AST. Після отримання AST рекомендується використовувати різноманітні техніки (у нашому випадку – Code2Vec), що надають змогу трансформувати AST у векторний вимір – “bag of vectors”. Після цього також можна провести оптимізацію у вигляді зниження кількості вимірів для вирішення проблеми “curse of dimensionality”, що значно оптимізує ці вхідні дані перед їх подачею на наступний крок. Цим наступним кроком є нейронна мережа виду Transformer з вже налаштованими параметрами та набором ваг. Після перевірки користувачеві буде запропоновано переглянути звіт про перевірку, а самі фрагменти коду можуть бути опрацьовані таким чином, щоб мати колекцію шинглів які можуть бути записані до індексованої бази даних та використані при наступних перевірках. Запропонована блок-схема алгоритму наведена на рисунку нижче.

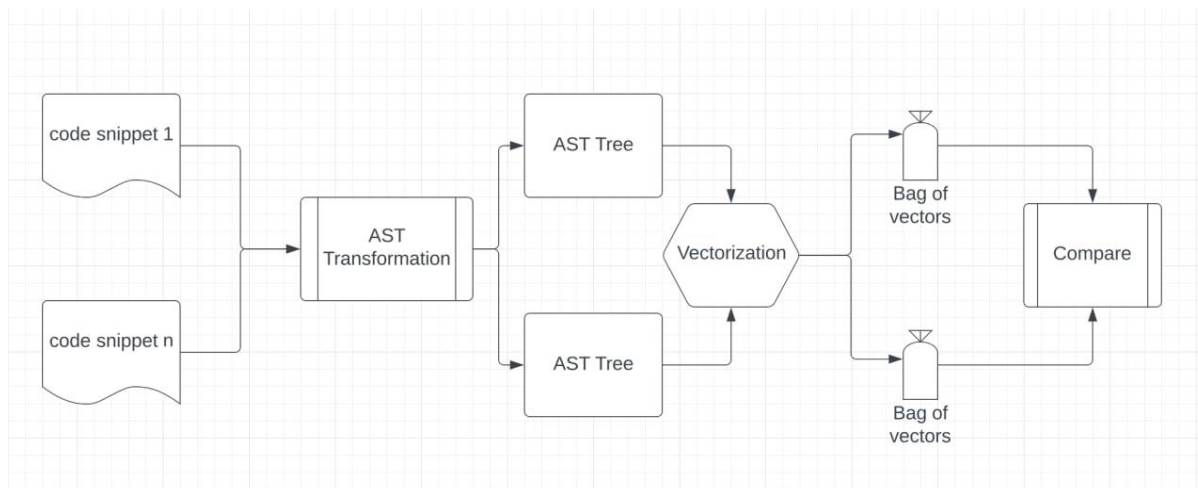


Рисунок 3.1- блок-схема запропонованого алгоритму

Перед тим як перейти до проектування системи введемо декілька теоретичних понять та пояснимо їх.

3.2 Древа абстрактного синтаксису

Древа абстрактного синтаксису (Abstract Syntax Tree, AST) – це представлення абстрактної синтаксичної структури програми.

В основному AST використовується у компіляторах, оскільки дана структура даних дуже добре підходить для представлення структури програмного коду. Зазвичай, дерево абстрактного синтаксису є результатом етапу аналізу синтаксису компілятором [13]. Використання саме дерев абстрактного синтаксису обумовлено тим, що деякі конструкції неможливо репрезентувати за допомогою незалежної від контексту граматики (в якості прикладу можна навести неявну типізацію). AST використовується в якості проміжного результату на декількох етапах роботи компілятора і, як результат, ця структура даних має великий вплив на кінцевий результат роботи компілятора [15]. Наприклад, якщо у згенерованому дереві наявні помилки, компілятор виведе повідомлення про помилку. Отже, якщо в загальному описати роботу компілятора, то виконуються такі кроки:

- Лексичний аналіз;

- Синтаксичний аналіз;
- Генерація коду.

Загальна схема процесу компіляції наведена на рисунку.

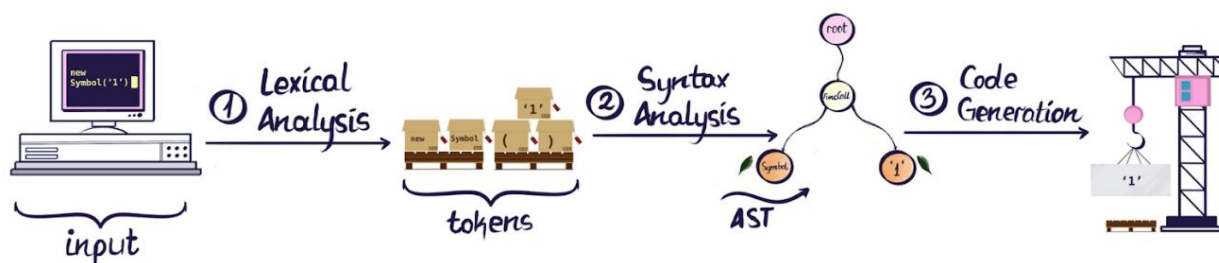


Рисунок 3.2 - Загальна схема процесу компіляції коду

Як можна побачити з рисунку 3.2, то дерева абстрактного синтаксису необхідні для коректного виконання кроку генерації коду.

Окрім використання в компіляторах, AST можуть бути застосовані до широкого спектру задач, які поділяються на 3 групи [14]:

- Читання;
- Зміна;
- Вивід на екран/друк.

Варто зазначити, що дані задачі є адитивними: тобто, якщо є необхідність друку дерева абстрактного синтаксису в рамках певного програмного коду, то висока ймовірність, що воно було створене та змінене в цьому ж самому коді.

Також необхідність генерування AST з програмного коду можна пояснити ще на одному простому прикладі:

Нехай існує деякий код на мові Java і є задача автоматично перетворювати даний код на мову C++. Написати код, який би «читав» код на Java та автоматично в ту саму мить перетворював його на C++

доволі складна задача. Окрім того, якщо синтаксис Java або C++ необхідно буде перероблювати значну частину зробленої роботи. Саме тому компілятори спочатку і генерують AST. За таким підходом вищеописана задача вирішується в декілька етапів:

- Написання коду, який зчитує код на Java та генерує з нього AST;
- Написання коду, який отримує та зчитує AST та на основі дерева абстрактного синтаксису генерує код на C++.

Отже, в результаті такого розбиття вихідної задачі на під задачі з використанням AST на етапі генерування коду на C++ розробнику буде абсолютно неважливо який був вихідний код, бо на момент «перекладу» на іншу мову програмування абсолютно неважливо як виглядав початковий програмний код, адже він буде представлений в абстрактному вигляді. Окрім того, у випадку зміни синтаксису Java або C++ потрібно буде ввести лише невеличкі зміни, замість того щоб переписувати значну частину програмного коду.

Тепер розглянемо побудову дерева абстрактного синтаксису на простих прикладах.

Приклад 1:

Нехай є наступний вираз: $11 + (15 * 365)$. Наведемо кроки формування Abstract Syntax Tree даного виразу:

- Одразу варто зауважити, що при формуванні дерева абстрактного синтаксису неважлива наявність дужок. Отже, в даному виразі, перша операція, яка виконується – це додавання;
- Лівим доданком являється число 11;
- Правим доданком виступає результат операції множення;

- Множником з лівої сторони виступає число 15;
- Другим множником є число 365.

Його представлення у вигляді AST буде виглядати наступним чином:

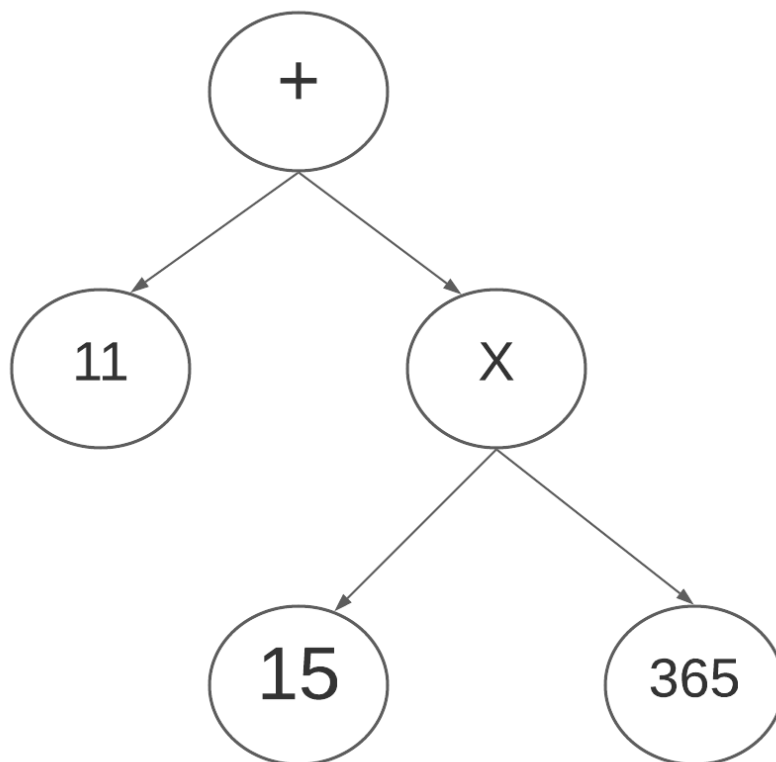


Рисунок 3.3 - Представлення виразу у вигляді AST

Приклад 2:

Нехай є наступний код:

```
public String checkEqualityTo11(int someNumber) {  
    if (someNumber > 11) {  
        return "Number is greater than 11";  
    }  
    else {  
        return "Number is not bigger than 11";  
    }  
}
```

Рисунок 3.4 - Приклад коду

Як можна побачити, зображена на рисунку 3.4 функція виконується порівняння деякого числа, яке подається на вхід функції з числом 11. У випадку, коли число більше 11, то функція повертає рядок «Number is greater than 11», а в іншому випадку – «Number is not bigger than 11».

Цей код є прикладом вузла з трьома дочірніми елементами:

1. Перевірка оператором if;
2. Інтерпретація результату у випадку якщо число більше 11 (результат перевірки оператором if дорівнює true);
3. Представлення результату, коли число менше 11 (ілюструється ситуація, коли результат перевірки дорівнює false).

Також необхідно наголосити, що не існує одного-єдиного «правильного» варіанту представлення коду у вигляді AST. Допускаються різні уявлення – наприклад, три людини можуть придумати три різні AST репрезентації і всі вони будуть вірні [15]. Отже, наводимо варіант можливого представлення фрагменту коду у вигляді дерева абстрактного синтаксису:

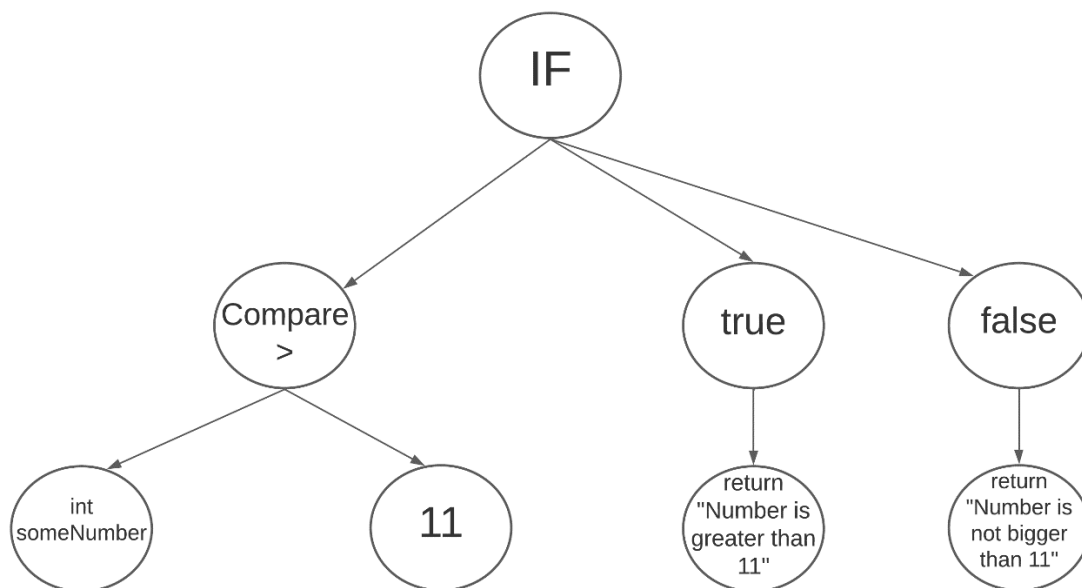


Рисунок 3.5 - Представлення коду з рисунку 3.3 у вигляді AST

Як можна побачити з наведених прикладів, то шлях обходу AST представлений у вигляді

$$n_1 d_1, \dots, n_k d_k, n_{k+1} d_{k+1},$$

де n_1 та $n_{k+1} \in T$, тобто являються термінальними вузлами для $i \in [2 \dots k]$, $n_i \in N$, тобто є нетермінальними вузлами для $i \in [1 \dots k]$,

$d_i \in \{\uparrow, \downarrow\}$ - напрямки руху (вгору та вниз), залежно від напрямку обходу дерева для створення шляху. У випадку якщо $d_i = \uparrow$, то $n_i \in \delta(n_i + 1)$. Якщо $d_i = \downarrow$, то $n_{i+1} \in \delta(n_i)$.

3.3 Code2Vec

Модель Code2Vec представляє код у вигляді векторів та використовується для отримання векторного представлення коду (code embeddings). Ця модель написана з використанням таких бібліотек як TensorFlow і Keras та навчається на більш ніж 14 мільйонах прикладів вихідного програмного коду. Принципи роботи даної моделі описані у статті «Code2Vec: Learning Distributed Representations of Code» [12]. Суть

подання коду у вигляді вектору аналогічна концепції векторної репрезентації слів: кожен фрагмент програмного коду, який використовується можна представити у вигляді певної множини дійсних чисел. Далі будуть наведені основні тези з даної публікації [12] з метою пояснення роботи Code2Vec.

Значення вектору, що був отриманий за допомогою Code2Vec містять в собі наступні ключові значення:

- Семантичне значення контексту;
- Чи є даний контекст «важливим». Іншими словами, яку увагу слід приділяти даному контексту.

Дані вектори потім використовуються в навчальному процесі моделі, оскільки модель розкладає фрагменти програмного коду на набір уявлень шляхів за допомогою дерев абстрактного синтаксису та вивчає атомарні представлення агрегованих шляхів. Отже, після використання даної моделі вихідний код буде представлено у вигляді Bag Of Path-Context.

Процедуру виявлення ознак в Code2Vec відбувається з використанням PathMiner - майнеру дерев абстрактного синтаксису з відкритим вихідним кодом [12, 25]. Наприклад, такий рядок як $a = 11$ можна представити у вигляді шляху по дереву абстрактного синтаксису наступним чином:

$$< a, NameExpr \uparrow AssignExpr \downarrow IntegerLiteralExpr, 11 >$$

Розглянемо AST та варіанти побудови можливих шляхів по заданому AST з використанням синтаксису PathMiner.

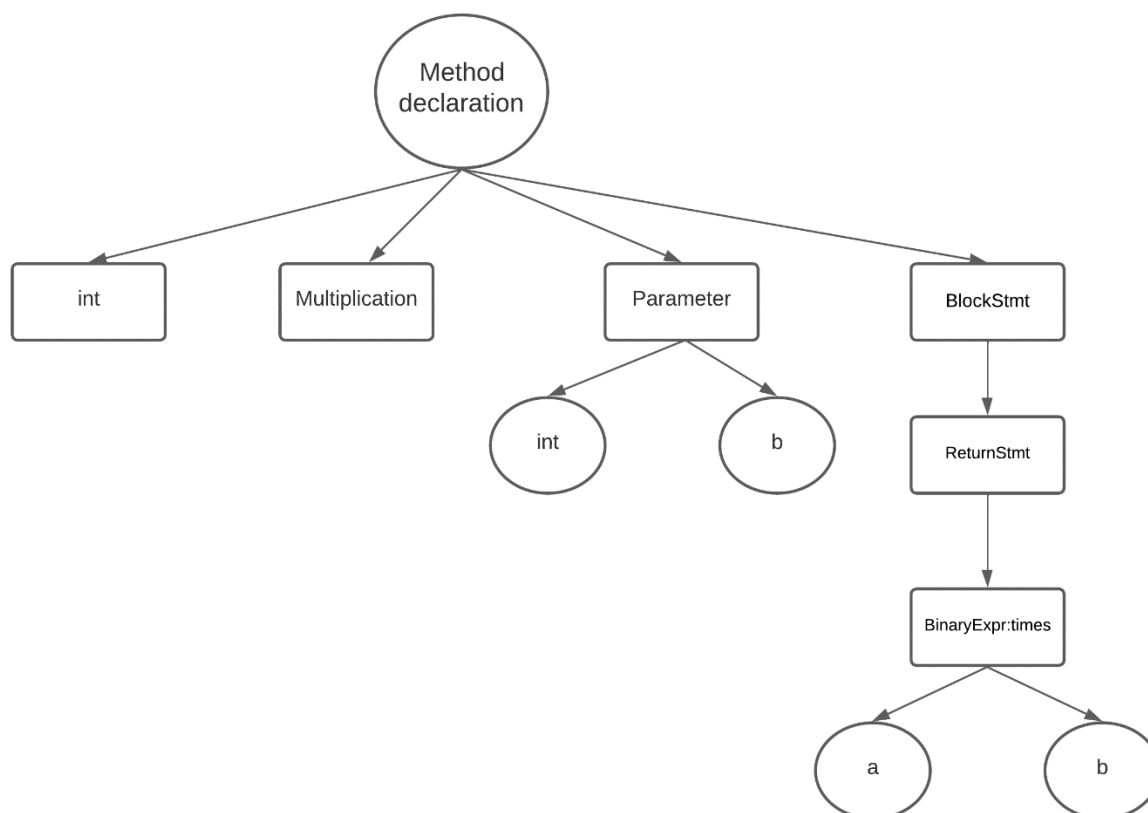


Рисунок 3.6 - Приклад Abstract Syntax Tree

Як можна побачити з наведеного вище рисунку, дана AST описує функцію під назвою Multiplication деякого числа a на число b . Дане дерево синтаксичної подібності можливо описати наступним чином:

1. $\langle \text{MultByX}, \text{Method Declaration} \downarrow \text{BlockStmt} \downarrow \text{ReturnStmt} \downarrow \text{Binary Expr: times}, b \rangle$;
2. $\langle b, \text{parameter} \uparrow \text{Method Declaration} \downarrow \text{BlockStmt} \downarrow \text{ReturnStmt} \downarrow \text{Binary Expr: times}, a \rangle$.

Отже, згідно з авторами моделі, основними аспектами підходу Code2Vec є:

- Фрагмент вихідного програмного коду може бути ефективно представлений в якості Bag Of Path-Context;
- Для здійснення точного прогнозу недостатньо одного контексту. Нейрон, на основі механізму уваги (attention-based

neuron) може оцінити важливість декількох контекстів шляхів та зробити відповідний прогноз після їх об'єднання;

- Code2Vec здатний розпізнавати невеликі відмінності між певними частинами коду, навіть якщо коди мають подібну синтаксичну структуру та мають багато спільних лексем та п-грам;
- Наявність великого корпусу, можливість передбачень імен методів між різними проектами;
- Незважаючи на той факт, що модель заснована на нейронній мережі, результат її роботи легко інтерпретується людиною.

В загальному, всі варіанти використання даної моделі можна поділити на дві великі групи [12]:

- Автоматична перевірка програмного коду;
- Отримання інформації та пошук необхідного API.

Що стосується автоматичної перевірки коду, то використання Code2Vec дозволяє пропонувати більш «підходящі» назви методів, виявляти випадки, коли назва функції не відповідає її призначенню та використанню. Тобто, таким чином значно покращується обслуговування коду та використання відкритих API полегшується. Окрім того, дану модель можна застосувати як частину системи, що перевіряє програмний код на плагіат.

В другому випадку застосування цієї моделі, отримане векторне представлення коду значно спрощує процедуру пошуку семантично схожих методів для покращення readability програмного коду. Наприклад, певний розробник пише на мові Java та використовує порівняння рядків методом equals() після попередньої їх обробки за допомогою методу toLowerCase(). При умові використання Code2Vec, стане можливим знайти семантично-

еквівалентний метод – `equalsIgnoreLowerCase()`. Іншими словами, це значно секономить час, необхідний на розробку, завдяки зменшенню зусиль на вивчення документації з метою оптимізувати існуючі функції.

Отже, підводячи підсумок, за допомогою застосування `Code2Vec` можна вирішити наступні практичні задачі машинного навчання:

1. Пошук коду;
2. Класифікація та тегування;
3. Ранжирування;
4. Виявлення плагіату програмного коду.

Наведемо приклад роботи `Code2Vec`, використовуючи веб-сайт [35], який розроблено саме для демонстрації основних принципів, описаних в статті [12].

Нехай є наступний фрагмент програмного коду:

```
String f(String txt) {
    txt = replace(txt, "&", "&");
    txt = replace(txt, "\"", "&quot;");
    txt = replace(txt, "<", "&lt;");
    txt = replace(txt, ">", "&gt;");
    return txt;
}
```

Рисунок 3.7 - Приклад програмного коду

Представлення даного коду у вигляді дерева абстрактного синтаксису буде мати наступний вигляд:

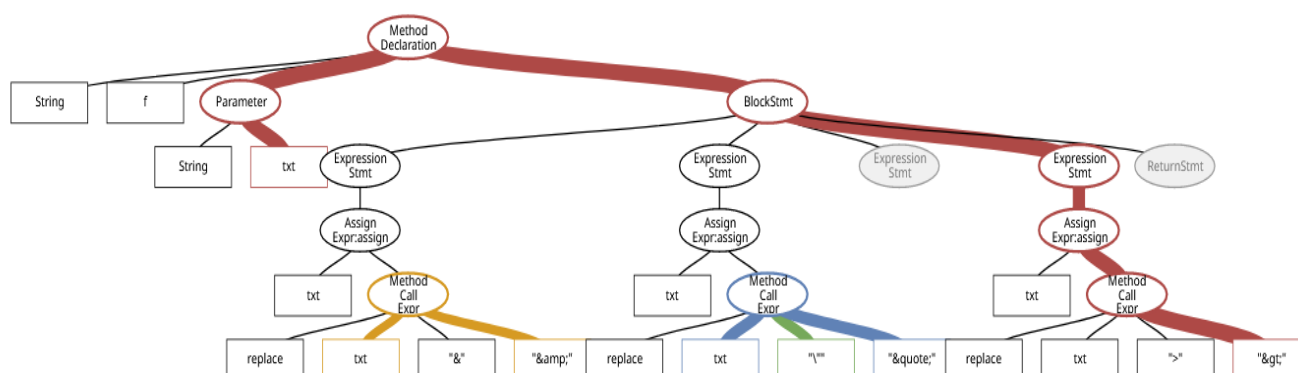


Рисунок 3.8 - Представлення коду у вигляді AST

3.4 Transformers

В галузі NLP Transformers – це архітектура, що призначена для вирішення задач перетворення виду seq2seq (sequence-to-sequence, «послідовність-послідовність»). Дана архітектура здатна легко обробляти long-range залежності. Трансформери покладаються на механізм самоуваги під час обчислення вхідних та вихідних даних та не використовують у своїй роботі rnn або згортку. Дана архітектура детально описується в статті “Attention is All You Need”[23]. За матеріалами цієї статті, Transformers являється моделлю трансдукції. В даному випадку термін «трансдукція» використовується для позначення процесу перетворення вхідних послідовностей на вихідні. Основна ідея даної архітектури полягає в тому, щоб повністю обробляти залежності між вхідними та вихідними даними з застосуванням механізму уваги та повтореннями.

Перед тим як розглядати архітектуру Transformer необхідно пояснити поняття самоуваги.

Згідно з авторами статті, самоуваги(внутрішня увага) — це механізм уваги, який пов’язує різні позиції однієї послідовності для обчислення представлення послідовності [3].

Розглянемо як працює механізм самоуваги на прикладі. Нехай у нас є наступне речення: «Чоловік забув купити хліб, бо він відволікся на телефонний дзвінок.». Модель буде намагатися встановити до чого саме в реченні відноситься займенник «він». Зобразимо це схематично:

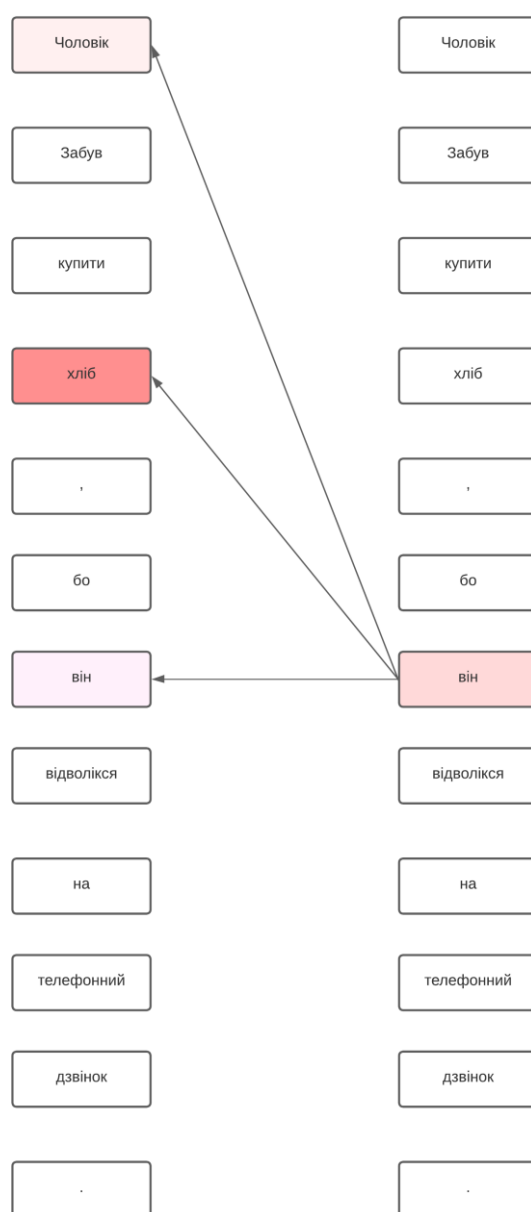


Рисунок 3.9 - Загальна ідея роботи механізму самоуваги

Як можна побачити з рисунку, алгоритм буде намагатись «зрозуміти» до якого саме слова відноситься займенник «він» в даному реченні. Безсумнівно, з лінгвістичної точки зору для людини не буде складати проблеми визначити який саме контекст і до чого відноситься займенник, проте алгоритм на відміну від людини, що практикує якусь мову і може визначити зв'язок іменника та займенника, має формально встановлювати зв'язок між цими словами. Це і завданням самоуваги: даний механізм дозволяє моделі дивитися на інші слова у вхідній послідовності, щоб краще зрозуміти певне слово в послідовності.

Тепер після введення та пояснення терміну самоуваги, перейдемо до розгляду загальної архітектури Transformer, яка наведена на рисунку нижче.

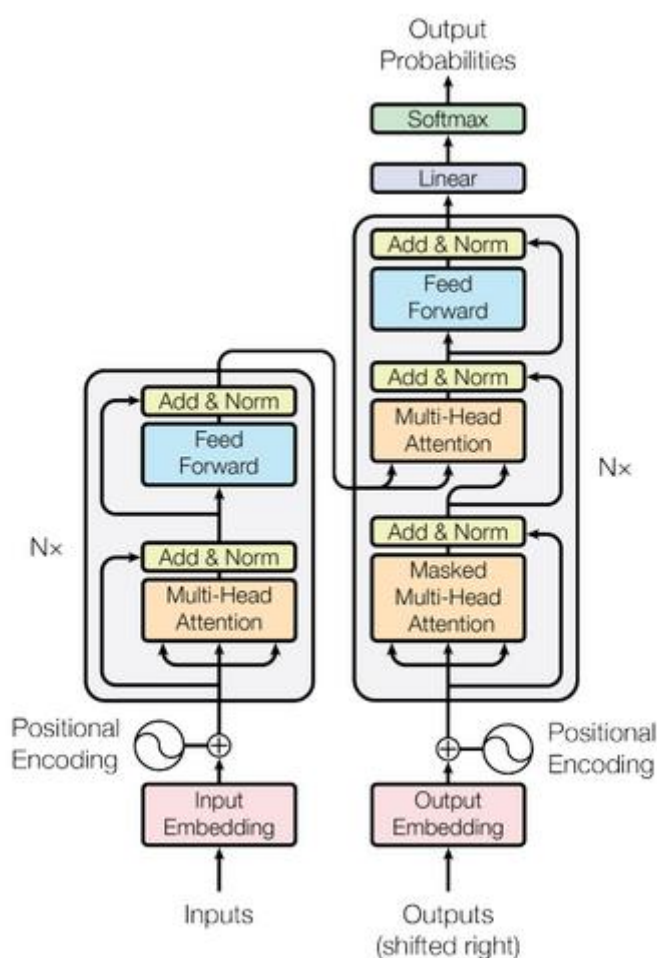


Рисунок 3.10 - Загальна архітектура Transformers

Як можна побачити з рисунку, наведена архітектура доволі складна, тому варто розглядати її по частинах.

Для початку зосередимо увагу на енкодерах та декодерах. З рис.3.10 видно, що у блока енкодера наявний 1 шар Multi-Head Attention, за ним слідує шар FNN (нейронної мережі нейронної мережі з прямим зв'язком). А декодер має ще один додатковий шар Masked Multi-Head Attention. Варто зазначити, що окрім шарів самоуваги та шарів FNN, декодери також мають ще один attention шар, а саме «Енкодер-Декодер». Це допомагає декодеру зосередитися на відповідних частинах вхідної послідовності. Також варто звернути увагу, що блоки енкодера і декодера насправді є безліччю однакових енкодерів і декодерів [4].

Тобто, стеки енкодера та стеки декодера мають однакову кількість імплементацій. Згідно з авторами, дане число являється гіперпараметром.

Далі розберемо як працюють енкодер та декодер поетапно:

- Векторні репрезентації слів вхідної послідовності передаються до першого енкодера;
- Подані векторні представлення трансформуються та передаються далі – до наступного енкодера;
- Вихідні дані з останнього енкодера передаються до всіх декодерів, як це зображено на рисунку 3.11.

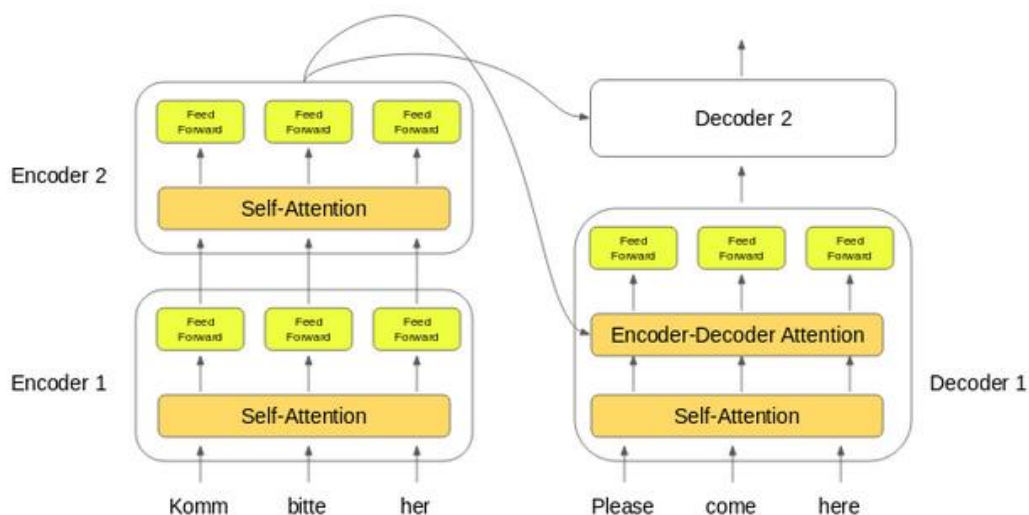


Рисунок 3.11 - Процес передачі даних з енкодера до декодера

Тепер зосередимось на Multi-Head Attention, який наявний в енкодерах та декодерах. Multi-Head Attention дозволяє моделі одночасно досліджувати інформацію з різних підпросторів уявлення в різних положеннях, що було сповільнено усередненням в рамках однієї голови уваги. Тобто, основною задачею Multi-Head Attention є надання моделі можливості зосередитися на різних позиціях шляхом обчислення власної уваги кілька разів з різними наборами векторів q , k і v , а потім взяти середнє значення всіх цих результатів, щоб отримати остаточне значення z [5, 23]. Тож замість того, щоб мати справу з величезними векторами та усереднювати декілька виходів, розмір векторів k , q та v зменшується до деякого меншого виміру — також зменшується розмір матриць W_q , W_k та W_v , зберігаються множинні набори (h) k , q і v та кожен такий набір називають «головою уваги». В результаті, всі набори значень об'єднуються для отримання кінцевого результату z замість того щоб проводити процедуру усереднення. Розмір отриманого вектору буде занадто великим для передачі до наступного шару, тому він зменшується за допомогою множення на іншу вивчену матрицю W_o [5]. Таким чином, Multi-Head

Attention дозволяє моделі одночасно оброблювати інформацію з різних підпросторів репрезентації даних.

Після цього відбувається процедура пост-обробки даних. Як вже було зазначено, вхідні слова представлені за допомогою векторної репрезентації. Ця робиться як для енкодера, так і для декодера. Векторне представлення слів не має жодної інформації про позицію, яка досягається в RNN завдяки їх послідовному характеру. Тим часом у механізмі самоуваги через наявність softmax будь-яка така інформація про позицію втрачається. Щоб зберегти інформацію про позицію, Transformer вводить вектор до окремих вхідних репрезентацій. Ці вектори дотримуються певної періодичної функції (наприклад, комбінація косинусів з різною частотою), яку модель вивчає і може визначити позицію окремого слова відносно один одного на основі значень. Цей введений вектор називається «позиційним кодуванням» і додається до вхідних вкладень у нижній частині стеків енкодерів і декодерів [3, 23].

3.5 Вибір методу для перевірки на наявність запозичень у програмному коді

Для імплементації вищеописаної системи було запропоновано два можливих варіанти реалізації перевірки на плагіат. Після їх реалізації, було проведено порівняльний аналіз методів перевірки програмного коду на плагіат з метою вибору найбільш ефективного для застосування в системі для перевірки на виявлення можливих запозичень.

Першим запропонованим варіантом є багат шаровий перцептрон, який порівнює векторні репрезентації програмного коду, отримані за допомогою векторної моделі Code2Vec [12], а другим – адаптація системи TreeBERT [26] для виконання задачі перевірки програмного коду на плагіат. Далі більш детально опишемо запропоновані підходи до вирішення задачі.

В той час, коли перший запропонований варіант є «класичною» імплементацією, оскільки використовуються векторні репрезентації отримані за допомогою Code2Vec без модифікацій, TreeBERT використовує вдосконалену векторну репрезентацію. В загальному, можна виділити такі відмінності системи TreeBERT:

- **Архітектура:** Згідно з авторами даної моделі [26], ця система заснована на оригінальній імплементації енкодера-декодера на основі архітектури «трансформер» [23]. Імплементація TreeBERT відрізняється від оригінального Transformer лише тим, що енкодер адаптовано для налаштування необхідної розмірності вхідних даних шляхом додавання повнозв'язного шару.
- **Фрагмент коду, який подається на вхід,** після перетворення на AST розбивається на послідовність токенів. В результаті, вихідний код буде представлено в наступному вигляді: $C = [LT], x_1, x_2, \dots, x_m, [CLS]$, де $[LT]$ – вектор подання, $[CLS]$ – вектор представлення, m – довжина фрагменту коду. Окрім того, що токен $[LT]$ позначає ідентифікатором початку речення на стороні декодера, даний токен ще й позначає цільову мову програмування. Наприклад, для мови програмування Python токен $[LT]$ буде дорівнювати $[PLT]$, а для мови Java – $[JLT]$. Після калькуляції, вектор C подається на вхід декодера.
- **Шлях** репрезентується наступним чином: $p_i = v_1^i v_2^i \dots v_{L-1}^i x_t^i$, де x_t^i - термінальний вузол шляху – це токен якого в відповідному фрагменті програмного коду. Та на відміну від Transformers, TreeBERT не додає кодування позиції (position encoding) для того щоб надавати інформацію про позицію до векторів шляхів, але надає зазначену інформацію вузлам дерева

використовуючи репрезентацію позиції вузла під час формування представлення вузла.

- **Представлення токенів:** В даній моделі використовується byte-pair encoding (BPE) представлена в роботі [27] для вирішення можливої проблеми «розмиття» векторної репрезентації.
- **Представлення позиції вузла:** на стороні енкодера було додано представлення позиції вузла до репрезентації вузла. Перевагою даної репрезентації є зменшення кількості параметрів та екстраполяція [28].

3.4.1 Порівняльний аналіз запропонованих методів

Метою проведення даного аналізу було визначення ефективності наступних моделей:

- Multilayer perceptron + Code2Vec
- TreeBERT
- MOSS

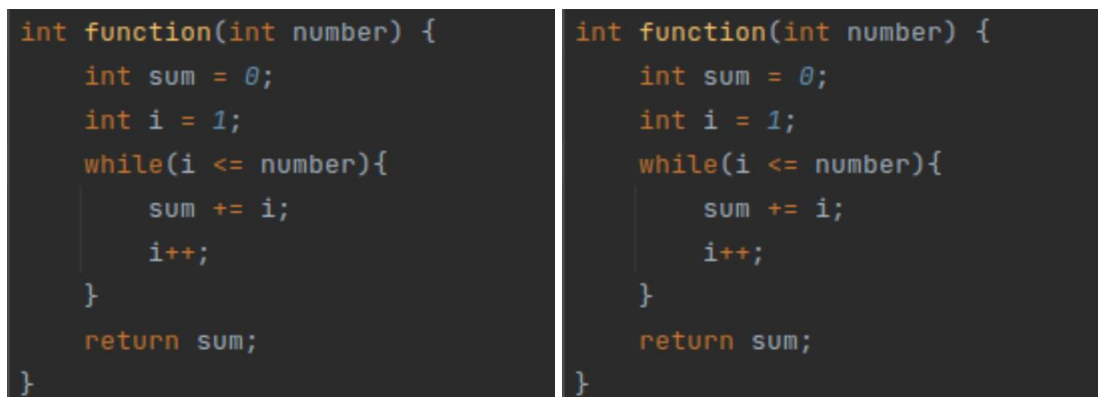
Система MOSS присутня в даному експерименті з метою оцінки ефективності інших методів, що пропонуються.

В якості параметрів для перевірки було обрано відсоток подібності та час проведення перевірки. Варто відмітити, що всі проаналізовані пари прикладів в ході цієї перевірки містять плагіат. Тобто, за максимальну оцінку, яку може надати система вважаємо 100 %.

В ході перевірки було розглянуто 4 типи плагіату програмного коду та окремо розглянуто приклад, що демонструє проблему порушення хешування. Всі розглянуті фрагменти коду під час перевірки, були написані на мові Java. Далі наведемо приклади, що аналізувалися.

Приклад 1. Ідентичний програмний код

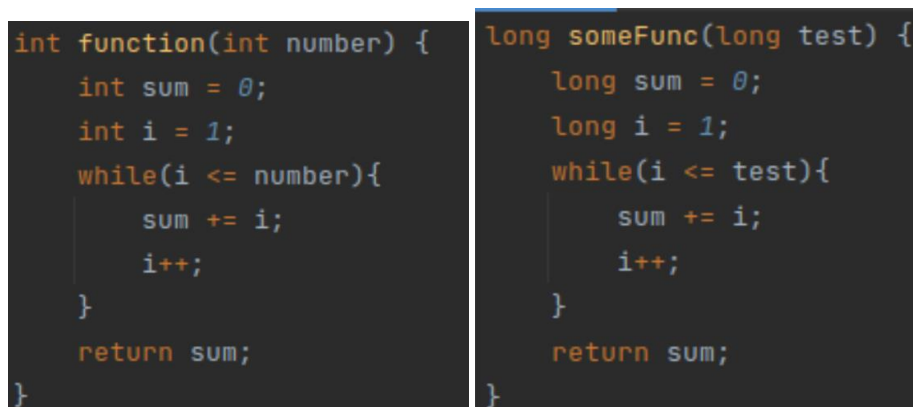
На рисунку 3.12 наведено пару ідентичних фрагментів програмного коду.



```
int function(int number) {
    int sum = 0;
    int i = 1;
    while(i <= number){
        sum += i;
        i++;
    }
    return sum;
}
```

Рисунок 3.12 - Ідентичні фрагмент програмного коду

Приклад 2. Зміна назв, типів значень, змінних та/або констант

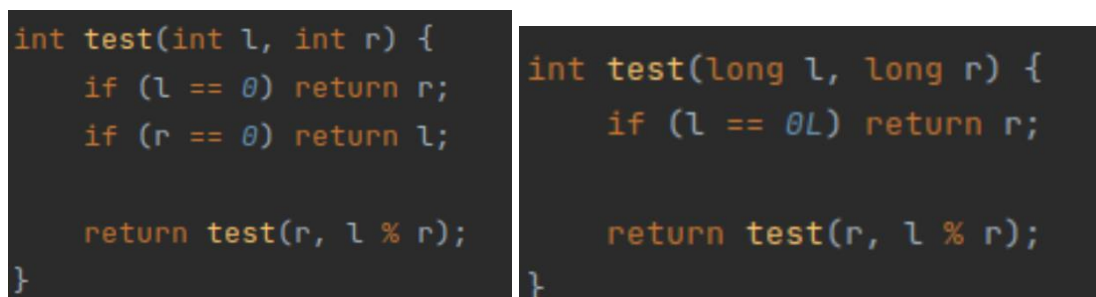


```
int function(int number) {
    int sum = 0;
    int i = 1;
    while(i <= number){
        sum += i;
        i++;
    }
    return sum;
}
```

```
long someFunc(long test) {
    long sum = 0;
    long i = 1;
    while(i <= test){
        sum += i;
        i++;
    }
    return sum;
}
```

Рисунок 3.13 - Приклад другого типу плагіату програмного коду

Приклад 3. Видалення або додавання окремих операторів



```
int test(int l, int r) {
    if (l == 0) return r;
    if (r == 0) return l;

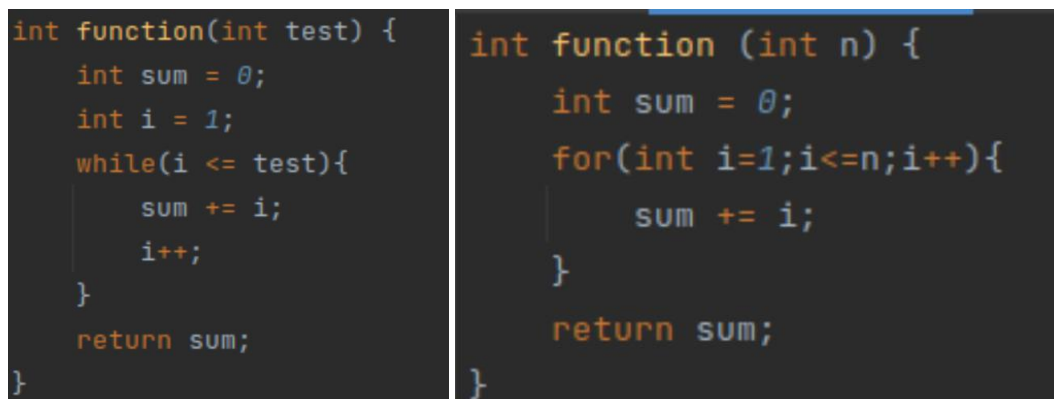
    return test(r, l % r);
}
```

```
int test(long l, long r) {
    if (l == 0L) return r;

    return test(r, l % r);
}
```

Рисунок 3.14 - Приклад третього типу плагіату

Приклад 4. Семантично еквівалентний код

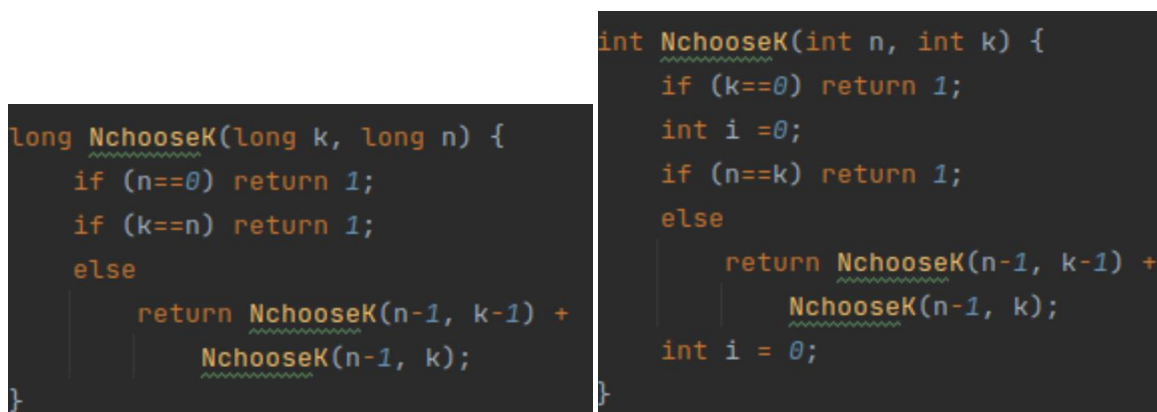


```
int function(int test) {
    int sum = 0;
    int i = 1;
    while(i <= test){
        sum += i;
        i++;
    }
    return sum;
}

int function (int n) {
    int sum = 0;
    for(int i=1;i<=n;i++){
        sum += i;
    }
    return sum;
}
```

Рисунок 3.15 - Приклад семантично еквівалентного коду

Приклад 5. Приклад проблеми порушення хешування



```
long NchooseK(long k, long n) {
    if (n==0) return 1;
    if (k==n) return 1;
    else
        return NchooseK(n-1, k-1) +
               NchooseK(n-1, k);
}

int NchooseK(int n, int k) {
    if (k==0) return 1;
    int i = 0;
    if (n==k) return 1;
    else
        return NchooseK(n-1, k-1) +
               NchooseK(n-1, k);
    int i = 0;
}
```

Рисунок 3.16 - Приклад проблеми порушення хешування

Результати аналізу наведено у таблиці 3.1.

Таблиця 3.1. Результати перевірки

Тип плагіату	TreeBERT	Multilayer perceptron + Code2Vec	MOSS
Ідентичний програмний код	100 %	100 %	100 %

Зміна назв, типів значень, змінних та/або констант	95 %	94 %	100 %
Видалення або додавання окремих операторів	95 %	95 %	92 %
Семантично еквівалентний код	79 %	85 %	64 %
Приклад проблеми порушення хешування	93 %	98 %	10 %

З результатів перевірки видно, що обидва запропонованих методи мають досить високі показники за точністю відсоткового виявлення співпадінь у фрагментах сирцевого програмного коду. На відміну від системи MOSS, обидва розроблених методи не є чутливими до проблеми порушення хешування. В той коли система MOSS вразлива до даної проблеми та видає результат у 10 % схожості, запропоновані методич отримують результат у 93% та 98% відповідно. Також розроблені моделі краще справляються з виявленням плагіату четвертого типу, а саме семантично еквівалентного коду. Варто зазначити, що тестові фрагменти програмного коду були дуже невеликі за розміром, але даних прикладів достатньо щоб продемонструвати переваги імплементованих моделей.

При порівнянні двох розроблених моделей, в якості висновку можна відзначити, що багатошаровий перцептрон + Code2Vec має кращі відносні

показники та через це вона була обрана для реалізації системи перевірки програмного коду на наявність плагіату.

3.6 Проектування системи

HLD діаграма пропонованої системи наведено на рисунку 3.17.

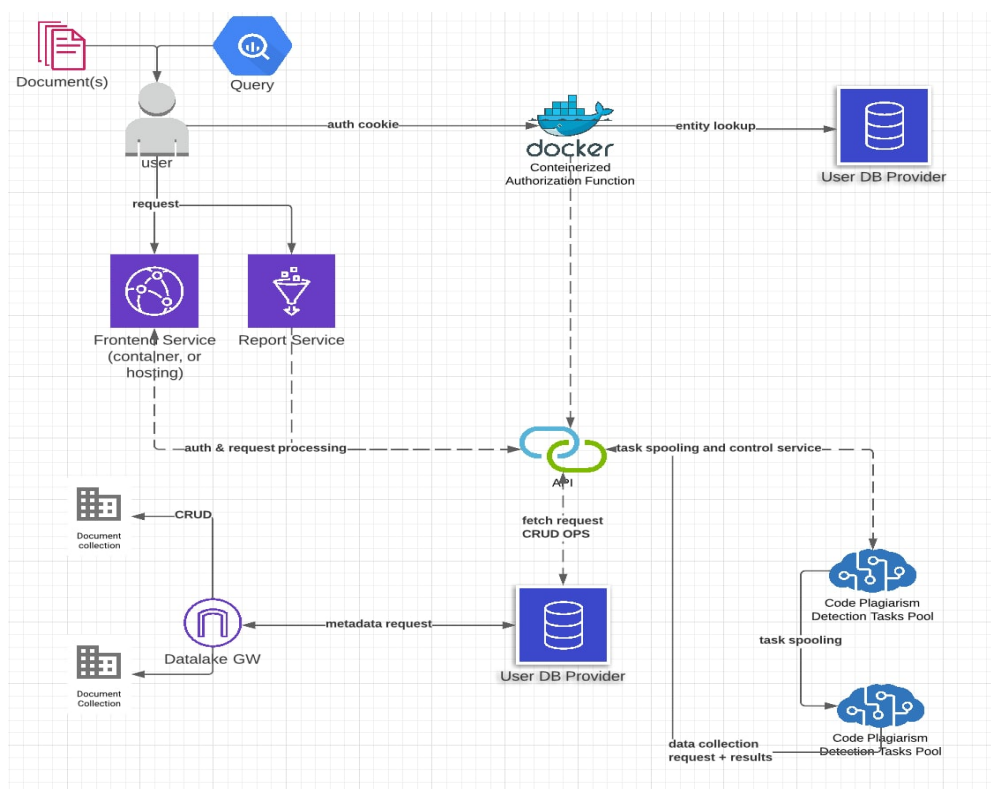


Рисунок 3.17. HLD діаграма системи

Опис запропонованої архітектури:

Для імплементації цього алгоритму, з урахуванням масштабованості, пропонується реалізувати наступну архітектуру, що буде застосовувати сучасні парадигми розгортання сервісів та зберігання даних, як то контейнеризація (що де факто вже є сучасним стандартом пакування коду), оркестрація та data lake.

Для зручного опису архітектури розіб'ємо її розгляд на окремі бізнес функції, а саме:

- Авторизація
- Обслуговування документів
- Порівняння документів
- Репрезентація Даних та Інтерфейс
- API шлюз

Авторизація – сама імплементація не суттєво важлива, оскільки може бути застосована будь яка база даних користувачів, чи інший флоу, що зможе надати інформацію про користувача, як то SAML. Суттєво важливим у даному випадку є сегрегація прав та можливість сервісу авторизації розрізняти права користувача на

- видозміну та доступ до колекцій документів
- застосування додатку
- зону видимості документів
- можливість додавати / вилучати документи
- можливість генерувати звіт

Обслуговування документів. Код, що пишуть студенти у якості творчого завдання, зазвичай не буде одноманітним у контексті того, що для виконання різних завдань студенти можуть і будуть використовувати різні мови програмування. Враховуючи це, а також природу того, що з часом даних буде ставати все більше (через політику зберігання протоколів/додатків до протоколів, згідно законодавства на певний термін) пропонується одразу розділяти код на декілька окремих баз даних, що будуть логічно репрезентувати якусь семантичну одиницю, як то наприклад мова програмування, кафедра, чи інша абстракція. Такий підхід називається sharding та вимагає деяких попередніх дій, що до стандартизації запитів до баз даних, а в гірших випадках робить частковий vendor lock у вигляді застосування якоїсь конкретної версії бази, типу бази, чи продукту у цілому.

За для полегшення та майже повного вирішення цієї проблеми був винайдений механізм data lake, що буде слугувати проксі між базами даних та кінцевим клієнтом/сервісом. Перевагою цього буде те, що більшість технологій data lake імплементують свій стандартизований API для пошуку та отримання інформації із баз даних, що вони об'єднують та надають single point of access для сервісу, таким чином імплементатору коду не потрібно буде думати про створення багатьох методів для обробки запитів до різних діалектів СУБД, чи NOSQL, натомість імплементация data lake буде сама надавати данні, а також категоризувати та зберігати їх у кінцевих базах даних.

Порівняння документів. За для того, щоб забезпечити відносну швидкість та багатопоточність запитів до ПО, що буде використовувати алгоритм та модель, що описані в цій роботі пропонується контейнізувати кінцеву модель, та відповідне середовище її роботи у контейнер docker, що буде легко масштабований за допомогою сучасних оркестраторів на кшталт Kubernetes, чи у більш простій імплементации docker swarm. Фактично це надасть гнучкість використання доступного апаратного забезпечення за принципом on-demand та легкість оновлення моделі, при потребі оновлення.

API шлюз. За для того, щоб полегшити додання нового функціоналу/видалення старого пропонується об'єднати усі мікросервіси за допомогою API шлюзу, що і надає можливості інтеграції та взаємодії різних компонентів та технологій.

Репрезентація даних та інтерфейс не є суттєво важлива у рамках цієї роботи, проте більшість сучасних фреймворків буде підтримувати взаємодію із API, що і буде репрезентатором endpoint усіх сервісів з обробки даних/авторизації/оркестрації контейнерів для фронтенду і транзитивно користувачеві.

3.7 Приклад роботи методу для виявлення наявності плагіату

В даному пункті буде наведено приклад можливого використання розробленого додатку, що призначений для ідентифікації наявності запозичень в програмному коді.

На рисунку 3.18 наведено загальний вигляд інтерфейсу користувача.

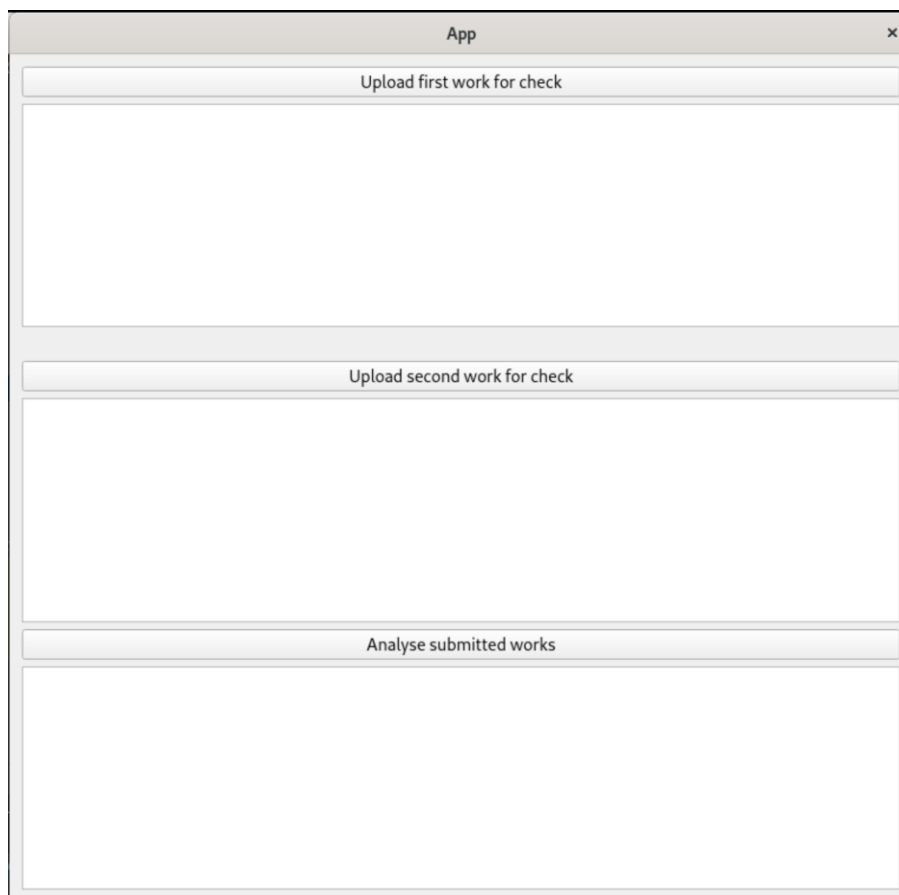


Рисунок 3.18 - Приклад базового інтерфейсу користувача

Для того, щоб надати системі програмний код для перевірки можливий як варіант мануального введення даних в текстові поля, так і через завантаження файлів з програмним кодом. Завантаження файлів до системи відбувається через кнопки «Upload first work for check» та «Upload second work for check» відповідно. Після натискання будь-якої з цих двох кнопок буде відкрито провідник через який можливо обрати потрібний файл, як це наведено на рис 3.19.

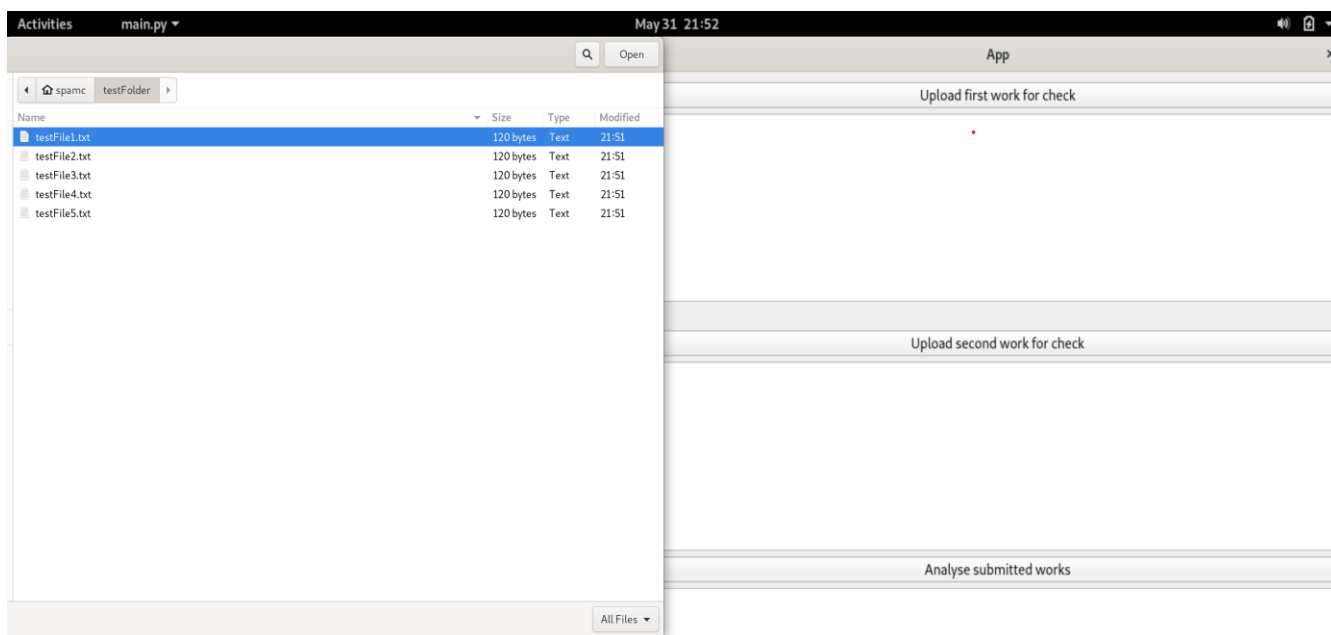


Рисунок 3.19 - Вибір файлу для перевірки

Після вибору необхідних файлів для перевірки, їх зміст буде відображено в текстових полях. Приклад завантажених та готових для перевірки файлів продемонстровано на рис 3.20.

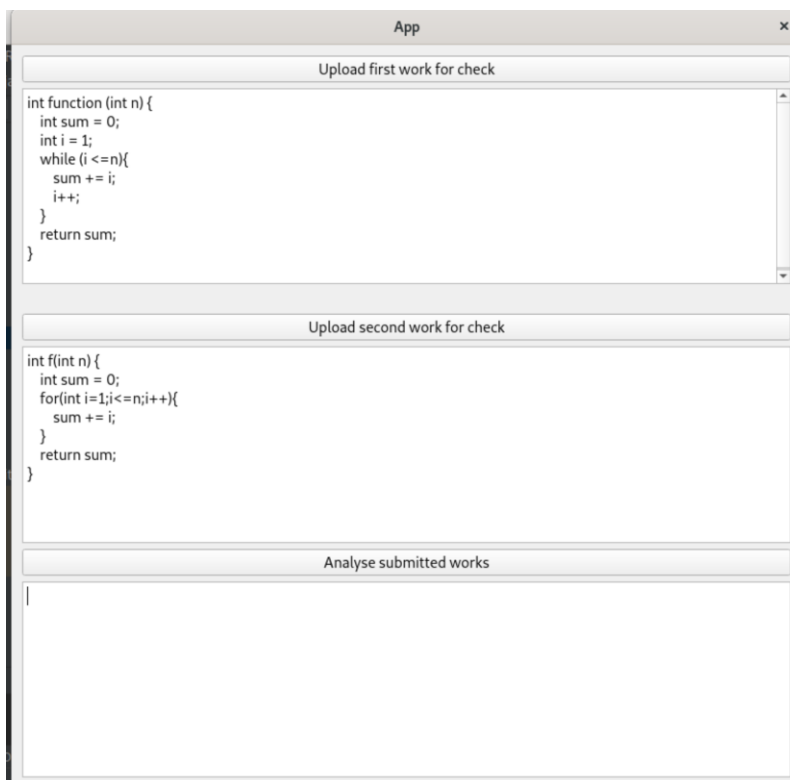


Рисунок 3.20 - Демонстрація змісту завантажених до системи файлів

Для того, щоб провести перевірку на плагіат необхідно натиснути кнопку «Analyse submitted works». Варто зазначити, що в даній версії системи реалізована перевірка програмного коду, написаного лише на мові програмування Java, але розроблений метод легко масштабується – необхідною умовою підтримки інших мов програмування є розробка парсерів для інших мов, внесення змін до методу перевірки не потрібні.

Як тільки користувач натисне зазначену кнопку, то зміст файлів буде надано моделі для подальшого аналізу. Після проведення перевірки на наявність запозичень, результат перевірки буде відображено у полі результатів. Поле результатів доступне лише для читання, для того, щоб запобігти зміні результатів перевірки. Приклад надання результатів системою наведено на рисунку 3.21.



Рисунок 3.21 - Приклад надання результатів перевірки

3.8 Висновки до розділу

Даний розділ було присвячено проектуванню та розробці авторської системи для перевірки програмного коду на плагіат. В рамках даного розділу було запропоновано архітектуру системи, та досліджено ключові поняття для цієї системи, як то AST, Code2Vec та Transformers, проведено порівняльний аналіз автоматизованих методів перевірки на плагіат з метою виявлення оптимального для використання в системі та продемонстровано приклад роботи базової версії додатку.

Запропонований алгоритм поділяється на наступні логічні кроки:

- Фрагменти коду перетворюються на дерева абстрактного синтаксису;
- Отримані дерева абстрактного синтаксису проходять процедуру векторизації;
- Вектори, що були отримані в результаті попереднього перетворення, порівнюються на подібність.

В розділі була запропонована HLD діаграма системи. Для опису даної архітектури було виділено та описано наступні бізнес-функції:

- Авторизація;
- Обслуговування документів;
- Порівняння документів;
- Репрезентація Даних та Інтерфейс;
- API шлюз.

В якості методу для перевірки на плагіат було запропоновано використання наступних двох моделей:

- Multilayer perceptron + Code2Vec
- TreeBERT
-

Модель на основі Multilayer perceptron + Code2Vec була розроблена та натренована на вибірці, що складається з 14 мільйонів фрагментів коду на мові Java, що були попередньо оброблені Code2Vec, для виявлення подібності у векторах, у які вихідний код перетворює Code2Vec.

Для існуючої моделі TreeBERT було допрацьовано функцію перевірки на подібність, адже автори даної системи не використовували її з метою перевірки на плагіат, а основною ціллю використання, згідно з авторами – є генерація коду .

Після проведення порівняльного аналізу було обрано модель, яка порівнює репрезентації отримані за допомогою Code2Vec в якості методу, який буде виконувати перевірку на наявність плагіату в системі, що пропонується та реалізовано, в рамках цієї роботи.

4 РОЗРОБКА СТАРТАП ПРОЕКТУ

4.1 Ідея проекту

Ідея даного проекту полягає у розробці системи, яка може перевіряти програмний код у лабораторних, контрольних роботах студентів на плагіат. На сьогоднішній день, запозичення в студентських роботах є дуже поширеним явищем через низку причин, а безпосередньо плагіат програмного коду є досить серйозною проблемою в вищих навчальних закладах технічного напрямку. Через велику кількість робіт та брак часу, викладач не завжди має змогу ретельно перевіряти та порівнювати роботу студента з роботами одногрупників. В результаті, студенти запозичують роботу одногрупника та отримують високу оцінку, не прикладаючи зусиль для виконання завдання самостійно.

В перспективі, такі дії впливають на залишковий рівень знань студента та репутацію університету.

У даному розділі описано послідовність розробки стартап проекту на основі проведеного дослідження. Сутність ідеї проекту викладена в таблиці 4.1.

Таблиця 4.1 Сутність ідеї стартапу

Зміст ідеї	Напрямки застосування	Переваги для користувача
Система для перевірки робіт студентів з програмування на плагіат	Перевірка програмного коду на плагіат у навчальних закладах	Можливість підвищення ефективності перевірки студентських робіт на плагіат, підвищення рівню залишкових знань студентів

Тепер необхідно розглянути вже наявні методи та інструменти для перевірки вихідного коду на наявність плагіату. Порівняння техніко-

економічних характеристик власного рішення з конкурентними наведено в таблиці 4.2.

Таблиця 4.2. Слабкі, нейтральні та сильні характеристики ідеї додатку

№ п/п	Техніко-економічні характеристики	Потенційні конкуренти			W Слабк а сторон а	N Нейтральна сторона	S Сильна сторона	
		Даний проект	MOSS	JPlag				
1	Надання статистики після перевірки	+	+	+			+	
2	Використання архітектури Transformers	+	-	-			+	
3	Вразливість до порушення хешування	-	+	+			+	

Як результат проведення аналізу слабких, нейтральних та сильних сторін системи, можна зробити висновок про її конкурентоспроможність та наявність всього необхідного функціоналу. Тому вона може протистояти конкурентам у даній галузі.

4.2 Технологічний аудит ідеї проекту

Обрані технології конкретного функціонала для реалізації проекту представлені у таблиці 4.3 в рамках технологічного аудиту ідеї проекту.

Таблиця 4.3. Технологічний аудит проекту

№ п/п	Ідея проекту	Технології	Наявність технології	Доступність технології

1	Розробка системи для перевірки на плагіат програмного коду студентських робіт	Мова програмування Python	Наявна	Доступна безкоштовно
		Transformers	Наявна	Доступна безкоштовно
		Бібліотеки Keras, Tensorflow	Наявні	Доступна безкоштовно

Представлений набір технологій є наявним та доступним.

4.3 Аналіз ринкових можливостей

Для запуску стартап-проекту необхідно провести аналіз його ринкових можливостей, щоб визначити конкурентоспроможність і зацікавленість потенційних клієнтів. Щоб провести аналізу попиту, потрібно оцінити основних гравців, що є на ринку. Результати аналізу зображені в таблиці 4.4.

Таблиця 4.4. Характеристика ринку стартап проекту

№ п/п	Показники ринку	Характеристика
1	Кількість головних гравців, од	< 10
2	Динаміка ринку (якісна оцінка)	На даний момент ринок активно розвивається
3	Наявність обмежень для входу (характер)	Обмеження для входу на ринок відсутні
4	Специфічні вимоги до стандартизації та сертифікації	Відсутні
5	Середня норма рентабельності в галузі (або по ринку), %	Невідомо

Тепер проаналізуємо групи потенційних клієнтів та їх характеристики. Результати аналізу представлено у таблиці 4.5.

Таблиця 4.5. Характеристика потенційних клієнтів

№ п/п	Формуюча ринок потреба	Цільова аудиторія	Відмінності у поведінці різних цільових груп	Вимоги споживачів до продукту
1	Ефективна перевірка робіт на наявність запозичень	Навчальні заклади	Розміри та об'єми необхідних перевірок	Необхідний функціонал для завантаження, перевірки роботи – порівняння як і двох робіт, так і порівняння роботи з колекцією документів
2	Зменшення кількості часу на перевірку роботи	Навчальні заклади	Розміри та об'єми необхідних перевірок	Можливість бачити результати перевірки та сформувані звіт про результат перевірки роботи студента

Також необхідно провести також аналіз середовища ринку та визначити фактори загроз. Результати аналізу загроз представлені в вигляді таблиці 4.6.

Таблиця 4.6. Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Конкуренція	Поява нової великої компанії	Вихід з ринку
2	Невеликий попит серед користувачів	Недостатня та обмежена кількість користувачів	Таргетована реклама, проведення дослідження щодо можливості розширення функціоналу

Окрім цього, було проведено аналіз факторів можливостей. Результати даного дослідження продемонстровані в таблиці 4.7.

Таблиця 4.7. Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Зниження популярності товарів конкурентів	Більш ефективна перевірка робіт на плагіат, яку не надають конкуренти	Таргетована реклама, проводити презентації ,тобто, демонструвати користувачам ефективність системи
2	Залучення нових клієнтів	За допомогою реклами та презентації переманювати клієнтів	Виступи на конференціях, таргетована реклама, надання пробних безкоштовних підписок

Проаналізувавши таблицю можна зробити висновок, що основною проблемою стартап-проекту є вузька функціональність і обмежена клієнтська база. Саме тому, спочатку варто розширювати функціональність та відкривати нові напрямки розвитку продукту.

Проведемо аналіз наявних пропозицій на ринку та виокремимо загальні характеристики, що наведено в таблиці 4.8.

Таблиця 4.8. Аналіз конкуренції на ринку

Особливість конкурентного середовища	В чому проявляється характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентноспроможною)
Вказати тип конкуренції	Олігополія	Брати до уваги ціни, що пропонують конкуренти за свої системи
За рівнем конкурентної боротьби	Національний	Врахування цін конкурентів, таргетована реклама та надання більш

		якісних результатів перевірки студентських робіт
За галузевою ознакою	Внутрішньогалузева	Створити більш ефективне рішення, ніж в конкурентів
Конкуренція за видами товарів	Товарно-видова	Можливе розширення функціоналу за потребами клієнта
За характером конкурентних переваг	Нецінова	Збільшувати зацікавленість користувачів
За інтенсивністю	Марочна	Отримати ім'я та/або товарний знак

Згідно з проведеним ступеневим аналізом конкуренції на ринку в ролі початкової стратегії можна обрати розширення функціоналу з метою отримання нових клієнтів.

В таблиці 4.9 зображено аналіз умов конкуренції в галузі.

Таблиця 4.9. Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Конкуренти		Постачальники	Клієнти	Товари-замінники
	Прямі	Потенційні			
	MOSS, JPlag, Copyleaks	Інформація відсутня			
Висновки	Існує певна кількість вирішень задачі виявлення плагіату програмного коду	Інформація відсутня	Умови не диктуються постачальниками послуг	У деяких клієнтів аналоги або не використовуються, або є певні домовленості з конкурентами.	Обмежень немає

Далі приведена таблиця 4.10 факторів конкурентоспроможності.

Таблиця 4.10. Фактори конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування
1	Підвищена ефективність перевірки	Багато систем мають досить низький відсоток правильно виявлених випадків плагіату та окрім того, у деяких з існуючих систем вже знайдено велику кількість вразливостей, які можуть бути використані для фальсифікації результатів перевірки
2	Кросплатформеність	Система буде доступна на всіх пристроях та платформах

За отриманими факторами конкурентоспроможності було проведено аналіз сильних та слабких сторін стартап проекту. Вони наведені в таблиці 4.11.

Таблиця 4.11. Порівняльний аналіз сильних та слабких сторін

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів конкурентів в порівнянні з нашим						
			-3	-2	-1	0	+1	+2	+3
1	Підвищена ефективність перевірки	18			+				
2	Кросплатформеність	15				+			

На основі виділених ринкових загроз та можливостей, сильних та слабких сторін, було проведено SWOT-аналіз стартап проекту, і наведено в таблиці 4.12.

Таблиця 4.12. SWOT-аналіз проекту

Сильні сторони: Доступність, адаптивність, простота у використанні,	Слабкі сторони: Для роботи з великою кількістю користувачів та постійного
--	--

орієнтованість виключно на газуль науки та освіти	розширення бібліотеки робіт необхідні ресурси
Можливості: Підвищення середнього рівня знань студентів, впровадження дисципліни, скорочення часу на перевірку студентських робіт викладачами, підвищення ефективності перевірок робіт	Загрози: Ризик низького початкового попиту через існування менш ефективних проєктів з відкритим кодом і безкоштовною ліцензією

На основі сильних сторін та можливостей проєкту, наведених у матриці SWOT-аналізу, можемо зробити висновок, що даний стартап-проєкт є конкурентоспроможним і має право на життя.

Таблиця 4.13. Альтернативи ринкового впровадження стартап-проєкту

№ п/п	Альтернатива ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Укладання контрактів з державними установами	50%	3-6 місяців
2	Розповсюдження проєкту через конференції, презентації	80%	5-7 місяців

Згідно з результатами, наведеними у таблиці 4.13, найбільш вигідною та перспективною є альтернатива 2, бо в цьому випадку значно більша вірогідність отримання ресурсів, не дивлячись на трохи більший час реалізації.

4.4 Розробка ринкової стратегії проєкту

Для розробки ринкової стратегії, необхідно визначити охоплення ринку через визначення цільових груп потенційних споживачів, що наведено в таблиці 4.14.

Таблиця 4.14. Вибір цільових груп потенційних користувачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів прийняти продукт	Орієнтовний попит в межах цільової групи	Інтенсивність конкуренції в сегменті	Простота входу в сегмент
1	Вищі навчальні заклади	Висока	Висока	Середня	Середня
2	Приватні навчальні інституції – курси програмування, тощо.	Низька	Висока	Низька	Середня

В результаті було обрано цільові групи потенційних користувачів, яким необхідна перевірка програмного коду на плагіат.

Розглянемо базову стратегію розвитку для стартап-проекту, що наведена в таблиці 4.15.

Таблиця 4.15. Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентноспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
1	Створення більш ефективної системи виявлення плагіату програмного коду на основі існуючих систем	Ринкове позиціонування	Підвищена ефективність перевірки, надійність	Оперативне оновлення продукту

В таблиці 4.16 визначено стратегію конкурентної поведінки на ринку.

Таблиця 4.16. Визначення базової стратегії конкурентної поведінки

№ п/п	Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента	Стратегія конкурентної поведінки
1	Ні, даний проект не є «першопрохідцем» на ринку.	Так, разом із пошуком шляхів залучення нових клієнтів та пошуків можливостей подальшого розширення функціоналу	Так, через схожість функціоналу. Мета даного проекту у підвищенні точності та ефективності перевірки	Зайняття конкурентної ніші

4.5 Розробка маркетингової програми

Першим кроком розробки маркетингової програми стартап-проекту є формування ключових переваг маркетингової концепції товару, який отримає споживач - зазначені у таблиці 4.17.

Таблиця 4.17. Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує продукт	Ключові переваги перед конкурентами
1	Перевірка робіт студентів на плагіат	Зручне завантаження та	Можливість більш ефективної

		ефективна перевірка роботи на плагіат	перевірки робіт на плагіат
2	Можливість формування статистичних звітів за результатами перевірки	Наявність можливості формування статистичного звіту після перевірки	Формування звітів після перевірки

Розроблено трирівневу модель товару, наведену в таблиці 4.18.

Таблиця 4.18. Трирівнева модель товару

Рівні товару	Сутність та складові	
I.Товар за задумом	Система для перевірки студентських робіт з програмування на плагіат	
II.Товар у реальному виконанні	Властивості/характеристики	Розмір
	Інтерфейс користувача	10 Мб
	Модуль перевірки на плагіат	20 Мб
	Якість: тестування додатку відповідно до сценаріїв	
	Пакування: веб-додаток	
	Марка: без марки	
III.Товар з підкріпленням	Надання додаткових напрямків згідно з побажаннями клієнтів.	
За рахунок чого потенційний товар буде захищено від копіювання: захист інтелектуальної власності		

Визначення цінових меж показано у таблиці 4.19.

Таблиця 4.19. Визначення меж встановлення ціни

№ п/п	Рівень цін на товари- замінники	Рівень цін на товари- аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межа встановлення ціни на товар
----------	------------------------------------	-------------------------------------	---	--

1	2900 – 4000 грн	0 – 6000 грн	~ 650 000 грн	3000 – 4000 грн
---	-----------------	--------------	---------------	-----------------

Далі визначимо оптимальної системи збуту в таблиці 4.20:

- Проводити збут власними силами або залучати сторонніх посередників (власна або залучена система збуту);
- Вибір та обґрунтування оптимальної глибини каналу збуту;
- Вибір та обґрунтування виду посередників.

Таблиця 4.20. Формування системи збуту

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Покупка підписки на певну кількість використань	Підтримка існуючого функціоналу, подальший розвиток, аналіз відгуків	Канал однорівневий	Право власності залишається у розробника, тобто, вертикальна система збуту

Останньою частиною маркетингової програми є розробка концепції маркетингових комунікацій, вона наведена в таблиці 4.21.

Таблиця 4.21. Концепція маркетингових комунікацій

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Концепція реklamного звернення
1	Мінімізація витрат	Професійні конференції, презентації	Покупка підписки на	Таргетована реклама

			певну кількість використань	
--	--	--	--------------------------------	--

4.6 Висновки до розділу

В даному розділі було описано розробку стартап-проекту на основі теми дослідження. В результаті роботи було виконано наступні основні завдання:

1. Сформовано ідею проекту, яка є актуальною та релевантною для ринку як сьогодні, так і в перспективі;
2. Проведено технологічний аудит проекту;
3. Проаналізовано ринкові можливості;
4. Розроблено ринкову стратегію проекту;
5. Розроблено маркетингову програму проекту.

В рамках першого завдання окрім визначення і формування ідеї проекту, також було проведено аналіз слабких, нейтральних та сильних сторін проекту та в результаті було доведено що проект є актуальним та конкурентоспроможним на ринку.

Під час проведення технологічного аудиту проекту було розглянуто необхідні для розробки технології та проаналізовано їх доступність. В результаті дослідження було зроблено висновок, що всі необхідні технології є доступними та наявними.

Наступним етапом був аналіз ринкових можливостей. В даному підрозділі було виконано наступні важливі завдання:

- Досліджено характеристики ринку стартап проекту та потенційних клієнтів;
- Сформовано фактори загроз, можливостей та конкурентоспроможності;

- Проведено аналіз конкуренції на ринку;
- Проаналізовано конкуренцію в галузі за Портером
- Проведено порівняльний аналіз сильних та слабких сторін;
- Було зроблено SWOT-аналіз проекту;
- Досліджено альтернативи ринкового впровадження та обрано найбільш перспективну альтернативу.

Під час розробки ринкової стратегії проекту було обрано цільові групи користувачів, визначено базову стратегію розвитку та конкурентної поведінки.

Заключним етапом розробки стартап-проекту була розробка маркетингової стратегії проекту. На цьому етапі було визначено ключові переваги концепції потенційного товару, розроблено трирівневу модель товару, ухвалено межі встановлення ціни, сформовано систему збуту та встановлено концепцію маркетингових комунікацій.

ВИСНОВКИ

Для виконання основної мети даної роботи, а саме розробки ефективного методу перевірки програмного коду на плагіат, було проведено як попереднє вивчення та дослідження теоретичного матеріалу, так і апробація запропонованих методів на практиці.

В рамках теоретичного дослідження було досліджено загальне поняття плагіату, описано види запозичень та вивчено статистичні дані щодо кількісної оцінки наявності плагіату в роботах студентів за останні роки.

Оскільки робота присвячена виявленню плагіату саме програмного коду, то було детально розглянуто класифікації типів плагіату вихідного програмного коду за Беллоном, Роєм та Зельцером та приділено увагу загальній класифікації методів пошуку фрагментів програмного коду. Окрім того, було розглянуто існуючі інструменти для перевірки на плагіат та було порівняно їх функціональні можливості. В даній перевірці порівнювались системи MOSS, JPlag, Codequiry та Plagie. В якості характеристик порівняння було обрано наступні параметри:

1. Зручність використання;
2. Можливість виключення шаблонного коду;
3. Масштабованість;
4. Демонстрація результатів;
5. Можливість використання у якості веб-сервісу;
6. Наявність коду системи у відкритому доступі;
7. Кількість доступних мов програмування.

За результатами даного порівняння, було зроблено висновок, що система MOSS має всі характеристики, за якими проводилось порівняння, на високому рівні. Також було практично проаналізовано ефективність

системи MOSS в порівнянні з іншою популярною системою JPlag. Тестовими даними були приклади, що демонструють чотири типи плагіату. Згідно з результатів даного тестування, система MOSS показала кращі показники ніж JPlag. Оскільки система MOSS показала себе на досить високому рівні в порівнянні з іншими доступними інструментами, то було більш детально розглянуто дану систему, алгоритм її роботи та проаналізовано можливі вразливості даного інструменту. В ході аналізу існуючих публікацій, було знайдено публікацію [22], яка описує значну вразливість системи MOSS – вразливість алгоритму winnowing до порушення хешування. Автори роботи [22] розробили систему Mossad, яка здатна генерувати варіації сирцевого коду, не змінюючи його семантичного змісту та доступні системи, а саме MOSS, JPlag, Sherlock не будуть виявляти плагіат під час перевірки. Mossad використовує вищезазначену вразливість та за допомогою додавання декількох рядків, які не впливають на поведінку вихідного програмного коду та його семантичне значення, створює проблему порушення хешування. З цього можна зробити висновок, що система MOSS не є дуже надійною та є необхідність знайдення альтернативного рішення, яке не буде чутливо до подібної вразливостей.

В якості альтернативних методів пошуку плагіату в програмному коді пропонуються дві моделі: багатошаровий перцептрон + Code2Vec та TreeBERT. Після реалізації обох методів, було проведено порівняльний аналіз з метою виявлення кращої альтернативи для використання у системі виявлення плагіату, що розробляється. Обидва методи були протестовані на 5 прикладах: чотири типи плагіату та код, який демонструє проблему порушення хешування. Для наочності, на цих самих прикладах було протестовано і систему MOSS. За результатами даної перевірки можна побачити зайве підтвердження тому, що система MOSS не виявляє плагіату при порушенні хешування, в той самий час як обидва запропонованих

рішення визначають схожість в більш ніж 90% для того самого прикладу. Але варто відмітити, що розроблені методи на даний момент більш чутливі до зміни назв змінних, ніж MOSS через відсутність на даний момент етапу «попередньої компіляції» як в системі MOSS, яка перетворює все до «загального вигляду», ігноруючи назви змінних. Отже, в якості найкращої альтернативи для подальшого використання в системі було обрано багатоваріантний перцептрон + Code2Vec, з таблиці 3.1 можна побачити що в деяких випадках результати точніші на 5-7%.

В результаті роботи, було запропоновано архітектуру системи у вигляді блок-схеми та HLD діаграми, виділено та розглянуто ключові бізнес-функції системи, а саме:

- Авторизація;
- Обслуговування документів;
- Порівняння документів;
- Репрезентація Даних та Інтерфейс;
- API шлюз.

Базовий інтерфейс користувача для демонстрації роботи розробленого методу було реалізовано на мові програмування Python з використанням фреймворку PyQt5. Імплементация моделей також була реалізована на мові програмування Python та з використанням можливостей бібліотеки Tensorflow. Варто зазначити, що на даному етапі, перевірка на плагіат відбувається лише для коду, написаному на мові Java, але додання підтримки інших мов програмування не приведе до внесення змін в код моделі. Необхідними умовами для підтримки інших мов програмування є наявність вибірки даних для навчання та написання парсеру для необхідної мови.

На вхід системи можна як завантажити файли, так і ввести програмний код вручну. Після надання необхідних вхідних даних до системи, необхідно натиснути кнопку для проведення перевірки на наявність плагіату. Після проведення аналізу, відповідь моделі буде відображена у полі результатів. Для запобігання внесення змін до результатів перевірки, поле результатів доступне лише у режимі читання.

Отже, за результатами проведеної роботи, можна зробити висновок, що в даній роботі було розроблено метод для перевірки програмного коду на плагіат, який окрім успішного виявлення чотирьох типів плагіату, являється стійким до такої вразливості як проблема порушення хешування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is plagiarism? [Електронний ресурс] Режим доступу до ресурсу : <https://www.plagiarism.org/article/what-is-plagiarism> (дата звернення: 10.05.2022)
2. Plagiarism [Електронний ресурс] Режим доступу до ресурсу : <https://www.ox.ac.uk/students/academic/guidance/skills/plagiarism> (дата звернення: 18.05.2022)
3. What is a Transformer? [Електронний ресурс] Режим доступу до ресурсу: <https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04> (дата звернення: 20.04.2022)
4. A Deep Dive Into the Transformer Architecture – The Development of Transformer Models [Електронний ресурс] Режим доступу до ресурсу: <https://towardsdatascience.com/a-deep-dive-into-the-transformer-architecture-the-development-of-transformer-models-acbdf7ca34e0> (дата звернення: 20.04.2022)
5. The Illustrated Transformer [Електронний ресурс] Режим доступу до ресурсу: <https://jalammar.github.io/illustrated-transformer/> (дата звернення: 22.04.2022)
6. The Transformer Model [Електронний ресурс] Режим доступу до ресурсу: <https://machinelearningmastery.com/the-transformer-model/> (дата звернення: 25.04.2022)
7. Stephen F. Hard, Jim Conway, Faculty and College Student Beliefs about the Frequency of Student Academic Misconduct, 2006.
8. Jason MacGregor, Martin Stuebs. To Cheat or Not to Cheat: Rationalizing Academic Impropriety, 2012.

9. Stephens, J. M. How to Cheat and Not Feel Guilty: Cognitive Dissonance and its Amelioration in the Domain of Academic Dishonesty. *Theory Into Practice*, 56(2), 111-120, 2017.

10. Paul W. Grimes. *Dishonesty in Academics and Business: A Cross-Cultural Evaluation of Student Attitudes*, 2004. [Електронний ресурс] Режим доступу до ресурсу:

https://www.researchgate.net/publication/226196400_Dishonesty_in_Academics_and_Business_A_Cross-Cultural_Evaluation_of_Student_Attitudes (дата звернення: 10.04.2022)

11. О.С. Ковальчук, Л.М. Потапюк. *Забезпечення академічної чесності у вищих навчальних закладах: міжнародний досвід*, 2017.

[Електронний ресурс] Режим доступу до ресурсу: <https://pedpsy.duan.edu.ua/images/PDF/2017/1/4.pdf> (дата звернення: 10.04.2022)

12. Uri Alon, Meital Zilberstein. *Code2vec: Learning Distributed Representations of Code*, 2018. [Електронний ресурс] Режим доступу до ресурсу: <https://arxiv.org/pdf/1803.09473.pdf> (дата звернення: 01.05.2022)

13. Abstract Syntax Tree [Електронний ресурс] Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Abstract_syntax_tree (дата звернення: 03.05.2022)

14. Abstract Syntax Tree (AST) in Java. [Електронний ресурс] Режим доступу до ресурсу: <https://www.geeksforgeeks.org/abstract-syntax-tree-ast-in-java/> (дата звернення: 03.05.2022)

15. AST (Abstract Syntax Tree) [Електронний ресурс] Режим доступу до ресурсу: <https://medium.com/@dinis.cruz/ast-abstract-syntax-tree-538aa146c53b> (дата звернення: 03.05.2022)

16. Lutz Prechelt, Guido Malpohl. Finding Plagiarisms among a Set of Programs with JPlag, 2003. [Электронный ресурс] Режим доступа до ресурсу:
https://www.researchgate.net/publication/2832828_Finding_Plagiarisms_among_a_Set_of_Programs_with_JPlag (дата звернення: 04.05.2022)
17. How to use JPlag. [Электронный ресурс] Режим доступа до ресурсу:
<https://github-wiki-see.page/m/jplag/JPlag/wiki/1.-How-to-Use-JPlag> (дата звернення: 04.05.2022)
18. Code Plagiarism & Similarity API. [Электронный ресурс] Режим доступа до ресурсу: <https://codequiry.com/usage/api> (дата звернення: 04.05.2022)
19. Moss – A System for Detecting Software Similarity. [Электронный ресурс] Режим доступа до ресурсу: <https://theory.stanford.edu/~aiken/moss/> (дата звернення: 04.05.2022)
20. Saul Schleimer, Daniel S. Wilkerson. Winnowing: Local Algorithms for Document Fingerprinting. [Электронный ресурс] Режим доступа до ресурсу: <https://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf> (дата звернення: 05.05.2022)
21. How MOSS works. [Электронный ресурс] Режим доступа до ресурсу: <https://yangdanny97.github.io/blog/2019/05/03/MOSS> (дата звернення: 05.05.2022)
22. Breanna Devore-McDonald, Emery D. Berger. Mossad: Defeating Software Plagiarism Detection, 2020. [Электронный ресурс] Режим доступа до ресурсу: <https://arxiv.org/pdf/2010.01700.pdf> (дата звернення: 10.05.2022)
23. Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with JPlag. Journal of Universal Computer Science, 8(11):1016–1038, 2002.

24. Ashish Vaswani, Noam Shazeer. Attention is All You Need, 2017. [Електронний ресурс] Режим доступу до ресурсу: <https://arxiv.org/pdf/1706.03762.pdf> (дата звернення: 01.05.2022)
25. Code2Vec. [Електронний ресурс] Режим доступу до ресурсу: <https://www.ponicode.com/blog/code2vec> (дата звернення: 11.05.2022)
26. Xue Jiang, Zhuoran Zheng. TreeBERT: A Tree-Based Pre-Trained Model for Programming Language, 2021. [Електронний ресурс] Режим доступу до ресурсу: <https://arxiv.org/pdf/2105.12485.pdf> (дата звернення: 11.05.2022)
27. Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2016. [Електронний ресурс] Режим доступу до ресурсу: <https://arxiv.org/pdf/1508.07909.pdf> (дата звернення: 12.05.2022)
28. Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning, 2017. [Електронний ресурс] Режим доступу до ресурсу: <https://arxiv.org/pdf/1705.03122.pdf> (дата звернення: 12.05.2022)
29. Оцінка ідентичності програмного коду. [Електронний ресурс] Режим доступу до ресурсу: <https://annimon.com/article/3730> (дата звернення: 12.04.2022)
30. Cyber Plagiarism & Statistics. [Електронний ресурс] Режим доступу до ресурсу: <https://www.checkforplagiarism.net/cyber-plagiarism> (дата звернення: 25.04.2022)
31. Ethan Baron. Stanford, UC Berkeley computer science cheating incidents part of widespread problem: report, 2017.
32. Рівень плагіату в студентських роботах у країнах Європи. Блог Романа Радейка. [Електронний ресурс] Режим доступу до ресурсу:

<http://www.aphd.ua/riven-plahiatu-v-studentskykh-robotakh-u-kranakh-ievropy-/> (дата звернення: 26.04.2022)

33. Aditiya Kanade, Petros Maniatis. Learning and Evaluating Contextual Embedding of Source Code, 2020. [Електронний ресурс] Режим доступу до ресурсу: <https://arxiv.org/pdf/2001.00059v3.pdf> (дата звернення: 15.05.2022).

34. Upul Bandara. A Machine Learning Based Tool for Source Code Plagiarism Detection, 2011. [Електронний ресурс] Режим доступу до ресурсу: <http://www.ijmlc.org/papers/50-A243.pdf> (дата звернення: 12.04.2022)

35. Code2Vec. [Електронний ресурс] Режим доступу до ресурсу: <https://code2vec.org/>. (дата звернення: 25.04.2022)