Вінницький національний технічний університет
Факультет інформаційних технологій та комп’ютерної інженерії
Кафедра комп’ютерних наук

Пояснювальна записка

до магістерської кваліфікаційної роботи

на тему «Інформаційна технологія виявлення дублікатів програмного коду»

Виконав: студент 2 курсу,
групи 1КН-18 м
спеціальності 122 «Комп'ютерні науки»
Никуляк А. В.
Керівник: к.т.н., доц. Месюра В.І.
Рецензент: к.т.н., доц. Романюк О.В.

Вінниця - 2019 року

ЗАТВЕРДЖУЮ

Завідувач кафедри____КН____
д.т.н., проф.. Яровий А.А.

_____
(підпис)
“____” _____2019 року

ЗАВДАННЯ

на магістерську кваліфікаційну роботу на здобуття кваліфікації магістра зі спеціальності: 122 – «Комп'ютерні науки»

08-22.МКР.013.18.000.ПЗ

Магістранта групи 1КН-18м Никуляка Артема Вікторовича

Тема магістерської кваліфікаційної роботи: «Інформаційна технологія виявлення дублікатів програмного коду»

Вхідні дані: об’єктно – орієнтована мова програмування; обсяг навчальної вибірки 700 елементів; обсяг тестової вибірки 500 елементів; точність виявлення дублікатів 80%.

Короткий зміст частин магістерської кваліфікаційної роботи:

1. Графічна: Загальна структура інформаційної технології виявлення дублікатів програмного коду; архітектура нейронної мережі; діаграма класів програми; результати тестування інформаційної технології; робочі вікна програми виявлення дублікатів програмного коду.
2. Текстова (пояснювальна записка): вступ, аналіз сучасного рівня розвитку інформаційної технології виявлення дублікатів програмного коду, розробка інформаційної технології виявлення дублікатів програмного коду, програмна реалізація інформаційної технології виявлення дублікатів програмного коду, економічна частина, висновки, перелік використаних джерел, додатки.

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ МКР

№ етапу	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	Аналіз сучасного рівня інформа-ційних технологій виявлення дублікатів програмного коду			Аналітичний огляд літературних джерел, задачі досліджень, розділ 1 ПЗ
2	Розробка методу та інформаційної технологіївиявлення дублікатів програмного коду.			Метод, інформаційна технологія , розділ 2
3	Програмн реалізація розробленої інформаційної технології, тестування та оцінка параметрів.			Програмне забезпечення,розділ 3
4	Підготовка економічної частини			розділ 4
5	Апробація та/або впровадження результатів дослідження			тези доповідей/акт впровадження
6	Оформлення пояснювальної записки, графічного матеріалу та презентації			Пояснювальна записка, графічний матеріал, презентація

Консультанти з окремих розділів магістерської кваліфікаційної роботи

1. Науковий керівник _____ канд. техн. наук, проф., проф. кафедри КН
(підпис) наук. ступінь, вчене звання (посада)
“____” _____20___ р. _____В. І. Месюра_____
ініціали та прізвище

2. Економічна частина _____ старший викладач кафедри ЕПВМ
(підпис) наук. ступінь, вчене звання (посада)
_____
ініціали та прізвище
“____” _____20___ р.

Дата попереднього захисту роботи “____” _____20___ р.

Рецензент _____ канд. техн. наук, доц., доц. кафедри ПЗ
(підпис) наук. ступінь, вчене звання (посада)
_____О.В. Романюк_____
ініціали та прізвище

Завдання видав
науковий керівник _____ канд. техн. наук, проф., проф. кафедри КН
(підпис) наук. ступінь, вчене звання (посада)
_____В. І. Месюра_____
ініціали та прізвище
“____” _____20___ р.

Завдання отримав магістрант _____ _____А.В. Никуляк_____
(підпис) ініціали та прізвище
“____” _____20___ р.

РЕФЕРАТ

Магістерська кваліфікаційна робота присвячена дослідженню застосування штучних нейронних мереж в задачах виявлення дублікатів в вихідному коді програм. Дослідження проведено на основі проектів з відкритим вихідним кодом. В ході даного дослідження розроблено інформаційну технологію виявлення дублікатів програмного коду. Розроблена технологія протестована на спеціальній вибірці - BigCloneBench. Тестування показують доцільність використання методу виявлення дублікатів.

ABSTRACT

This qualification work is devoted to the study of the use of artificial neural networks in the problems of detection of duplicates in the source code of programs. The study was conducted on the basis of open source projects. In the course of this study, information technology for detecting duplicates of program code was developed. The technology developed was tested on a special sample - BigCloneBench. Testing has shown the feasibility of using the duplicate detection method.

ЗМІСТ

ВСТУП 8
1 АНАЛІЗ СУЧАСНОГО РІВНЯ РОЗВИТКУ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ ВИЯВЛЕННЯ ДУБЛІКАТІВ ПРОГРАМНОГО КОДУ 13
1.1 Аналіз предметної області виявлення дублікатів програмного коду 13
1.2 Дослідження відомих методів виявлення дублікатів програмного коду .. 18
1.3 Аналітичний огляд відомих програмних реалізацій методів виявлення дублікатів програмного коду 21
1.4 Обґрунтування вибору методу виявлення дублікатів програмного коду . 23
1.5 Постановка задачі виявлення дублікатів програмного коду 26
1.4 Висновок 27
2 РОЗРОБКА ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ ВИЯВЛЕННЯ ДУБЛІКАТІВ ПРОГРАМНОГО КОДУ 27
2.1 Розробка подання вихідного коду за допомогою абстрактного синтаксичного дерева 28
2.2 Обґрунтування вибору виду нейронної мережі для методу виявлення дублікатів програмного коду 32
2.3 Обґрунтування використання рекурентної нейронної мережі 33
2.4 Розробка алгоритму гібридного методу виявлення дублікатів програмного коду 35
2.4.1 Передобробка 38
2.4.2 Перетворення 38
2.4.3 Навчання і використання нейронної мережі 39
2.4.4 Сіамська архітектура нейронної мережі 40
2.4.5 Модель подання Sequence-to-Sequence 43
2.4.6 Постобробка 44

2.5 Висновок 45
3 ПРОГРАМНА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ ВИЯВЛЕННЯ ДУБЛІКАТІВ ПРОГРАМНОГО КОДУ 46
3.1 Обґрунтування вибору мови та середовища програмування інформаційної технології виявлення дублікатів програмного коду 46
3.2 Програмна реалізація інформаційної технології виявлення дублікатів програмного коду 48
3.2.1 Реалізація модуля побудови AST 49
3.2.2 Реалізація модуля виявлення дублікатів 51
3.3 Тестування та аналіз результатів роботи інформаційної технології 54
3.4 Висновок 61
4 ЕКОНОМІЧНА ЧАСТИНА 62
4.1 Оцінювання комерційного потенціалу розробки 62
4.2 Прогнозування витрат на виконання науково-дослідної, дослідно-конструкторської та конструкторсько-технологічної роботи 63
4.3 Прогнозування комерційних ефектів від реалізації результатів розробки 67
4.4 Розрахунок ефективності вкладених інвестицій та періоду їх окупності 68
4.5 Висновок 69
ВИСНОВКИ 72
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 73
Додаток А Інструкція користувача 76
Додаток Б Лістинг програми 78
Додаток В Графічна частина 95

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ	Програмне забезпечення
ШНМ	Штучна нейронна мережа
API	Application Program Interface
AST	Abstract Syntax Tree
RNN	Recurrent neural network
LSTM	Long Short-Term Memory
XML	eXtensible Markup Language
CBOW	Continuous Bag of Words
PDG	Program Dependency Graph

ВСТУП

Актуальність теми дослідження. Однією з актуальних проблем розробки і супроводу програмного забезпечення є наявність дублікатів в вихідному коді програми. Фрагмент коду, який був скопійований і вставлений з незначними або істотними змінами називається дублікатом ПЗ або дублікатом. За даними різних досліджень сучасне ПЗ нараховує до 30% програмних дублікатів. Дублікати можуть з'являтися в програмних системах з різних причин. Однією з причин може бути копіювання ділянок коду. Такий підхід може стати розумною відправною точкою для написання програми. Інша причина – розробка вже існуючої ділянки коду іншим програмістом. Також причиною появи дублікатів може служити невиразність мови програмування чи API. Серед багатьох проблем, пов'язаних з наявністю дублікатів в вихідному коді програми, однією з головних є її подальший супровід. У тому випадку, коли є помилка в одному з дубльованих фрагментів, її необхідно знайти і справити в усіх ідентичних ділянках, а не тільки в одній. Виявлення дублікатів є важливою проблемою для підтримки ПЗ.

Зв'язок роботи з науковими програмами, планами, темами. Магістерська робота виконана відповідно до напрямку наукових досліджень кафедри комп’ютерних наук Вінницького національного технічного університету, теми кафедральної науково-дослідної роботи 22 К1 «Моделі, методи технології та пристрої інтелектуальних інформаційних систем управління, економіки, навчання та комунікацій» і плану наукової та навчально-методичної роботи.

Мета і задачі дослідження. Основною метою магістерської кваліфікаційної роботи є підвищення точності виявлення дублікатів програмного коду за рахунок розробки інтелектуального методу.

Виходячи з цього у межах даного дослідження необхідно розв’язати такі основні задачі:

- аналіз предметної області;
- розглянути існуючі методи виявлення дублікатів програмного коду;
- на основі існуючих методів розробити модифікований метод виявлення дублікатів програмного коду;
- програмна реалізація інформаційної технології виявлення дублікатів програмного коду;
- тестування та аналіз результатів тестування інформаційної технології.

Об’єкт дослідження – процес виявлення дублікатів програмного коду.

Предмет дослідження – методи виявлення дублікатів програмного коду та їх властивості.

Методи дослідження. В роботі використано теоретичні та змішані методи наукових досліджень. Використано загальні методи виявлення дублікатів тексту на основі нейронних мереж, методи на основі абстрактних синтаксичних дерев, токенізація тексту, а також методи об’єктно-орієнтованого програмування для програмної реалізації запропонованої інформаційної технології виявлення дублікатів програмного коду.

Наукова новизна одержаних результатів:

- Вперше запропоновано інформаційну технологію виявлення дублікатів програмного коду, яку засновано на гібридному методі виявлення дублікатів коду, щл забезпечило підвищення точності виявлення дублікатів за рахунок подання вихідного коду програм у вигляді абстрактного синтаксичного дерева і виявлення однакових фрагментів у створених поданнях за допомогою рекурентних нейронних мереж, який за рахунок забезпечив покращення повноти виявлення дублікатів.

- Дістав подальшого розвитку метод виявлення текстових дублікатів на основі рекурентних нейронних мереж, який за рахунок використання сіамської архітектури нейронної мережі, дав змогу порівнювати вже опрацьовані мережею дані з новими.

Практичне значення одержаних результатів полягає у наступному:

1. Розроблено алгоритм гібридного гібридного методу виявлення дублікатів прогамного коду, які забезпечують підвищення точності процесу виявлення дублікатів.

2. Створено функціонал конверсії програмного коду в структуру абстрактного синтаксичного дерева для створення вхідних параметорів нейронної мережі.

3. Здійснено програмну реалізацію інформаційної технологій виявлення дублікатів програмного коду.

Достовірність теоретичних положень магістерської кваліфікаційної роботи підтверджується строгістю постановки задач, коректним застосуванням математичних методів під час доведення наукових положень, строгим виведенням аналітичних співвідношень, порівнянням результатів з відомими, та збіжністю результатів математичного моделювання з результатами, що отримані під час впровадження розроблених програмних засобів.

Особистий внесок магістранта. Всі дослідження, представлені в магістерській кваліфікаційній роботі, проведені особисто. У тезах, здобувачу належать: [1] – запропонований гібридний метод виявлення дублікатів програмного коду за допомогою рекурентної нейронної мережі та абстрактного синтаксичного дерева; розробка алгоритму методу.

Апробація результатів роботи. Результати роботи представлені на всеукраїнській конференції «Інформаційні технології та взаємодії», (м. Київ, Україна 2019 р).

Публікації. За результатами досліджень тези доповіді науково-технічної конференції [1].

1 АНАЛІЗ СУЧАСНОГО РІВНЯ РОЗВИТКУ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ ВИЯВЛЕННЯ ДУБЛІКАТІВ ПРОГРАМНОГО КОДУ

1.1 Аналіз предметної області виявлення дублікатів програмного коду

Дослідження з обслуговування програмного забезпечення показали, що багато програм містять значну кількість дублюваного коду. Такий код вважається шкідливим через дві причини: множинні, можливо, непотрібні, дублікати, які збільшують обсяги коду та витрати на обслуговування та несумісні зміни в дублікатованому коді, які можуть створювати помилки і, отже, призводити до неправильної поведінки програми. Хоча в останні роки виявлення дублікатів було дуже активною областю досліджень, дотепер не існує глибокого розуміння ступеня шкідливості дублюваннявання коду. Насправді, деякі дослідники навіть почали сумніватися у шкідливості дублікатування взагалі [2]. Важливо розуміти, що дублікати безпосередньо не викликають несправностей але непослідовні зміни дублікатів можуть призвести до несподіваної поведінки програми. Особливо небезпечний тип змін до дублікатованого коду – непослідовне виправлення помилок.

Виявлення дублікату коду є важливою проблемою для програмного забезпечення обслуговування та розвитку. Багато підходів враховують структуру, або ідентифікатори, але жоден із існуючих методів виявлення не моделює обидва джерела інформації.

Ці методи також залежать від особливостей, що представляють фрагменти коду. Відповідно, сховища програмного коду є абстракціями, які дають інженерам-програмістам можливість керувати складністю, відокремлюючи проблеми та керувати різними деталями на різних рівнях.

Абстракції на всіх рівнях деталізації доповнюються реалізаціями. Ці реалізації можуть бути розроблені з нуля, або вони можуть дублікатувати з існуючих фрагментів коду. Якщо існує код, що забезпечує вихідний пункт для впровадження, тоді інженер програмного забезпечення може дублювати код шляхом копіювання та додавання фрагмента.

Ще один спосіб введення дублікатів в програмну систему – це коли інженер несвідомо розробляє реалізацію, схожу на існуючу. Копіювання та вставлення коду та подальші модифікації скопійованого фрагменту можуть отримати текстово подібні фрагменти коду, де подібності можуть бути охарактеризовані синтаксисом. З іншого боку, коли інженер несвідомо розробляє реалізацію, подібну до чогось, що вже існує, він може створити такі дублікати функціонально схожийми, але синтаксично інший ми.

Програмісти створюють і підтримують кодові дублікати в програмному забезпеченні. Попередні дослідження показують, що кодові дублікати є поширеними у програмних системах. Наприклад, на 19% дублюється код у Windows System, та 15-25% дублюється код у ядрі Linux [3]. Недавнє дослідження для 7800 проектів з відкритим кодом, написаних на C або C ++, це показує 44%, що проектів мають принаймні одну пару однакового фрагменту коду.

Копіюючи та вставляючи код з або без модифікацій, розробники повторно використовують існуючий код для підвищення продуктивності програмування. Сучасні проблеми щодо обслуговування програмного забезпечення можуть вимагати послідовне застосування одних і тих же або подібних виправлень помилок або зміни в декількох місцях коду.

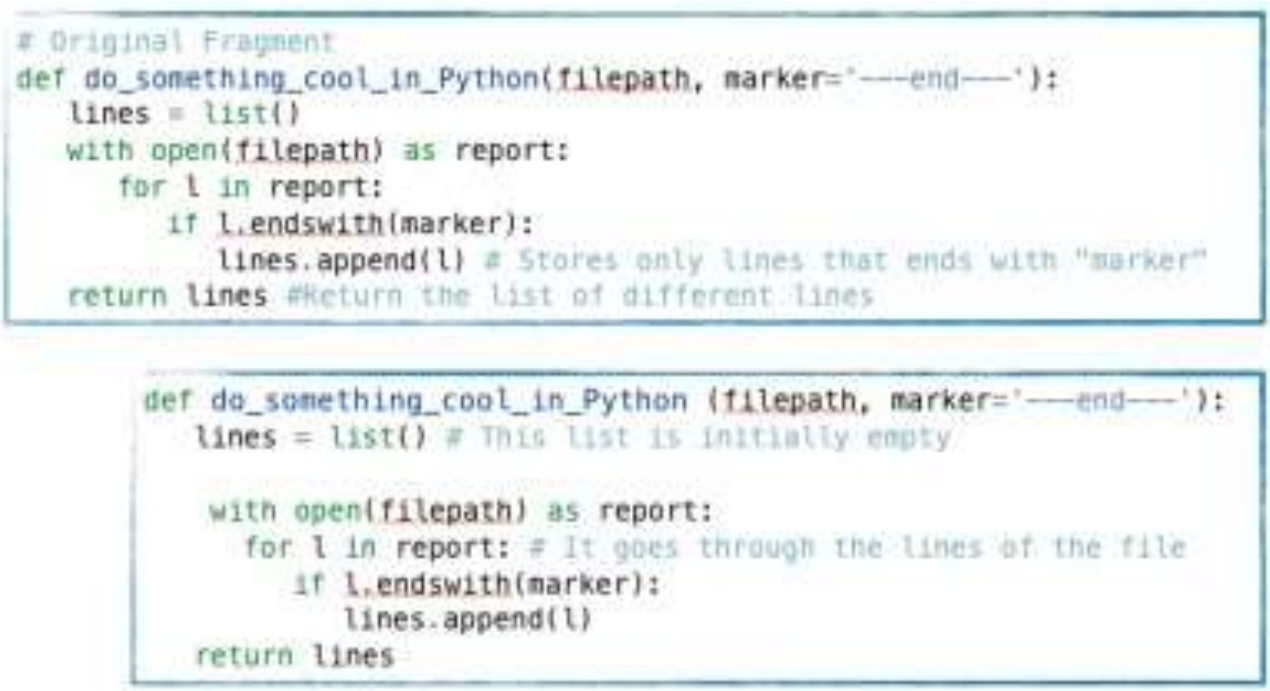
З точки зору ідентичності дубльованих фрагментів, дублікати діляться на наступні типи:

- I: Повністю ідентичні фрагменти програми с точністю до розмітки і коментарів.

- II: Дублікати, в яких не враховуються відмінності ідентифікаторів, типів і літералів в доповненні до змін, що враховуються в першому типі.

- III: Синтаксично схожі фрагменти, що розрізняються на рівні операторів, тобто оператори можуть бути додані, змінені або видалені на додаток до змін, що враховуються під другому типі.

- VI: Фрагменти програми, вирішальні схожу задачу, але реалізовані різними способами (семантичні дублікати).



```
# Original Fragment
def do_something_cool_in_Python(filepath, marker='---end---'):
    lines = list()
    with open(filepath) as report:
        for l in report:
            if l.endswith(marker):
                lines.append(l) # Stores only lines that ends with "marker"
    return lines #Return the list of different lines
```

```
def do_something_cool_in_Python (filepath, marker='---end---'):
    lines = list() # This list is initially empty

    with open(filepath) as report:
        for l in report: # It goes through the lines of the file
            if l.endswith(marker):
                lines.append(l)
    return lines
```

Рисунок 1.1 – Дублікат програмного коду типу I

```
# Original Fragment
def do_something_cool_in_Python(filepath, marker='---end---'):
    lines = list()
    with open(filepath) as report:
        for l in report:
            if l.endswith(marker):
                lines.append(l) # Stores only lines that ends with "marker"
    return lines #Return the list of different lines
```

```
# Type 2 Clone
def do_something_cool_in_Python(path, end='---end---'):
    targets = list()
    with open(path) as data_file:
        for t in datae:
            if l.endswith(end):
                targets.append(t) # Stores only lines that ends with "marker"
    #Return the list of different lines
    return targets
```

Рисунок 1.2 – Дублікат програмного коду типу II

```
import os
def do_something_with(path, marker='---end---'):
    # Check if the input path corresponds to a file
    if not os.path.isfile(path):
        return None
    bad_ones = list()
    good_ones = list()
    with open(path) as report:
        for line in report:
            line = line.strip()
            if line.endswith(marker):
                good_ones.append(line)
            else:
                bad_ones.append(line)
     #Return the lists of different lines
    return good_ones, bad_ones
```

Рисунок 1.3 – Дублікат програмного коду типу III

```
# Original Fragment
def do_something_cool_in_Python(filepath, marker='---end---'):
    lines = list()
    with open(filepath) as report:
        for l in report:
            if l.endswith(marker):
                lines.append(l) # Stores only lines that ends with "marker"
    return lines #Return the list of different lines
```

```
def do_always_the_same_stuff(filepath, marker='---end---'):
    report = open(filepath)
    file_lines = report.readlines()
    report.close()
    #Filters only the lines ending with marker
    return filter(lambda l: len(l) and l.endswith(marker), file_lines)
```

Рисунок 1.4 – Дублікат програмного коду типу IV

При розгляді дублікатів з точки зору їх розміру, виділяють наступні види:

- З фіксованою гранулярністю: ідентичні фрагменти коду фіксованого розміру (методи класи і т.д.).
- З довільної гранулярністю: ідентичні фрагменти довільного розміру

Повторне використання коду, логіки, дизайну або всієї системи є основними причинами виникнення дублікатів коду. Можна виділити два види повторного використання коду:

Повторне використання коду за допомогою копіювання чи вставки (з незначними змінами або без них) найпростіший і найпоширеніший вид механізму повторного використання в процесі розробки. Такий спосіб є швидким способом повторного використання надійних синтаксичних і семантичних конструкцій.

Відгалуження. Термін «відгалуження» позначає повторне використання подібних рішень з їх можливим розбіжністю. Наприклад, при розробці драйвера для сімейства апаратних пристроїв, схоже сімейство пристроїв може вже мати драйвер. Таким чином, цей драйвер може бути використаний з незначними змінами.

Аналогічно, дублікати можуть з'являтися при перенесенні ПО з однієї платформи на іншу.

Дублікати, також можуть з'являтися через використовуваного підходу. наприклад: Злиття двох схожих систем. Іноді дві програмні системи зі схожою функціональністю об'єднуються з метою створення нової. Також, такі системи можуть бути розроблені різними командами, і, тоді, дублікати при об'єднанні з'являються через реалізацію схожих функциональностей в обох системах.

Розробка системи за допомогою генеративного підходу. Генерація коду може породити величезну кількість дублікатів через використання однакових шаблонів для генерації однакової або схожі логіки.

Ідентифікація пар, що майже повторюються, у великій колекції коду є суттєвою проблемою для програм великого обсягу. Два вихідні коди майже повторюються, якщо вони відрізняються один від одного в дуже невеликій пропорції.

Проблема виявлення наближеного дублювання добре вивчена для документів та веб-сторінок. Одним із популярних підходів є такий, де кожен документ розбивається на фрагменти, що перекриваються. Для кожного документа отримуються всі послідовності сусідніх слів. Якщо два документи містять однаковий набір, їх можна визначити як майже дублікати.

Заснований на співвідношенні величини шифтового з'єднання і перетину, обчислюється ступінь, того як два документи схожі один на одного. Інший популярний підхід до виявлення майже дублюючих документів заснований на створенні підписів з n-грамів.

Вибір n-грамів базується на методі, який поєднує в собі попередні стоп-слова з короткими ланцюжками суміжних змістових термінів.

До неідентичних дублікатів, можна віднести дублікати типу III.

Підрядковий код називається непослідовним дублікатом, якщо є ще однин підрядок t коду, що їх відстань редагування нижче заданого порогу, і t не має значного перекриття з s.

Відстань редагування – це показник, що підраховує кількість операцій редагування (вставка, видалення або зміна єдиного блоку), необхідних для перетворення однієї послідовності в іншу. Це визначення це залежить від обраного порогу та значення "значного перекриття". Однак визначення описує інтуїтивне розуміння непослідовного дублікату.

Група дублікатів може розглядатися як пов'язаний графік, де кожен вузол є підрядком, а ребра зображаються між підрядками, що є дублікатами один одного. Як мінімум одна пара непослідовних дублікатів знаходиться в групі, вона називається непослідовна група дублікатів.

Переваги виявлення дублікатів і їх подальшого рефакторинга є наступними:

Виявлення потенційних бібліотечних методів. Такий код може бути включений в бібліотеку і безперешкодно повторно використовуватися в подальшому.

Поліпшення наочності програми. У разі описаної функціональності програми, з'являється можливість створення повної картини про інші файли, що містять схожі копії цього фрагмента. Наприклад, якщо фрагмент коду відповідає за управління пам'яттю, можна зробити висновок, що всі файли, що містять копію, повинні мати реалізацію структури з динамічним розподілом пам'яті.

Виділення шаблонів. У тому випадку, коли всі дубльовані фрагменти однієї вихідної ділянки можуть бути знайдені, може бути виявлена функціональна схема використання цього фрагмента. Іншими словами, на основі такого фрагмента і його копій може бути виділений шаблон.

Пошук плагіату і порушення авторських прав. Пошук схожого коду може бути корисним в пошуку плагіату і порушення авторських прав.

Рефакторинг. Методи пошуку дублікатів дозволяють зробити вихідний код більш компактним

При аналізі підходів до виявлення дубльованих ділянок необхідно виділити основні характеристики для їх порівняння. Ключовими характеристиками методу можна вважати:

- Типи дублікатів для виявлення;
- Повнота одержуваних результатів;
- Точність отриманих результатів;
- Швидкість роботи.

Повнота результатів визначає відношення кількості дублікатів виявлених методом до загальної кількості дублікатів.

Точність результатів визначає відношення кількості виявлених дублікатів до кількості дійсних дублікатів.

Тому основна вимога до методу – виявлення дублікатів перших трьох типів.

1.2 Дослідження відомих методів виявлення дублікатів програмного коду

Як правило, процес виявлення дублікатів складається з двох етапів: трансформації та порівняння.

На першому етапі код перетвориться в проміжне внутрішнє подання, яке дозволяє використовувати більш ефективні і спеціалізовані алгоритми порівняння. У той же час, вибір проміжного подання накладає обмеження на алгоритми, що використовуються і визначає якість підсумкових результатів.

Методи пошуку дублікатів можуть бути класифіковані залежно від способу внутрішнього подання коду:

- Метод на основі аналізу тексту. Підходи такого типу виробляють малу (або не виробляють взагалі) кількість змін або нормалізації перед фактичним порівнянням.

У більшості випадків використовується безпосередньо вихідний код програми, представлений у вигляді послідовності рядків. В отриманій послідовності проводиться пошук однакових фрагментів найбільшої довжини. Пошук таких фрагментів здійснюється за допомогою методів аналізу даних або строкових алгоритмів.

До таких відносяться методи порівняння відбитків рядків, пошуку за шаблоном, знаходження частою послідовності. Однак, висока чутливість до змін у вихідній програмі є головним недоліком підходів такого типу. Без застосування модифікацій таке уявлення дозволяє виявляти тільки дублікати першого типу.

Для виявлення дублікатів другого типу необхідно застосовувати різного роду модифікації: літерали і ідентифікатори замінюються на спеціальні константи. завдяки описаним модифікаціям, при виявленні подібних фрагментів, такі відмінності не враховуються.

- Метод на основі аналізу токенів. У даному методі вихідний код обробляється і представляється у вигляді послідовності токенів (об'єктів, одержуваних в процесі лексичного аналізу). На наступному етапі застосовуються рядкові алгоритми для пошуку дублікатів в послідовності токенів.

На відміну від методу на основі аналізу текстів, такий метод дозволяє використовувати більш ефективні і стійкі до змін програми методи. Ефективність таких методів досягається за рахунок більш компактного внутрішнього подання, а їх стійкість – за рахунок фільтрації і нормалізації токенів.

- Метод на основі аналізу графів. Методи, засновані на аналізі графів основуються на абстракції вихідного коду за рахунок обліку семантичної інформації, що містяться в графах залежностей, захоплюючих інформацію

управління і потоку даних. Для пошуку схожих під-графів використовується алгоритм ізоморфізму для під-графів, застосовуваний на графі програмних залежностей (PDG).

Один з провідних способів пошуку дублікатів, заснованого на даному сімействі методів – пошук ізоморфних під-графів PDG за допомогою (зворотного) розбиття програми на елементи.

- Метод на основі аналізу синтаксичних дерев. В рамках даного методу вихідний код програми представляється у вигляді дерева розбору або абстрактного синтаксичного дерева (Abstract Syntax Tree - AST).

Такий підхід використовує структурну інформацію про програму, що дозволяє виявляти дублікати перших трьох типів. Однак, семантика програми не враховується, що призводить до неможливості виявлення дублікатів четвертого типу, а також дублікатів зі зміненим порядком операторів. Для виявлення дубльованих фрагментів коду за допомогою аналізу дерев, використовуються алгоритми пошуку однакових під дерев. Такі методи, як правило, базуються на алгоритмах динамічного програмування, або намагаються звести задачу до пошуку однакових підстрок.

Ще один метод, що відноситься до досліджуваних – додавання вузлів дерева метриками, які характеризують відповідні піддерева. Такий прийом сильно спрощує завдання, дозволяючи знайти рішення за час, пропорційний довжині вихідного коду.

При конвертації AST в XML (eXtensible Markup Language) і використанні технологій глибинного аналізу даних, з'являється можливість знаходження точних і параметризованих дублікатів на більш абстрактному рівні. Для уникнення складнощів, пов'язаних з повним порівнянням AST, під-дерева представляються у вигляді серіалізованих послідовностей токенів (суфіксного дерева), що дозволяє більш ефективно знаходити синтаксичні дублікати.

- Метод на основі програмних метрик Розглянутий метод заснований на зборі різних метрик, пов'язаних з фіксованими фрагментами коду. Такий метод дозволяє після збору метрик порівняти їх між собою, замість порівняння вихідного коду безпосередньо.

У запропонованих методах використовувалися різні програмні метрики для пошуку дублікатів. Наприклад, в якості метрики можна приймати кількість рядків вихідного коду (LOCs), кількість викликів функцій, кількість ребер графа потоку управління (CFG). Такі метрики розраховуються для кожного функціонального елемента програми. Ті елементи у яких схожі метрики вважаються дублікатами.

Розглянутий метод дозволяє відшукати дублікати на рівні функцій, але не справляється з пошуком дублікатів менших розмірів.

- Змішані методи. Крім розглянутих методів, існує така група, яка використовує змішаний підхід. Іншими словами, використовуються кілька з розглянутих вище структур для створення внутрішнього подання програми.

Одним з найпопулярніших гібридних підходів є отримання і серіалізація синтаксичного дерева в послідовність токенів. Таке змішування дозволяє аналізувати структурну інформацію, що отримується з AST, використовуючи ефективні рядкові алгоритми.

1.3 Аналітичний огляд відомих програмних реалізацій методів виявлення дублікатів програмного коду

Програмний засіб Deckard – це вдосконалений механізм аналізу на основі AST, який швидко виявляє повторювані шаблони в коді та дозволяє знаходити дублікати коду та виявиляє помилки копіювання та вставки.

Deckard – це технологія, яка поєднує в собі обмін даними з розумінням синтаксису та семантики коду. Результатом є глибоке відкриття шаблону та аналіз,

який допускає зміни функцій та змінних назв, а також вставлення та видалення операторів.

Система спочатку аналізує вихідний код, а потім здійснює створення AST. Усі подібні сегменти коду ідентифікуються, після чого виконується виявлення невідповідності. Результати зберігаються в базі даних для подальшого використання. Веб-інтерфейс підтримує ефективний перегляд та аналіз усіх виявлених дублікатів коду разом з потенційними дефектами. Інтерфейси командного рядка та API доступні для сценаріїв та індивідуальних інтеграцій. Вікно програми Deckard зображено на рисунку 1.5.

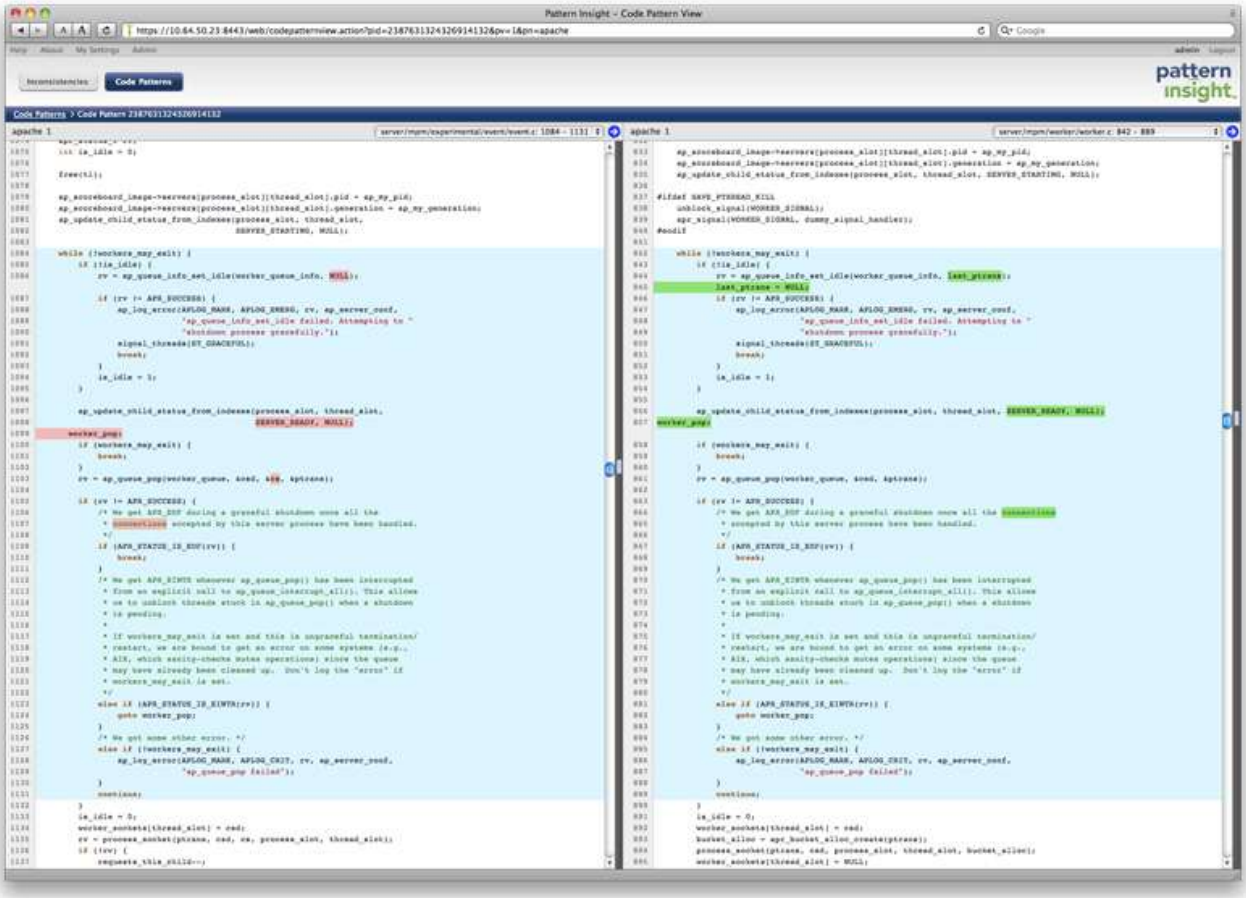


Рисунок 1.5 – Вікно програми Deckard

SourcererCC – це текстовий, незалежний від мови детектор дублювання коду, що використовує концепцію ланцюга дублювання та токенізації. Інструмент написаний на Java та працює на всіх основних платформах. На рисунку 1.6 зображено вікно програми SourcererCC під час сканування файлів.

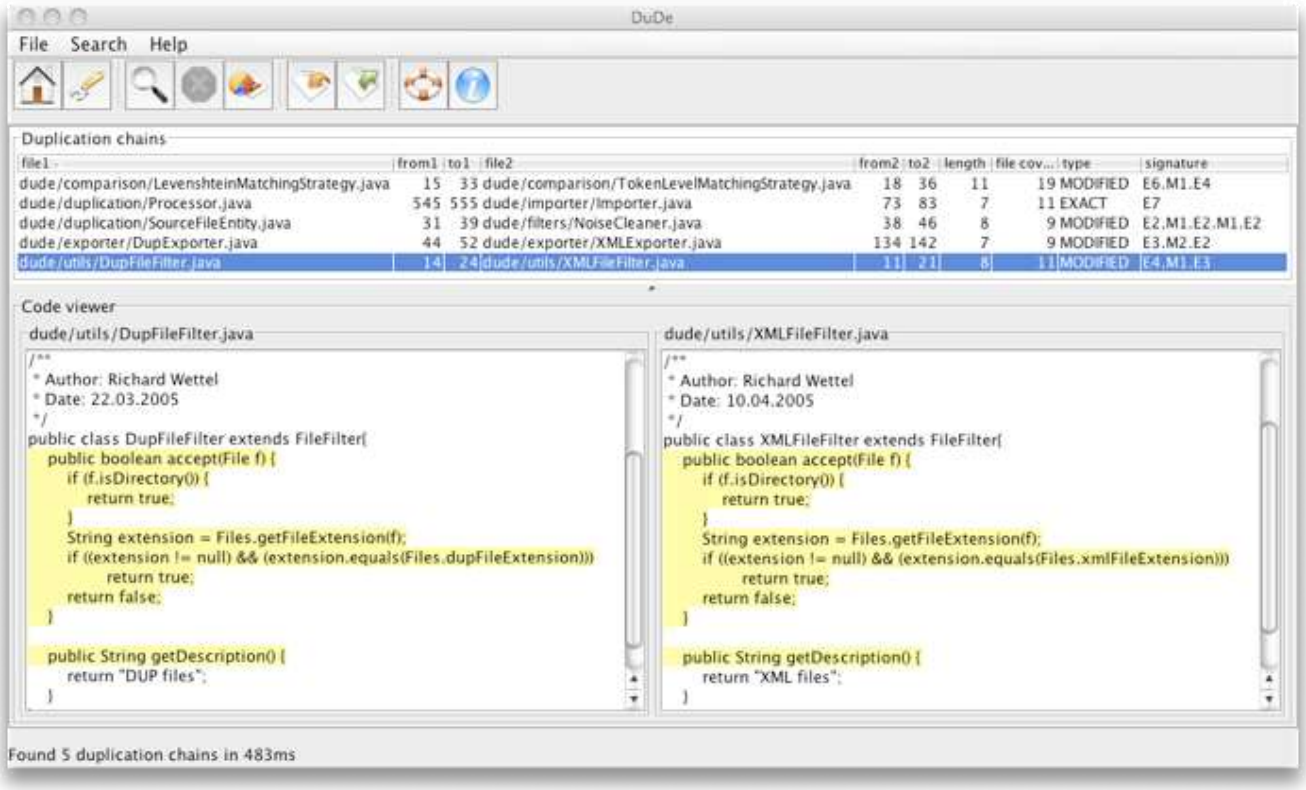


Рисунок 1.6 – Вікно програми SourcererCC

1.4 Обґрунтування вибору методу виявлення дублікатів програмного коду

Процес виявлення дублікатів починається з перетворення в код подання, придатний для оцінки подібності. Наприклад, представляти фрагменти інструментами виявлення дублікатів на основі дерева, що залежать від функції.

У цьому відношенні інформація про домен, яка є корінням в ідентифікаторах відкидається, перериваючи зв’язок між інформацією, яку можна дізнатись на

лексичному рівні та синтаксичному рівні. Однак програмні системи з різних областей додатків і в різні етапи розвитку дають унікальні зразки у вихідному коді, які з’являтимуться для таких проблем, як виявлення дублікатів коду.

Ці зразки не обов'язково відображаються за допомогою підходів, що встановлюють загальний опис функцій, і єдиний спосіб описати ці корисні, приховані функції це використання перспектив коду, які вивчаються, тобто, вивчення самих представлень. Розробка ефективних функцій вихідного коду, аналіз мови ідентифікаторів у вихідному коді, аналіз синтаксичних моделей та інженерних підходів, які можуть адаптуватися до змін сховищ є основними проблемами.

Фрагмент коду – суміжний сегмент вихідного коду, вказаного вихідним файлом та рядками, де коди-дублікати (або дублікати) – це два чи більше фрагментів, подібних відносно типу дублікату.

Робота методів виявлення дублікатів зазвичай починаються з подання коду перед вимірюванням подібності, і ці методи можуть бути класифіковані за поданням вихідного коду.

Текстові методи застосовують незначні перетворення для кодування та вимірювання подібності, порівнюючи послідовності тексту. Отже, текстові методи обмежені в їх здатність розпізнавати два фрагменти навіть як дублікатну пару якщо різниця між ними настільки ж несуттєва, як систематичне перейменування ідентифікаторів.

Методи, що базуються на токенах відображають правило, засноване на тексті, діючи на більш високому рівні абстракції. Ці методи лексично аналізують код для створення потоку лексем та порівняння послідовностей для виявлення дублікатів. Узгодження послідовностей токенів, як правило, покращує розпізнавання, але абстракція лексеми має тенденцію визначати більше помилкових позитивних результатів.

Методи на основі дерева вимірюють схожість підрядів у синтаксичних поданнях. Позначення підмножини вузлів яка відповідно становить обробку абстракції для фрагментів.

Кожен компонент характеризується вектором і представляє кількість зустрічей відповідних вузлів у відповідне піддерево, тому розмірність вектора – це кількість моделей дерев, які вважаються релевантними для наближення для даного дерева. Ця функція являє собою момент розбіжності, для особливостей даних, а не декларувати ряд специфічних особливостей. Причому характерні вектори наближають структурну інформацію, нехтуючи інформацєю про домен. Насправді, як правило, немає спеціального режиму для ідентифікаторів та типів літералу в методах, заснованих на основі дерев.

Методи на основі графіків використовують статичний програмний аналіз для перетворення коду в графік залежності програми (PDG), проміжне подання залежностей від даних та управління.

В таблиці 1.1 представлено порівняльний аналіз основних методів.

Таблиця 1.1 – Порівняльний аналіз методів виявлення дублікатів

Метод	Гранулярність	Тип дублікатів	Швидкість	Повнота	Точність
Текстовий	Довільна	I	Висока	Низька	Висока
Токени	Довільна	I- II	Середня	Висока	Низька
Синтаксичні дерева	Довільна	I- III	Середня	Низька	Висока
Графи	Довільна	I-IV	Низька	Середня	Висока
Метрики	Фіксована	I-IV	Середня	Середня	Середня

Ґрунтуючись на наведених вимогах, представлених у розділі, можна зробити висновок про неможливість використання методів з внутрішнім поданням у вигляді тексту або токенів. А в зв'язку з низькою повнотою виявлення при використанні методів, заснованих на аналізі синтаксичних дерев, такі методи також відпадають. Тому доцільно використовувати гібридний метод.

1.5 Постановка задачі виявлення дублікатів програмного коду

Головна вимога, яка пред'являється до методу, що розробляється – виявлення програмних дублікатів перших трьох типів. Для вирішення даної задачі доцільно використовувати нейронні мережі.

Нейронні мережі виконують завдання пошуку схожих елементів з високою швидкістю і точністю. Таким чином, метод виявлення дублікатів, що розробляється, повинен забезпечувати вирішення наступних завдань:

- виявлення дублікатів I-III типів;
- максимізація повноти і точності виявлення;
- визначення інформації про дублікати у вигляді дублікатових класів.

Незважаючи на той факт, що метод повинен бути інтелектуальним, в якості попередньої обробки необхідно привести вихідний код до одного з відповідних видів внутрішнього подання.

Для вирішення поставленої задачі доцільно використати кілька форм подання (гібридний метод), а саме, AST і послідовність токенів.

Використання AST дозволяє зберегти інформацію про структуру вихідного тексту програми, саме завдяки цьому стає можливим збільшити повноту і точність результатів.

Використання токенів як подання вихідного коду дозволяє, в свою чергу, значно скоротити розмір вхідних даних. Загальна структура внутрішнього подання вихідного коду зображена на рисунку 1.7

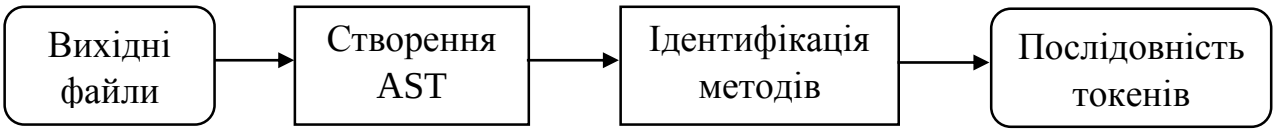


Рисунок 1.7 – Структура внутрішнього подання коду

1.6 Висновок

В даному розділі були наведені основні визначення та класифікації, пов'язані з предметною областю. Проаналізовано сучасний стан предметної області.

Був проведений аналіз існуючих методів виявлення дублікатів та було визначено характеристики методів у задачах виявлення дублікатів програмного коду.

Під час аналізу методів визначено, що існуючі методи виявлення дублікатів мають недостатні характеристики повноти і точності, тому визначено, що створення гібридного методу, який має внутрішнє подання і способи обробки від різних методів підвищить ці характеристики.

2 РОЗРОБКА ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ ВИЯВЛЕННЯ ДУБЛІКАТІВ ПРОГРАМНОГО КОДУ

2.1 Розробка подання вихідного коду за допомогою абстрактного синтаксичного дерева

Подамо синтаксис мови програмування як ієрархічну деревоподібну структуру, для чого використаємо спосіб подання абстрактного синтаксичного дерева (AST). Така структура дозволить здійснити генерацію таблиць символів для компіляторів і подальшої генерації коду. Отже, подамо за допомогою дерева всі конструкції мови. Абстрактне синтаксичне дерево представляє всі синтаксичні елементи мови програмування, подібно до синтаксичних дерев, які лінгвісти використовують для людських мов. Його структура фокусується на правилах, а не на елементах, таких як фігурні дужки або крапки з комою, які завершують твердження в деяких мовах.

Дерево є ієрархічним, а елементи програмних операторів розбиті на частини. Наприклад, дерево умовного оператора має правила для змінних, що залежать від необхідного оператора[4].

Абстрактне синтаксичне дерево широко використовуються в компіляторах для перевірки коду на точність. Якщо згенероване дерево містить помилки, компілятор повідомляє про них.

AST використано для того, щоб деякі конструкції не могли бути представлені в контекстно-вільної граматики, такій як неявне типування. Такі структури дуже специфічні для мов програмування.

На рисунках 2.1 і 2.2 зображено програмний код та його подання у вигляді абстрактного снтаксичного дерева.

```
static int
parser_compare(PyST_Object *left, PyST_Object *right)
{
    if (left == right)
        return (0);

    if ((left == 0) || (right == 0))
        return (-1);

    return (parser_compare_nodes(left->st_node, right->st_node));
}
```

Рисунок 2.1 – Програмний код представлення

Рисунок 2.2 – Загальний вигляд представлення

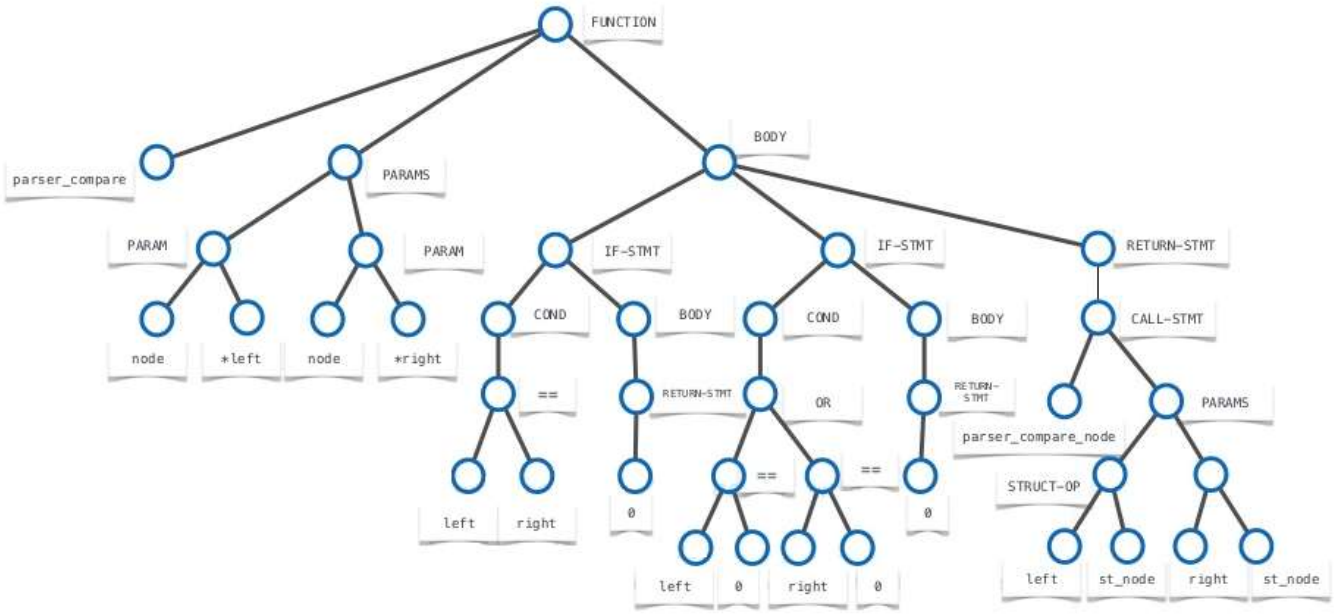


Рисунок 2.3 – Структура подання вихідного коду у вигляді AST

Структура AST представляє один з типів проміжного коду, який представляє ієрархічну синтаксичну структуру програми. Основною метою використання AST є визначення методів, заснованих на навчанні, для кодування довільно довгих послідовностей лексичних елементів.

Починаючи з нетермінальних вузлів AST включають послідовності лексичних елементів, припустимо, кожен вузол AST має спеціальний атрибут, який зберігає векторне подання, що характеризує вузол з розширенням послідовності лексичних елементів, вузла.

Ступінь вузла AST дорівнює або нулю, одиниці, двом або більше двох. За визначенням, AST-вузли зі ступенем нуль або два задовольняють властивість вузлів у повному двійковому дереві, але підрядні корені у вузлах ступеня одиниці або більше двох повинні бути перетворені для того, щоб перетворити піддерево в повне бінарне дерево[5].

Перший Крок трансформації – сканування AST та видалення метаданих (наприклад, вузли Javadoc в AST для фрагментів Java) а також вузли для порожніх

оголошень класу, ініціалізатори порожнього масиву, порожні блоки, порожні класи, порожні одиниці компіляції та порожні оголошення. Відбувається сканування порожніх вузлів в AST, а також сканування послідовності однакових типів з тим самим батьківським класом. Система буде кодувати парні комбінації вузлів AST; це дозволить уникнути кодування пар одного типу, відвідуючи нетермінальні вузли, оглядаючи їх підвузли та згортаючи сусідні, однакові типи на один екземпляр.

Наприклад, true true, true true true елементи перетворяться на true. Скорочення цих послідовностей допомагає контролювати глибину дерево за зменшує ризик втратити роздільну здатність.

Далі, щоб отримати синтаксичне дерево, рядки виокремлюються у випадку вузлів зі ступенем більше двох вони повинні бути реорганізованими, щоб вузли-нащадки були належним чином влаштовані. Визначено граматичний підхід для кожного нетермінального типу, для систематичної реорганізації вузлів-нащадків.

Наприклад, оператор If в представленні може мати два або три вузли-нащадки. Для цього нетермінального типу визначена а нова граматика, яка виробляє підтипи основані на синтаксисі вузлів Вираз і Оператор (2.1).

Оскільки кожен оператор мови програмування має або одну, або дві конструкції необхідно збільшити граматику мови, ввівши нові штучні нетермінальні типи Гілки (2.2):

$$\langle \text{IF} \rangle ::= \langle \text{Вираз} \rangle \langle \text{Гілки} \rangle, \tag{2.1}$$

$$\langle \text{Гілки} \rangle ::= \langle \text{Вираз} \rangle \ [\langle \text{Вираз} \rangle]. \tag{2.2}$$

Для нетермінальних типів з довільним максимальним ступенем (наприклад, вузли Блоки) підвузли представлені послідовністю висловлювань, і продукція буде мати вигляд 2.3.

⟨Блок⟩ ::= {⟨Вираз⟩} (2.3)

Це проектне рішення випливає обумовлене тим, що батьківський вузол, як правило, є більш використовуваним та описаним під час програмування, ніж дочірній вузол. Зокрема, цим підходом забезпечено наступне:

-Певні рівні зернистості захищені і ніколи перезаписуються іншими вузлами.

- При злитті двох вузлів виразніші типи кращі порівняно з більш загальними типами, такими як Вираз та Блок.

-Штучні нетермінальні вузли, створені в попередньому крокі для обробки ніколи не заміняють нетермінальні типи в початковій граматиці.

2.2 Обґрунтування вибору виду нейронної мережі для методу виявлення дублікатів програмного коду

На поточний момент штучні нейронні мережі (ШНМ) досягли високої продуктивності в таких завданнях пошуку схожих елементів, як, наприклад, пошук однакових зображень, фотографій і тексту. Це досягнення – одна з основних причин, за якою штучні нейронні мережі використовуються у запропонованому підході. Для того, щоб визначити, яка з архітектур ШНМ найбільше підходить для вирішення поставленого завдання, необхідно провести порівняльний аналіз:

- Мережа прямого поширення. Мережа передає інформацію тільки в одному напрямку, від входу до виходу. Мережі складаються з повнозв'язних шарів (кожен нейрон з одного шару пов'язаний з кожним нейроном наступного шару), проте, сам шар між собою ніяк не пов'язаний. Кожен шар складається з вхідних, прихованих або вихідних осередків. Області застосування таких мереж досить вузькі, їх можна застосовувати, наприклад, для простих завдань класифікації або передбачень.

- Нейронна мережа Хопфілда. Вже згадана мережа є ШНМ із симетричною матрицею зв'язків. Кожен вузол, в такій мережі, є вхідним до початку процесу навчання, прихованим – під час і вихідні після навчання.

Навчання мережі виконується за допомогою встановлення бажаного значення нейрона, після чого можуть бути розраховані ваги. Як тільки ваги задані, навчена мережа стає здатною «Розпізнавати» вхідні сигнали - тобто, визначати, до якого з запам'ятованих зразків вони відносяться.

- Згорткова нейронна мережа. Згорткова нейронна мережа відрізняється від інших. Основним її завданням є обробка зображень. Досить типовим завданням для неї є класифікація або виявлення об'єктів на зображеннях.

Як правило, такі мережі починають свою роботу з, так званого, вхідного "сканера", який не намагається аналізувати всі вхідні дані разом, а аналізує їх невеликими фрагментами, відповідними його розмірами.

Рекурентна нейронна мережа. В даному виді ШНМ зв'язки між нейронами утворюють своєрідний спрямований граф. Завдяки такій будові з'являється можливість обробки серії подій у часі або послідовні ланцюжка. Однак вузьким місцем таких мереж є проблема зникаючого градієнта, тобто інформація з часом втрачається [6].

Ґрунтуючись на проведеному аналізі архітектур штучних нейронних мереж, можна зробити висновок, що мережі прямого поширення або згорткові нейронні мережі не підходять для завдання аналізу вихідного коду програм.

2.3 Обґрунтування використання рекурентної нейронної мережі

Для виконання операцій виявлення однакових фрагментів у запропонованому підході вирішено використовувати рекурентну нейронну мережу (RtNN) (рис. 2.4) –

мережа з глибинним навчанням, яка добре підходить моделювання послідовностей термінів у вихідному коді з словарем $\mathcal{V}$ де $|\mathcal{V}| = \mathrm{m}$ термінів.

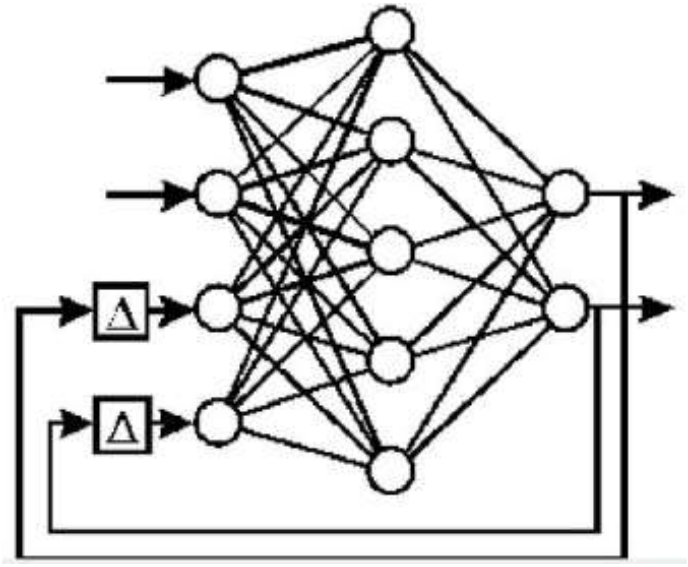


Рисунок 2.4 – Структура рекурентної нейронної мережі

Нехай n, вказаний користувачем параметр – кількість прихованих шарів. RtNN містить вхідний шар $x \in R^{m+n}$ прихований шар $z \in R^{n}$ і вихідний шар $z \in R^{m}$ (якщо припускати евристику наприклад, вихідні шари на основі класів [7]).

Регулювання n регулює ємність моделі. Глибина RtNN призводить до рекурентного виконання, прихований стан копіюється назад у вхідний шар, тому вхідний шар з’єднується з поточним терміном t(i) і минулим станом z(i-1):

$$x(i) = [t(i); z(i-1)] \tag{2.4}$$

Цей вхідний вектор множиться на матрицю $[\alpha, \beta] \in R^{n*(m+n)}$ і переходить до нелінійної векторної функції.

Цей вектор стану помножений на іншу матрицю $y \in R^{m*n}$ і нормалізований для обчислень термінів знову [8].

$$y(i) = p\left(t\middle|x(i)\right) = \max(z(i)) \quad (2.5)$$

В даному фрагменті, інформація надходитиме з кінцевих вузлів через нетермінальні вузли до коріня ієрархічної структури. Цей перевернутий потік інформації схожий на процедури обчислення характерних векторів у традиційних структурно-орієнтованих техніках або обчислення показників у методах на основі метрик.

2.3 Розробка алгоритму гібридного методу виявлення дублікатів програмного коду

Запропонований метод складається з наступних основних етапів:

1. Попередня обробка
2. Перетворення
3. Робота ШНМ
   - Навчання ШНМ
   - Виявлення дублікатів
4. Постобработка

В рамках даного методу, на першому етапі, вихідний код програми представляється у вигляді синтаксичного дерева. Далі з нього виокремлюються піддерева, пов'язані з функціями або методами класів, які потім перетворюються в послідовності токенів. Перед наступним перетворенням здійснюється фільтрація і нормалізація піддерев

На наступному етапі проводиться створення векторного подання токенів. З його допомогою виконується подальше навчання нейронної мережі і аналіз вихідного коду на наявність програмних дублікатів.

Далі слідує процес навчання нейронної мережі. У тому випадку, коли мережа вже навчена, на цьому етапі буде проводитись пошук дублікатів за допомогою цієї мережі.

На останньому етапі з усіх виявлених дублікатів формуються класи дублікати, які, надалі будуть представлені користувачеві. Загальна структура інформаційної технології виявлення дублікатів програмного коду представлена на рисунку 2.5.

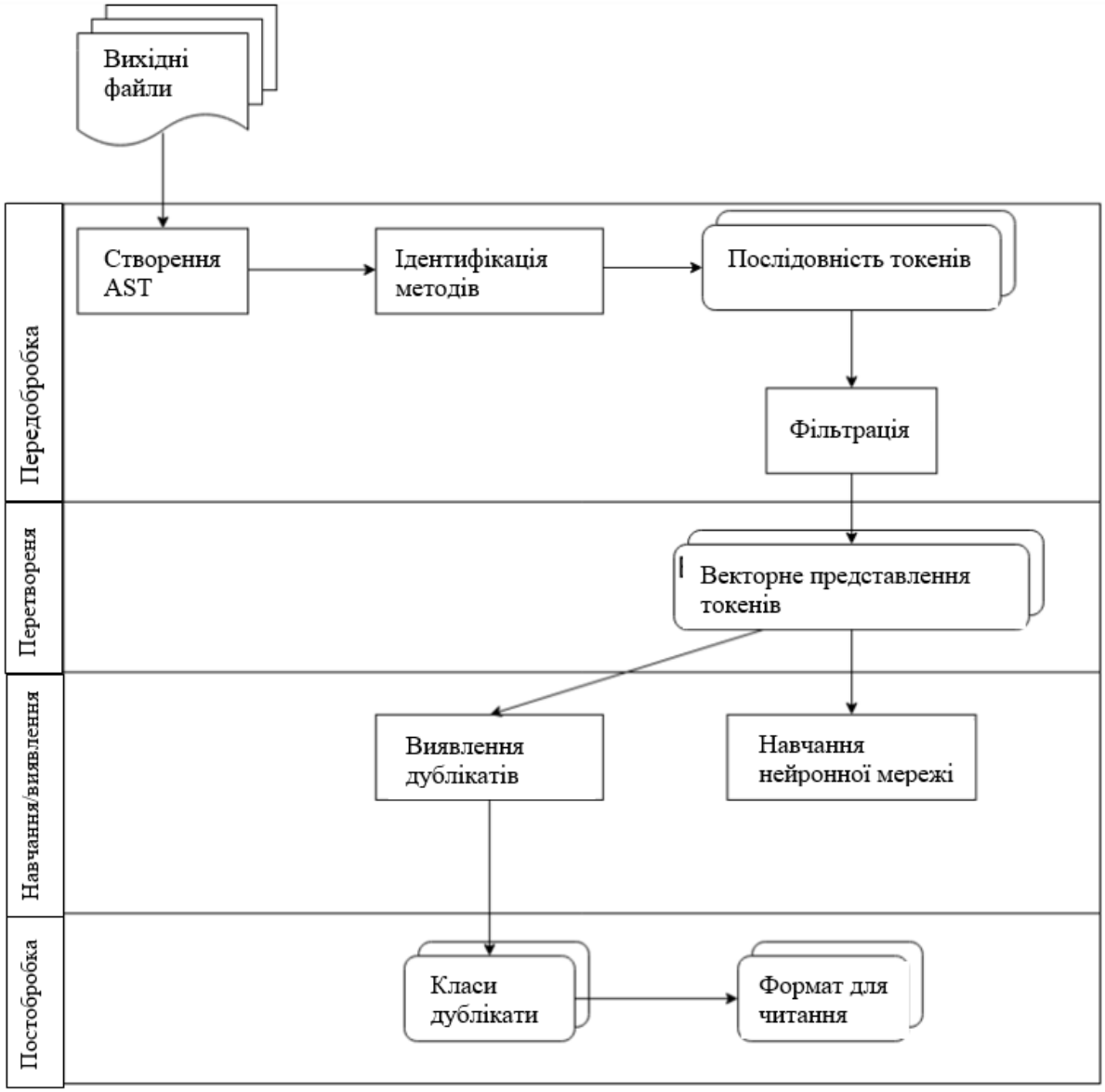


Рисунок 2.5 - Загальна структура інформаційної технології виявлення дублікатів програмного коду

2.5.1 Передобробка

На даному етапі, за допомогою аналізатора, заснованого на частини середовища розробки (IDE Intellij IDEA community), проводиться подання вихідного коду програми у вигляді AST. Після чого, з отриманого дерева виділяються тільки ті піддерева, які відносяться до методів або функцій. Такий пошук здійснюється за допомогою обходу в глибину всіх вершин AST. У разі виявлення вершини шуканого типу, подальший спуск в дану гілку не провадиться.

В наступному етапі отримані піддерева перетворюються в послідовності токенів шляхом того ж обходу в глибину всіх їх вершин. Після чого проводиться фільтрація цих послідовностей. Вироблена фільтрація дозволяє виключити вплив незначних відмінностей фрагментів вихідного коду один від одного. В рамках даного підходу фільтрації піддаються токени наступних типів:

- коментарі різного типу;
- елементи форматування;
- списки параметрів;

Зазвичай, для виявлення дублікатів складніше I типу, необхідно виробляти анонімізація токенів. Однак, в цьому методі, нейронна мережа буде порівнювати типи токенів. З цього можна зробити висновок про непотрібність анонімізації. Таким чином, процес фільтрації – останній процес на етапі попередньої обробки.

2.5.2 Перетворення

На даному етапі проводиться перетворення отриманих послідовностей токенів в їх векторне подання. Перетворення проводиться за допомогою моделі Word2Vec. Дана модель включає в себе набір алгоритмів розрахунку векторних прудставлень слів, припускаючи семантичну близькість слів які використовуються в схожих контекстах.

Вже згадана модель в своїй роботі використовує нейронну мережу прямого поширення. У Word2Vec існують два алгоритми навчання: CBOW (Continuous Bag of Words) і Skip-gram, зображена на рисунку 2.6, CBOW і Skip-gram – це нейромережеві архітектури, які описують, як саме нейромережа навчається на даних і запам'ятовує подання слів. Принципи у обох архітектур різні.

Принцип роботи CBOW - передбачення слова при даному контексті, а Skip-gram навпаки - передбачається контекст при даному слові [9].

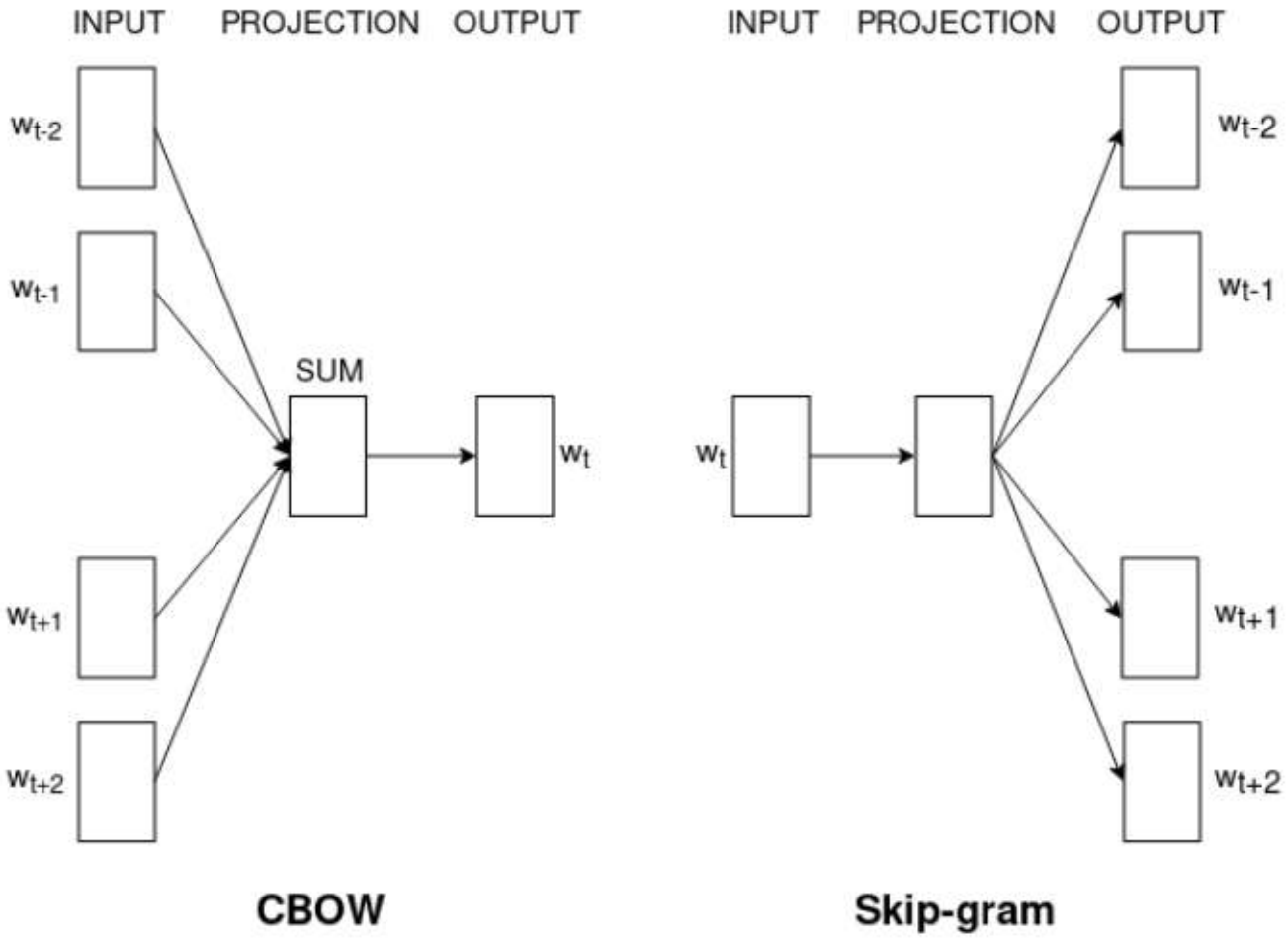


Рисунок 2.6 – Підходи до навчання Word2Vec

В даному підході використовується алгоритм Skip-gram. Як вже говорилося раніше, мета навчання Skip-gram моделі - знайти такі уявлення слів, які будуть корисні для передбачення контексту в реченні або документі. Формально, при

заданій послідовності навчальних слів w_1, w_2, w_3, ..., w_T метою навченою моделі є максимізація середньої логарифмічною ймовірності.

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c<j<c,j=0}\log p\left(w_{t+j}\middle|w_t\right) \tag{2.6}$$

де c - розмір навчального контексту (який може бути функцією центрального слова wt).

Таким чином, після виконання даного етапу, були отримані векторні подання для кожного токена.

2.5.3 Навчання і використання нейронної мережі

В рамках запропонованого методу виявлення програмних дублікатів здійснюється за допомогою нейронних мереж. для коректного використання мереж їх необхідно навчити. У даній секції буде розглядатися етап роботи нейронної мережі, який ділиться на два інших:

- Навчання НС;
- Робота навченої НС.

Одним з найважливіших елементів в навчанні нейронних мереж є набір навчальних даних. У відкритому доступі, для даної завдання, які підходять наборів даних високої якості дуже мало.

Використання результатів інших утиліт, як набір даних, не дозволить об'єктивно оцінити пропонований підхід, так як таке рішення призведе до копіювання функціональності таких інструментів.

В якості основної навчальної і тестової вибірки був обраний набір даних BigCloneBench. Дана вибірка представляє колекцію, що складається з 8 млн перевірених програмних дублікатів і 25 тис. Java систем з відкритим вихідним

кодом. BigCloneBench містить як внутріпроектние, так і міжпроектної програмні дублікати чотирьох типів.

Головною перевагою даної вибірки є її незалежне створення від інструментів пошуку дублікатів. Створення вибірки таким чином дозволяє уникнути копіювання функціональності існуючих інструментів.

Як було згадано раніше, в пропонованому методі будуть використовуватися рекурентні нейронні мережі. Однак, вони мають недолік, який полягає у втраті інформації з часом. Такий недолік може бути легко усунено при використанні мереж з довгою короткостроковою пам'яттю (LSTM). У LSTM-мережах внутрішні нейрони «обладнані» складною системою так званих воріт (gates), а також концепцією клітинного стану (cell state), яка і являє собою якийсь вид довгострокової пам'яті. Ворота ж визначають, яка інформація потрапить в клітинне стан, яка зітреться з нього, і яка вплине на результат, який видасть РНС на даному етапі.

2.5.4 Сіамська архітектура нейронної мережі

Як архітектура основної нейронної мережі була обрана, так звана, сіамська архітектура. Така архітектура застосовується для оцінки семантичної схожості пропозицій.

Сіамська архітектура нейронної мережі (SNN) складається з двох нейронних мереж, які поділяють однакові ваги і з'єднуються на одному або декількох шарах. SNN отримують пари векторів як вхід на етапах навчання та тестування для розвитку знань на рівні об'єкт-об'єкт.

Структура архітектури зображена на рисунку 2.7.

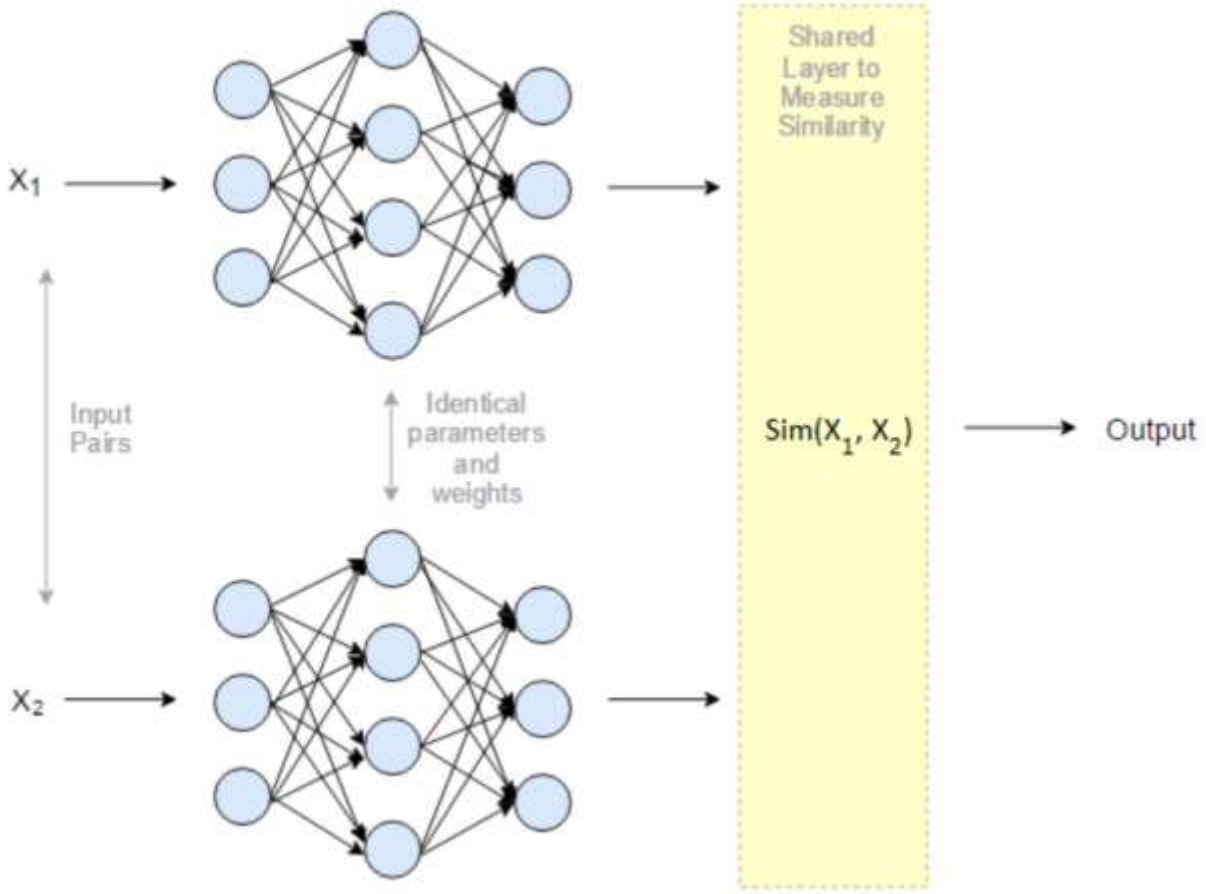


Рисунок 2.7 – Сіамська архітектура нейронної мережі

Під час навчання, ці пари позначаються як "справжні" – якщо приклади поділяють той самий клас або "самозванці" – якщо приклади є різними класами. Це дозволяє мережі для розробки багатовимірного простору на основі особливостей справ, де "справжні" пари підштовхуються ближче один до одного, а пари "самозванці" відокремлюються далі одне від одного.

Вихід однакових нейронних мереж – це вектори функцій для кожного члена вхідної пари. Відстань між цими векторами вимірюється за рівнем схожості та встановлення належать вони до одного класу на основі порогового значення.

SNN використовують "контрастні втрати", що введено в [10]. Контрастна втрата обчислюється шляхом підсумовування результатів окремих формул втрат за справжніми і парами самозванців. Справжні пари діляться коефіцієнтом L_G коли

вони занадто далеко, поки негативні пари діляться на коефіцієнт L_I, якщо їх відстань потрапляє в межах заданого значення. Після цього ваги підмереж оновлюються за допомогою зворотного розповсюдження втрати відносно ваг. Це означає, що справжні пари підштовхуються ближче один до одного під час тренувань, забезпечуючи при цьому пари самозванців витримувати принаймні встановлену відстань.

Прказники контрастної втрати описані в рівняннях (2.6), (2.7) та (2.8). Y_A і Y_P - двійкові значення, які дорівнюють 0 для справжніх пар і 1 для пари самозванців:

$$L_G = (1 - Y_A)Y_p, \tag{2.6}$$

$$L_1 = Y_A\big(\max(M - Y_p, 0)\big)^2, \tag{2.7}$$

$$L = L_G + L_1. \tag{2.8}$$

де Y_A - фактична мітка,

Y_P - передбачувана мітка,

M - маржа.

Архітектура складається з двох LSTM мереж зі зв'язаними вагами, кожна з яких обробляє одна з пропозицій взаданої парі.

Дана модель використовує LSTM для читання векторних представлень слів кожної вхідної пропозиції і використовує свій остаточний прихований стан як векторне подання пропозицій. Згодом, схожість між цими поданнями використовується як предиктор семантичного подібності [16].

Навчальна вибірка для сіамської мережі складається з триплетів x_1, x_2, y, де x_1 і x_2 порівнювані послідовності, а $y \in \{0, 1\}$ визначає схожість x_1 та x_2 ($y = 1$) або їх відмінність ($y = 0$). Метою навчання є зменшення дистанції між схожими парами і її збільшення між різними.

Навчання мережі проводиться за допомогою використання контрастного коефіцієнта втрат і квадратичної регуляції l_2, що призводить до наступної цільової функції навчання:

$$L(w) = \frac{\lambda}{2}\|w\|_2^2 + \frac{1}{2N}\sum_{(i,j)\in D} y_{ij} d_{ij}^2 + \left(1 - y_{ij}\right)\max\left(1 - d_{ij}^2, 0\right) \qquad (2.9)$$

де w - ваги нейронної мережі,

D - набір навчальних пар,

$d^2{}_{ij}$ - квадратична відстань

l_2 відстань між i і j послідовностями (розраховане між двома останніми шарами сіамської ШНМ),

$y_{ij} \in 0, 1$ – як і було розглянуто раніше, значення відповідає за схожість або відмінність послідовностей.

2.5.5 Модель подання Sequence-to-Sequence

У запропонованому підході програмні дублікати будуть визначатися на рівні методів. Так як методи можуть бути різних розмірів, то і їх векторні подання можуть бути також різних розмірів. Існує два способи приведення уявлень методів до єдиного розміру:

- Вибір максимального розміру і доповнення нулями до нього
- Використання моделі sequence-to-sequence (seq2seq)

В даному методі була використана модель seq2seq.

Дана модель має більший успіх в різних завданнях, наприклад: розпізнавання мови, машинний переклад, узагальнення текстів [17].

Суть такої моделі полягає в використанні LSTM для зчитування вхідної послідовності невеликими фрагментами до отримання великого векторного подання

з фіксованою розмірністю. Потім використовується інша LSTM для вилучення вихідної послідовності з цього вектора [18].

Як необхідне подання з фіксованим розміром використовується значення прихованого шару на останньому етапі розгортання кодувальника (thought vector). Модель Sequence-to-Sequence зображено на рисунку 2.8.

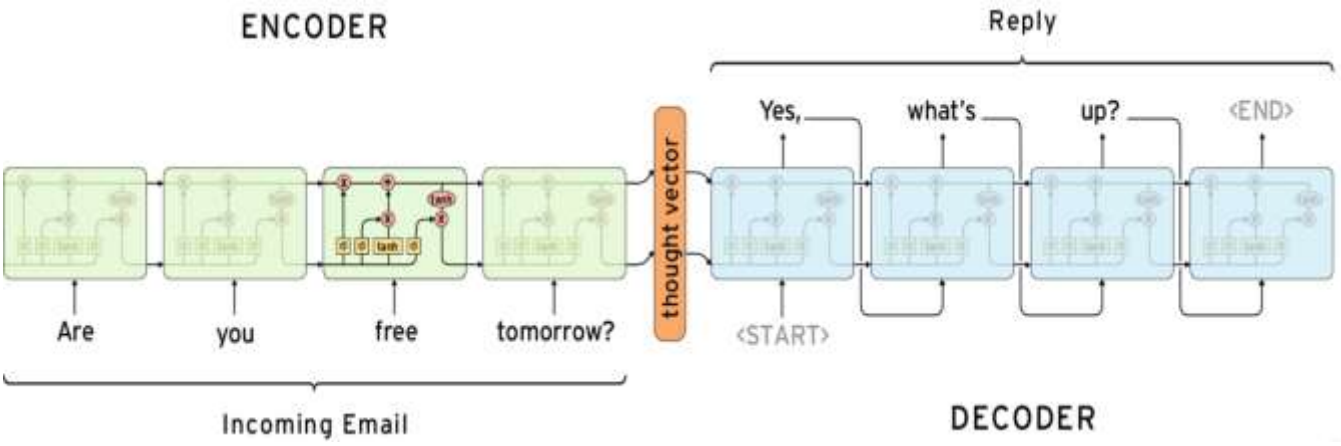


Рисунок 2.8 – Модель Sequence-to-Sequence

2.5.6 Постобробка

Фінальний етап запропонованого методу - постобработка. Суть даного етапу полягає у приведенні результатів в компактному форматі для сприйняття. Без цього етапу результат роботи мережі виглядає як набір векторних представлень з бінарною відповіддю про схожість двох методів.

Таким чином, даний етап складається з двох частин:

- Створення дублікатових класів (Компактність)
- Визначення шляху методу і його розташування у файлі (номери рядків початку методу і його кінця)

2.6 Висновок

В даному розділі була сформована задача для методу виявлення дублікатів програмного коду, що розробляється.

Був представлений гібридний метод виявлення дублікатів. Розглянуто загальну структуру процесу виявлення, а також основні етапи методу.

Було проаналізовано властивості абстрактних синтаксичних дерев у задачах подання вихідного коду програм.

Проведено аналіз нейронних мереж в задачах виявлення дублікатів. Визначено, що рекурентні нейронні мережі найкраще виконують такі задачі. Визначено архітектуру нейронної мережі і технології для створення навчальних і тестових даних.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ ВИЯВЛЕННЯ ДУБЛІКАТІВ ПРОГРАМНОГО КОДУ

3.1 Обґрунтування вибору мови та середовища програмування інформаційної технології виявлення дублікатів програмного коду

C# – об'єктно-орієнтована мова програмування. Розроблена в 1998-2001 роках групою інженерів під керівництвом Андерса Хейлсберга в компанії Microsoft як мова розробки додатків для платформи Microsoft .NET Framework і згодом була стандартизована як ECMA-334 і ISO / IEC 23270.

C# відноситься до сім'ї мов з C-подібним синтаксисом, з них його синтаксис найбільш близький до C++ і Java. Мова має статичну типізацію, підтримує поліморфізм, перевантаження операторів (в тому числі операторів явного і неявного приведення типу), делегати, атрибути, події, властивості, узагальнені типи і методи, ітератори, анонімні функції з підтримкою замикань, LINQ, виключення, коментарі в форматі XML.

Перейнявши багато від своїх попередників – мов C++, Паскаль, Модула, Smalltalk і, особливо, Java, спираючись на практику їх використання, C# виключає деякі моделі, що зарекомендували себе як проблематичні при розробці програмних систем, наприклад, C# на відміну від C++ не підтримує множинне успадкування класів (між тим допускається множинне спадкування інтерфейсів).

C# розроблялась як мова програмування прикладного рівня для CLR і, яка залежить, перш за все, від можливостей самої CLR. Це стосується системи типів C#. Присутність або відсутність тих чи інших виразних особливостей мови диктується тим, чи може конкретна мовна особливість бути трансльований в відповідні конструкції CLR. Так, з розвитком CLR від версії 1.1 до 2.0 значно збагатився і сам C#.

CLR надає C#, як і всім іншим .NET-орієнтованим мовам, багато можливостей, яких позбавлені «класичні» мови програмування. Наприклад, Прибирання сміття не реалізоване в самому C#, а проводиться CLR для програм, написаних на C# точно так, як це робиться для програм на VB.NET, J # і ін.

Microsoft Visual Studio – збірка продуктів компанії Microsoft, що включають інтегроване середовище розробки програмного забезпечення і ряд інших інструментальних засобів. Дані продукти дозволяють розробляти як консольні додатки, так і додатки з графічним інтерфейсом, в тому числі з підтримкою технології Windows Forms, а також веб-сайти, веб-додатки, веб-служби для всіх платформ, підтримуючих Windows, Windows Mobile, Windows CE, .NET Framework, Xbox, Windows Phone Платформа .NET Compact Framework і Silverlight.

Visual Studio включає в себе редактор вихідного коду з підтримкою технології IntelliSense і можливістю найпростішого рефакторінга коду. Вбудований відладчик може працювати як відладчик рівня вихідного коду, так і відладчик машинного рівня. Інші вбудовані інструменти включають в себе редактор форм для спрощення створення графічного інтерфейсу додатку, веб-редактор, дизайнер класів і дизайнер схеми бази даних.

Visual Studio дозволяє створювати і підключати сторонні додатки (плагіни) для розширення функціональності практично на кожному рівні, включаючи додавання підтримки систем контролю версій вихідного коду (як, наприклад, Subversion і Visual SourceSafe), додавання нових наборів інструментів (наприклад, для редагування і візуального проектування коду в предметно-орієнтованих мовах програмування) або інструментів для інших аспектів процесу розробки програмного забезпечення (наприклад, клієнт Team Explorer для роботи з сервером Team Foundation).

Visual Studio включає один або декілька з наступних компонентів: Visual Basic .NET, Visual C++, Visual C#, Visual J#, Visual F# (входить до складу Visual Studio 2010), Visual Studio Debugger

Багато варіантів постачання також включають Microsoft SQL Server або MSDE Visual Source Safe — файл-серверна система управління версіями.

3.2 Програмна реалізація інформаційної технології виявлення дублікатів програмного коду

Загальна структура програмної реалізації зображена на рисунку 3.1. Інфрормаційна система виявлення дублікатів складається і двох модулів:

- Модуль побудови AST – виконує відповідні операції для передобробки перетворення;
- Модуль пошуку дублікатів за допомогою - виконує пошук дублікатів за допомгогю нейронної мережі.

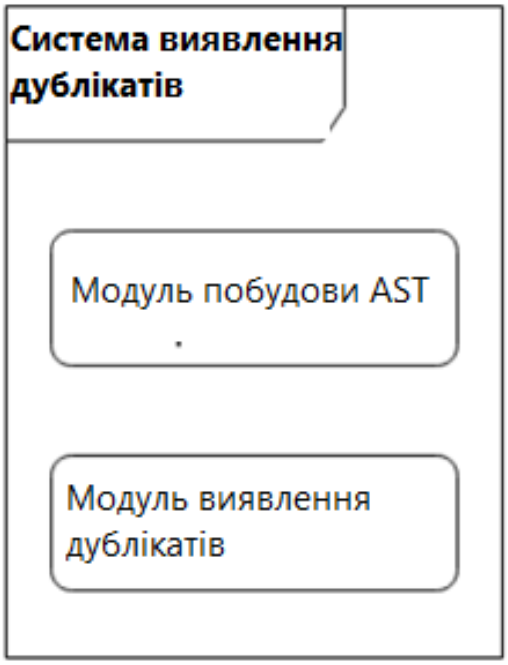


Рисунок 3.1 – Структура програмної реалізації інформаційної технології

3.2.1 Реалізація модуля побудови AST

Модуль побудови AST складається з декількох основних проектів:

- TreeProcessor - реалізує побудову AST із заданого вихідного коду;

- PreProcessor - забезпечує перетворення AST в послідовність токенів. Також виконує векторне подання токенів;

- DatabaseHelper - реалізує спілкування з базою даних PostgreSQL ввипадку використання BigCloneBench.

Структура модуля побудови AST зображена на рисунку 3.2.

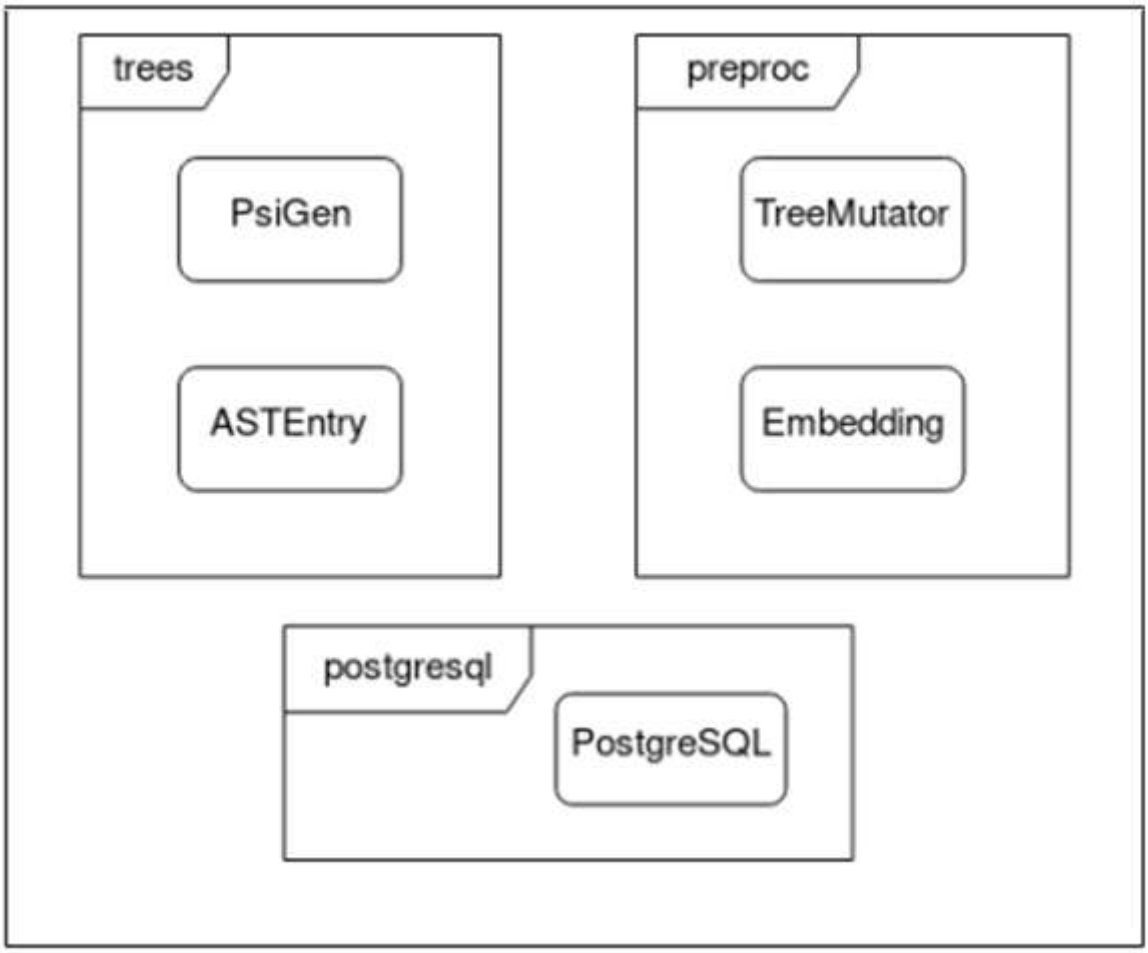


Рисунок 3.2 – Структура модуля побудови AST

Проект TreeProcessor містить класи, що забезпечують перетворення вихідного коду в абстрактне дерево. В даний проект входять два класи:

AstGen – клас, який відповідає за аналіз вихідного коду програми і створення з нього PSI. Крім створення AST, що розглядається клас перетворює AST в послідовність токенів.

ASTEntry – клас, який відповідає за аналіз вихідного коду програми і створення з нього AST. Крім створення AST, що розглядається клас перетворює AST в послідовність токенів.

Клас ASTEntry представляє реалізацію токена, що входить в AST. Цей клас включає в себе наступні і методи класу:

- nodeName - елемент класу, який містить в собі ім'я вузла;
- sourceStart - елемент класу, який визначає рядок, де починається даний токен;
- sourceEnd - елемент класу, який визначає рядок, де закінчується даний токен;
- parent - елемент класу, який вказує на батьківський токена;
- children - список всіх дочірніх токенів для розглянутого
- filePath - містить в собі шлях до файлу з якого отримано даний токен;
- removeSpaces - метод, що виконує видалення непотрібних токенів зі списку дочірніх;
- getAllTokensList - метод, який обходить список всіх дочірніх токенів і повертає тільки їх імена у вигляді списку;
- mutate - метод, що виконує мутації різного типу;

У пороект Preprocessor входять класи, які відповідають за створення векторних представлень токенів і мутації цих токенів:

TreeMutator - даний клас організовує роботу з вихідним кодом. А саме - викликає створення AST, виконує перетворення AST в послідовність токенів, виконує її фільтрацію і, при необхідності - мутацію;

Embedding - реалізує створення векторних уявлень токенов і записує отримані результати в файли. Таким чином, даний клас організовує створення вибірки даних для подальшого навчання мереж.

Клас Embedding містить у собі реалізацію моделі word2vec з допомогою якої проводиться створення векторних представлень.

В даний клас входять такі методи:

Train - метод, який здійснює навчання моделі word2vec на заздалегідь обраному проекті. В даному методі, також, проводиться серіалізація навченої моделі з метою її повторного використання.

CreateEmbedding - метод, який відповідає за створення векторного подання заданих токенів і збереження їх в файл.

3.2.1 Реалізація модуля виявлення дублікатів

Структура модуля виявлення дублікатів зображена на рисунку. 3.3. Даний модуль складається з декількох файлів, в яких описані основні класи і методи:

ClonesRecognition - основний проект, в якому організовується робота з інструментом. Викликаються різні методи ініціалізації, навчання, обчислення;

Model - проект, в якому описуються моделі нейронних мереж, методи їх навчання і використання;

Helpers - допоміжний проект, в якому описуються різні методи перетворення інформації, збереження, читання;

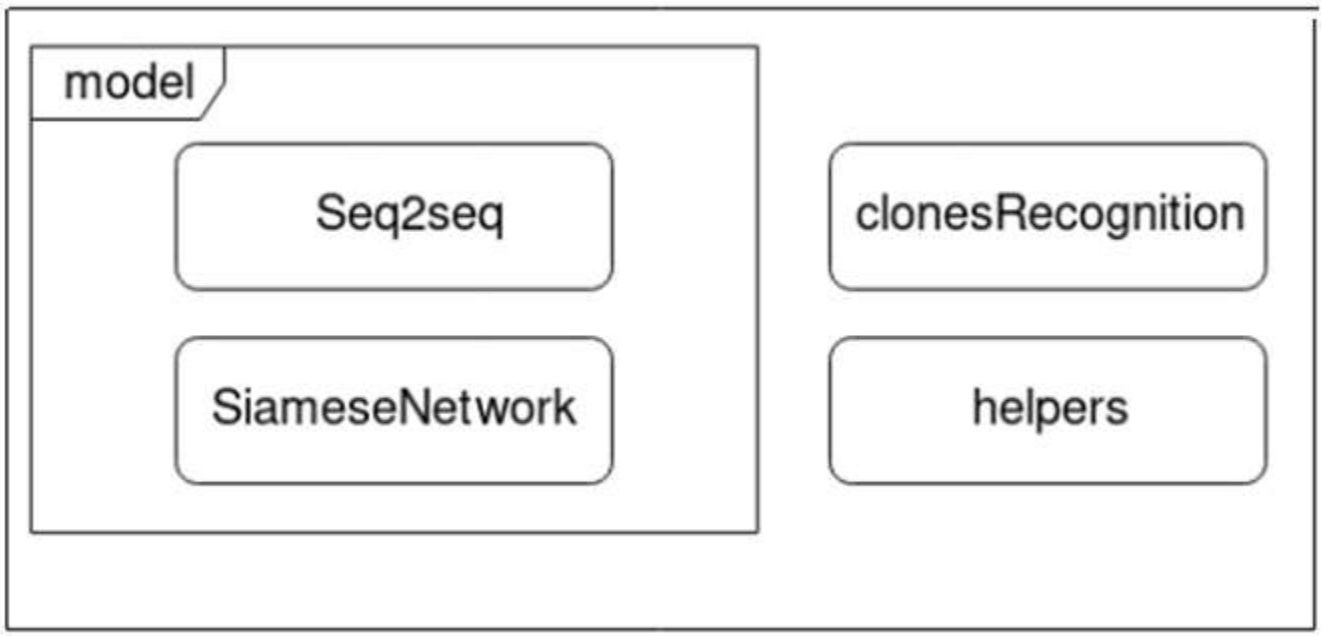


Рисунок 3.3 – Структура модуля виявлення дублікатів

Проект model зберігає класи, що описують нейронні мережі:

- Seq2seq - клас, який реалізує seq2seq алгоритм;
- SiameseNetwork - клас, який реалізує сіамську архітектуру нейронної мережі
- Клас Seq2seq включає наступні елементи:
- Scope - містить в собі назву межі, в якій будуть існувати всі змінні, необхідний для коректної сериализации і десереалізаціі навченої моделі;
- Sess - сесія всередині якої будуть проходити навчання і обчислення;
- encoder_cell - осередок кодувальника, відповідає за спроби привести інформацію до вектору фіксованого розміру;
- decoder_cell - осередок декодувальника;
- vocab_size - розмір словника, довжина векторного подання токена;
- input_embedding_size - розмір вхідної послідовності.

Крім зазначених елементів, в даному класі присутні методи навчання (train) і обчислення (get_encoder_status).

В класі SiameseNetwork присутні елементи схожі з елементами попереднього класу, а також додаткові елементи:

- input_x1 - змінна першої послідовності для порівняння;
- input_x2 - змінна другої послідовності для порівняння;
- input_y - змінна вказує на схожість чи відмінність перших двох (0 - схожі, 1 - різні);
- sequence_length - змінна, яка вказує довжину послідовності, результат роботи seq2seq;

layers - кількість шарів для навчання.

У проекті helpers містяться допоміжні методи. До них входять:

- Batch - створює серії значень із заданої послідовності;
- siam_batches - представляє задані параметри для навчання сіамської мережі у вигляді трійки значень $\{x1, x2, y\}$;
- random_sequences
- save_model - виконує збереження моделі в зазначений файл;
- load_model - виконує завантаження моделі з зазначеного файлу

Проект clonesRecognition є початковою точкою розглянутої утиліти. У ньому містяться такі методи:

- Main - вхідна точка програми. У ній виробляється ініціалізація основних значень;
- seq2seq_train - навчання seq2seq моделі;
- siam_train - навчання сіамської неуронної мережі;
- eval - виконує повний розрахунок.

3.3 Тестування та аналіз результатів роботи інформаційної технології

Для тестування розробленої системи використовуються вибірки BigCloneBench та OJClone, два публічних набори даних, які зазвичай використовуються для оцінки підходів виявлення дублікатів коду.

BigCloneBench, орієнтований на великі дані реальних дублікатів для оцінки сучасних інструментів виявлення дублікатів коду. Він був побудований на основі часто реалізованих функцій в 25000 Java систем, загальною кількістю 365 мільйонів рядків коду.

Поточна версія BigCloneBench налічує близько 8 мільйонів справжніх пар дублікатів, що охоплюють 43 функціональності. Через неоднозначність між виявленням дублікатві типу III і дублікатві типу VI, творці BigCloneBench розділили ці два типи дублікатів на чотири категорії на основі їх синтаксичної схожості:

- [0,0, 0.5] - переважно дублікати IV типу;
- [0.5, 0.7]- переважно дублікати III типу;
- [0.7, 1.0] - строго III тип дублікатів.

Вимірюється синтаксична схожість як відношення рядків або лексем, за якими поділяються фрагменти коду після нормалізації типу I і типу 2, ідентифіковані за допомогою інструменту Linux "diff".

Оскільки більшість пар дублікатів коду - це дублікати перших трьох типів BigCloneBench цілком доречно використовувати для оцінка виявлення дублікатів програмного коду.

OJClone є іншим публічним набором даних, що використовується для оцінки виявлення дублікатів коду. Набір даних складається з рішень написаних мовою програмування C.

Для тестування розробленої інформаційної технології з вибірки BigCloneBench та OJClone були виділені екземпляри з різною кількістю методів: 5000, 7000, 10000, 20000 методів.

Метою магістерської квалфікаційної роботи є підвищення точності виявлення дублікатів програмного коду за рахунок розробки інтелектуального методу.

Основними вимогами, до даного методу, є збільшення точності та виявлення дублікатів перших трьох типів.

Дані характеристики досліджуються на прикладі використання розробленої системи виявлення дублікатів. Основні характеристики тестування системи:

K - кількість рядків вихідного коду (в тисячах);

N_m - кількість досліджуваних методів;

N_d - кількість знайдених дублікатів;

R - повнота результату;

P - точність результату;

F_1 - гармонійне середнє значення повноти і точності;

t - повний час роботи (без урахування навчання мереж).

Значення повноти результату, точності і середнього значення F_1 визначаються за формулами (3.1), (3.2), (3.3).

$$R = \frac{TP}{TP + FN}, \tag{3.1}$$

$$P = \frac{TP}{TP + FP}, \tag{3.2}$$

$$F_1 = 2 * \frac{P * R}{P + R}. \tag{3.3}$$

де: TP - істинно-позитивні результати;

TN - істинно-негативні результати;

FP - хибно-позитивні результати;

FN - хибно-негативні результати.

Таблиця 3.1 – Метрики визначення основних характеристик тестування системи

	Дублікат	Не дублікат
Дублікат	TP	TN
Не дублікат	FP	FN

Тестування проводиться на мережах, навчених на двох різних вибірках. У першому випадку, сіамська нейронна мережа була навчена на вибірці з BigCloneBench, у другому – на вибірці OJClone. В якості навчальної вибірки використовувався набір з 40000 пар: 30000 з них - дублікати, інші 10000 - ні.

Для набору даних BigCloneBench під час тестування використано 9000 фрагментів коду. Для набору даних OJClone 7, 500 фрагментів коду.

Результат тестування наведено в таблицях 3.1 і 3.2

Таблиця 3.2 – Результати тестування для набору даних BigCloneBench

N_m	K	N_d	R	P	F_1	t
4112	232	1734	0.96	0.94	0.95	4.23 хв
8343	756	5547	0.89	0.95	0.92	6.47 хв
10324	3164	32489	0.88	0.94	0.90	9.35 хв
19874	2920	25476	0.90	0.91	0.90	14.52 хв

Таблиця 3.3 – Результати тестування для набору даних OJClone

N_m	K	N_d	R	P	F_1	t
3951	195	1028	0.95	0.94	0.94	04.16 хв
7954	721	5267	0.97	0.98	0.97	06.34 хв
12573	2803	16612	0.93	0.97	0.95	12.36 хв
18539	2947	32679	0.96	0.98	0.97	15.18 хв

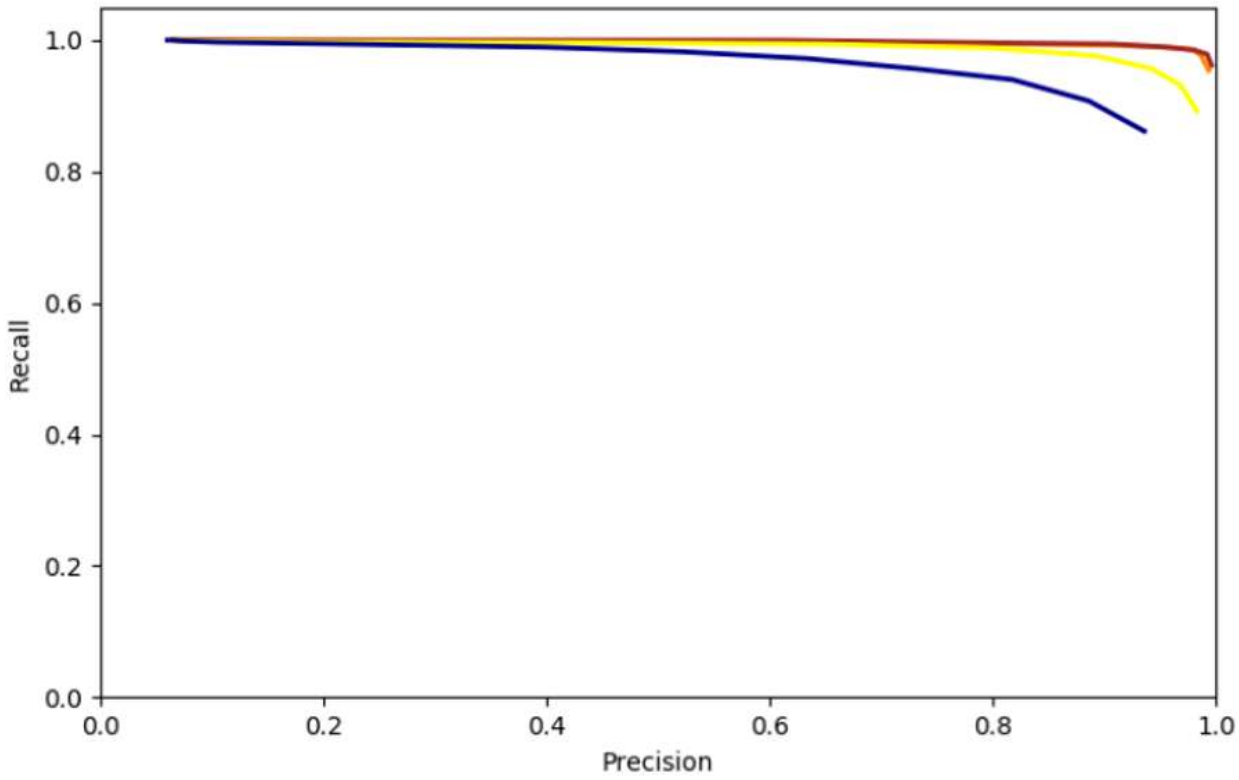


Рисунок 3.4 – Результати тестування для набору даних BigCloneBench

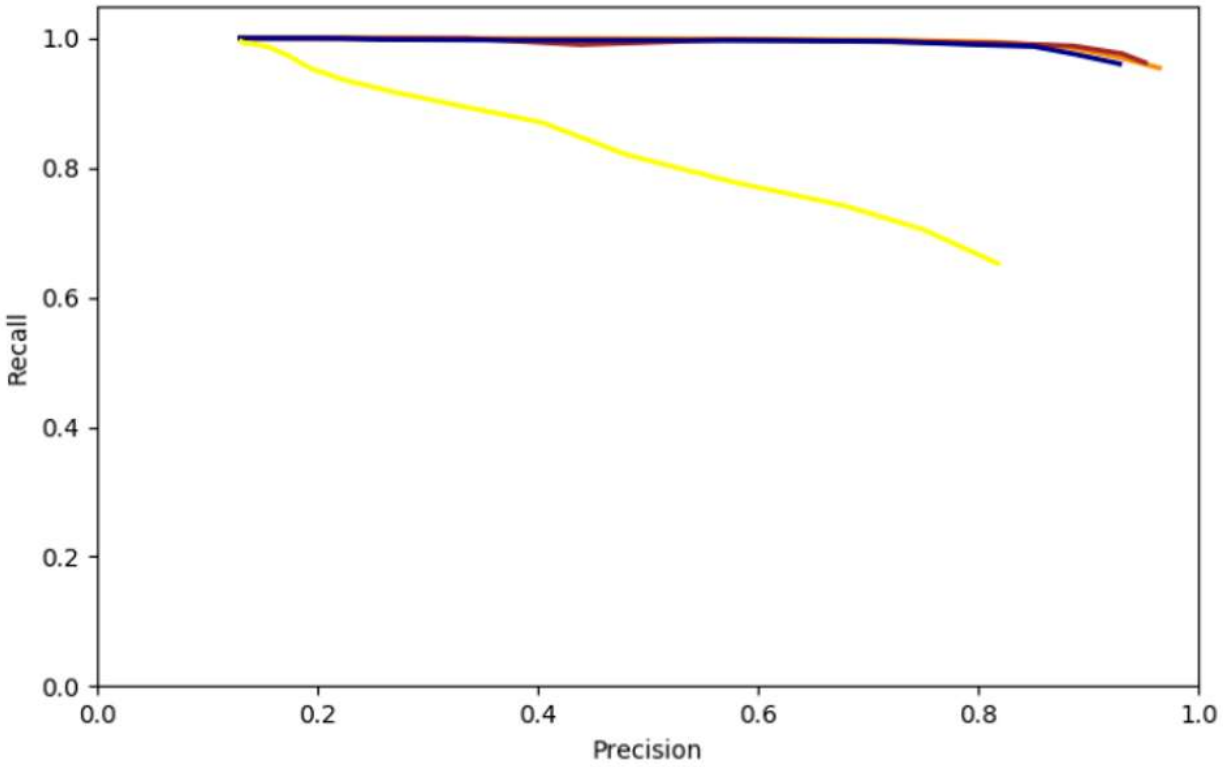


Рисунок 3.5 – Результати тестування для набору даних OJClone

Щоб дослідити ефективність запропонованого інтелектуального методу необхідно його порівняти із іншими підходами:

- Deckard – популярний підхід на основі AST, який також використовує векторний елемент для подання AST та його підрядів.

- Глибоке навчання дублікатів коду (CDLH) – підхід, що досліджує використання глибокого навчання для виявлення кодових дублікатів. Для цього використовується рекурсивний автокодер.

- SourcererCC – підхід на основі лексем. Може виявити дублікати типу III, використовуючи подання у вигляді тотокенів.

Дані підходи не потребують навчання, а отже, їх результати не повинні відрізнятися для різних тестових наборів. Під час тестування використано експериментальні загальні налаштування для експериментів з оцінки підходів з точки зору точності P, повноти R і та середнього значення F_1.

В таблиці 3.4, розроблений підхід перевершує в показниках F1 на обох наборах даних Використовуючи як структурну інформацію в AST та лексичну інформація в лексемах коду, маркер досягає збільшення F1 на OJClone та збільшення F1 на BigCloneBench, порівняно з CDLH.

CDLH нехтує інформацією про типи та використання вузлів AST тільки структура AST як керівництво для кодування кожного маркерк у послідовності в один вектор у корені; отже, він не повністю використовує структурну інформацію вихідного коду.

Навпаки, метод SourcererCC спеціально призначений для збору структурної інформації мов програмування. Структурна інформація має важливе значення для семантики коду.

Хоча SourcererCC призначений для захоплення як лексичної, так і синтаксичної інформації, підхід перевершує в наблорі OJClone, використовуючи лише синтаксичну інформацію.

Результати показують, що Deckard недостатньо охоплює синтаксичну інформацію, і найкраще фіксувати синтаксичну інформація про вихідний код явно, а не неявно.

Результати свідчать про те, що метод на основі дерев є більш ефективним ніж на основі LSTM у захопленні кодової семантики.

Один Причиною результатів може бути те, що шарів багато вузлів в AST (типовий випадок), і велика глибина навіть LSTM (призначений для поліпшення RNN щодо залежності від тривалості) не може запам’ятати інформацію, яка є на велику відстань, тоді як підхід не страждає ця проблема.

Окрім порівняння ефективності підходу з CDLH також порівняно ефективність. Завдяки повторюваній структурі, рекурентної нейронної мережі немає можливості обчислювати паралельно. В результаті навчання RNN займає досить багато часу, обмежуючи його додатки.

Таблиця 3.4 – Результати тестування методів виявлення дублікатів

	BigCloneBench			OJClone		
Метод	P	R	F_1	P	R	F_1
Deckard	0.83	0.91	0.86	0.79	0.86	0.82
CDLH	0.85	0.80	0,82	0.71	0.67	0.68
SourcererCC	0.76	0.82	0.78	0.63	0.74	0.68
AST+RNN	0.94	0.96	0.95	0.94	0.95	0.94

Головними характеристиками, які аналізувалися при тестуванні системи є точність P, повнота R і їх гармонійне середнє значення $F1$. Слід виділити характеристику F_1, яка врівноважує значення повноти і точності. Таким чином, чим більше значення F_1 – тим якісніше працює розроблена система.

- Зі збільшенням розміру проекту збільшується час, що витрачається на аналіз;

- Зі збільшенням розміру проекту несуттєво знижується якість роботи системи.

Час аналізу безпосередньо залежить від обсягу методів в вибірці. Це пов'язано з послідовною обробкою методів і відсутністю обчислювальної оптимізації.

Зниження точності обчислення пов'язано з відсутністю чіткого розмежування III і IV типів дублікатів. Таким чином, дублікати IV типу розпізнаються невірно, що і є причиною падіння точності.

З результатів тестування системи, на даних вибірки OJClone в таблиці 3.3 слід зауважити наступне:

- Зі збільшенням розміру проекту збільшується час, що витрачається на аналіз;
- Середня точність аналізу досягає 92%;

- Точність аналізу має більшу повноту в порівнянні з результатами з таблиці 3.4;

Збільшення часу аналізу пояснюється тими ж причинами, що і в попередньому випадку. Точність, в даному випадку, безпосередньо залежить від навчальної вибірки, яка містить в собі безліч неточностей і не є якісною.

Повнота обумовлюється відсутністю розмитих кордонів між дублікатами III і IV типів в даній вибірці. У навчальному наборі дублікати IV типу не розглядалися. Саме тому методи, що відносяться до дублікатів IV типу, системою не визначалися.

При розгляді швидкості роботи запропонованого методу, можна зробити висновок про її варіативность. На поточний момент, швидкість виконання нижче середньої. Однак, дана система ще не була оптимізована для досягнення максимальної швидкості роботи.

3.4 Висновок

У даному розділі було розглянуто програмну реалізацію інформаційної технології виявлення дублікатів. Здійснено опис мови програмування та середовища розробки.

Наведено загальну структуру системи і описані її основні складові: модуль побудови AST та модуль виявлення дублікатів.

Проведено тестування розробленої інформаційної технології, що реалізує запропонований метод пошуку дублікатів. Тестування показало вищі результати точності і повноти виявлення у порівнянні з існуючими аналогами.

4 ЕКОНОМІЧНА ЧАСТИНА

4.1 Оцінювання комерційного потенціалу розробки

Метою проведення технологічного аудиту є оцінювання комерційного потенціалу розробки. Для проведення технологічного аудиту було залучено 2-х незалежних експертів. Такими експертами будуть ктн., доцент Месюра В.І. та ктн., доцент Колесницький О.К.

Здійснюємо оцінювання комерційного потенціалу розробки за 12-ма критеріями за 5-ти бальною шкалою.

Результати оцінювання комерційного потенціалу розробки наведено в таблиці 4.1.

Таблиця 4.1 – Результати оцінювання комерційного потенціалу розробки

Критерії	Прізвище, ініціали, посада експерта	
	1. Експерт 1	2. Експерт 2
	Бали, виставлені експертами:	
1	4	4
2	3	3
3	4	4
4	4	4
5	3	3
6	3	4
7	3	3
8	3	4
9	4	4
10	4	3
11	3	4
12	3	4
Сума балів	$СБ_1 = 42$	$СБ_2 = 44$
Середньоарифметична сума балів $\overline{СБ}$	$\overline{СБ} = \frac{\sum_{1}^{3} СБ_i}{2} = 43$	

Отже, з отриманих даних таблиці 5.1 видно, що нова розробка має високий рівень комерційного потенціалу.

4.2 Прогнозування витрат на виконання науково-дослідної роботи та конструкторсько–технологічної роботи.

Для розробки нового програмного продукту необхідні такі витрати.

Основна заробітна плата для розробників визначається за формулою (4.1):

$$З_о = \frac{М}{Т_р} \cdot t, \qquad (4.1)$$

де М- місячний посадовий оклад конкретного розробника;

$Т_р$ - кількість робочих днів у місяці, $Т_р$ = 22 дні;

t - число днів роботи розробника, t = 40 днів.

Розрахунки заробітних плат для керівника і програміста наведені в таблиці 4.2.

Таблиця 4.2 – Розрахунки основної заробітної плати

Працівник	Оклад М, грн.	Оплата за робочий день, грн.	Число днів роботи, t	Витрати на оплату праці, грн.
Науковий керівник	5000	227,27	8	1818,16
Інженер-програміст	3600	163,63	40	6545,2
Всього:				8363,36

Розрахуємо додаткову заробітну плату:

$$З_{дод} = 0,1 \cdot 8363,36 = 836,33 \text{ (грн.)}$$

Нарахування на заробітну плату операторів НЗП розраховується як 37,5…40% від суми їхньої основної та додаткової заробітної плати:

$$Н_{зп}= (З_о + З_р)\cdot \frac{\beta}{100}, \quad (4.2)$$

$$Н_{зп}= (8363,36+836,33)\cdot \frac{36,3}{100} = 3339,48 \text{ (грн.)}.$$

Розрахунок амортизаційних витрат для програмного забезпечення виконується за такою формулою:

$$А=\frac{Ц\cdot Н_а}{100}\cdot \frac{Т}{12}, \quad (4.3)$$

де Ц – балансова вартість обладнання, грн;

$Н_а$ – річна норма амортизаційних відрахувань % (для програмного забезпечення 25%);

Т – Термін використання (Т=3 міс.).

Таблиця 4.3 – Розрахунок амортизаційних відрахувань

Найменування програмного забезпечення	Балансова вартість, грн.	Норма амортизації, %	Термін використання, міс.	Величина амортизаційних відрахувань,грн
Персональний комп’ютер	12000	25	2	500
Всього:				500

Розрахуємо витрати на комплектуючі. Витрати на комплектуючі розрахуємо за формулою:

$$K = \sum_1^n Н_і \cdot Ц_і \cdot К_і, \quad (4.4)$$

де n – кількість комплектуючих;

Ні - кількість комплектуючих і-го виду;

Ці – покупна ціна комплектуючих і-го виду, грн;

Кі – коефіцієнт транспортних витрат (приймемо Кі = 1,1).

Таблиця 4.4 – Витрати на комплектуючі, що були використані для розробки ПЗ.

Найменування матеріалу	Одиниці виміру	Ціна, грн.	Витрачено	Вартість витрачених матеріалів, грн.
Флешка	шт.	200	1	200
Пачка паперу	уп.	120	1	120
Ручка	шт.	5	1	5
Всього з урахуванням транспортних витрат				357,5

Витрати на силову електроенергію розраховуються за формулою:

$$В_е=В\cdot П\cdot Ф\cdot К_п\ ; \qquad (4.5)$$

де В – вартість 1кВт-години електроенергії (В=1,7 грн/кВт);

П – установлена потужність комп’ютера (П=0,6кВт);

Ф – фактична кількість годин роботи комп’ютера (Ф=200 год.);

$К_п$ – коефіцієнт використання потужності ($К_п< 1$, $К_п = 0,7$).

$$В_е= 1,7 \cdot 0,6 \cdot 200 \cdot 0,7 = 142,8 \text{ (грн.)}$$

Розрахуємо інші витрати $В_{ін}$.

Інші витрати $І_в$ можна прийняти як (100…300)% від суми основної заробітної плати розробників та робітників, які були виконували дану роботу, тобто:

$$В_{ін} = (1..3) \cdot (З_о + З_р). \qquad (4.6)$$

Отже, розрахуємо інші витрати:

$$В_{ін} = 1 * (8363,36 + 836,33) = 9199,69 \text{ (грн)}.$$

Сума всіх попередніх статей витрат дає витрати на виконання даної частини роботи:

$$В = З_о + З_д + Н_{зп} + А + К + В_е + І_в$$

В = 8363,36+836,33+3339,48+500+142,8+357,5+9199,69= 22739,16 (грн.)

Розрахуємо загальну вартість наукової роботи $B_{заг}$ за формулою:

$$В_{заг} = \frac{В_{ін}}{\alpha} \quad (4.7)$$

де α – частка витрат, які безпосередньо здійснює виконавець даного етапу роботи, у відн. одиницях = 1.

$$В_{заг} = \frac{22739,16}{1} = 22739,16$$

Прогнозування загальних витрат ЗВ на виконання та впровадження результатів виконаної наукової роботи здійснюється за формулою:

$$ЗВ = \frac{В_{заг}}{\beta} \quad (4.8)$$

де β – коефіцієнт, який характеризує етап (стадію) виконання даної роботи.

Отже, розрахуємо загальні витрати:

$$ЗВ = \frac{22739,16}{0,9} = 25265,73 \text{ (грн.)}$$

4.3 Прогнозування комерційних ефектів від реалізації результатів розробки

Спрогнозуємо отримання прибутку від реалізації результатів ої розробки. Зростання чистого прибутку можна оцінити у теперішній вартості грошей. Це забезпечить підприємству (організації) надходження додаткових коштів, які дозволять покращити фінансові результати діяльності .

Оцінка зростання чистого прибутку підприємства від впровадження результатів наукової розробки. У цьому випадку збільшення чистого прибутку підприємства $\Delta П_і$ для кожного із років, протягом яких очікується отримання позитивних результатів від впровадження розробки, розраховується за формулою:

$$\Delta П_i = \sum_1^n (\Delta П_я \cdot N + П_я \Delta N)_i \quad (4.9)$$

де $\Delta П_я$ – покращення основного якісного показника від впровадження результатів розробки у даному році;

N – основний кількісний показник, який визначає діяльність підприємства у даному році до впровадження результатів наукової розробки;

ΔN – покращення основного кількісного показника діяльності підприємства від впровадження результатів розробки;

$П_я$ – основний якісний показник, який визначає діяльність підприємства у даному році після впровадження результатів наукової розробки;

n – кількість років, протягом яких очікується отримання позитивних результатів від впровадження розробки.

В результаті впровадження результатів наукової розробки витрати на виготовлення інформаційної технології зменшаться на 35 грн (що автоматично

спричинить збільшення чистого прибутку підприємства на 35 грн), а кількість користувачів, які будуть користуватись збільшиться: протягом першого року – на 200 користувачів, протягом другого року – на 150 користувачів, протягом третього року – 100 користувачів. Реалізація інформаційної технології до впровадження результатів наукової розробки складала 700 користувачів, а прибуток, що отримував розробник до впровадження результатів наукової розробки – 300 грн.

Спрогнозуємо збільшення чистого прибутку від впровадження результатів наукової розробки у кожному році відносно базового.

Отже, збільшення чистого продукту $\Delta П_1$ протягом першого року складатиме:

$$\Delta П_1 = 35 \cdot 700 + (300 + 35) \cdot 200 = 91500 \text{ грн.}$$

Протягом другого року:

$$\Delta П_2 = 35 \cdot 700 + (300 + 35) \cdot (200 + 150) = 141750 \text{ грн.}$$

Протягом третього року:

$$\Delta П_3 = 35 \cdot 700 + (300 + 35) \cdot (200 + 150 + 100) = 175250 \text{ грн.}$$

4.4 Розрахунок ефективності вкладених інвестицій та період їх окупності

Визначимо абсолютну і відносну ефективність вкладених інвестором інвестицій та розрахуємо термін окупності.

Абсолютна ефективність $Е_{абс}$вкладених інвестицій розраховується за формулою:

$$Е_{абс} = (ПП - PV), \qquad (4.10)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої НДДКР, грн;

т – період часу, протягом якого виявляються результати впровадженої НДДКР, 3 роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні; для України цей показник знаходиться на рівні 0,1;

t – період часу (в роках) від моменту отримання чистого прибутку до точки 2, 3,4.

Рисунок, що характеризує рух платежів (інвестицій та додаткових прибутків) буде мати вигляд, рисунок 4.1.

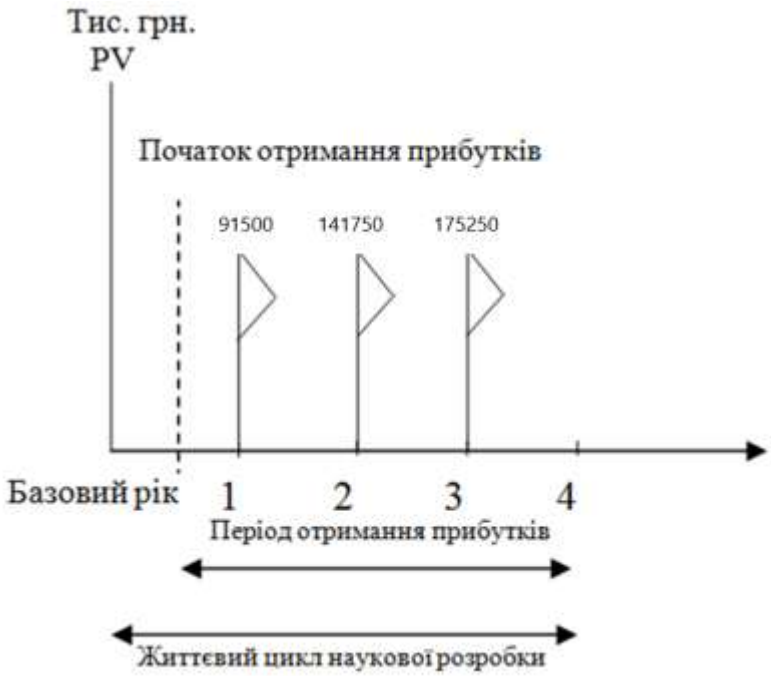


Рисунок 4.1 – Вісь часу з фіксацією платежів, що мають місце під час розробки та впровадження результатів НДДКР

Розрахуємо вартість чистих прибутків за формулою:

$$ПП = \sum_1^m \frac{\Delta П_i}{(1+\tau)^t} \quad (4.11)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої НДДКР, грн;

т – період часу, протягом якого виявляються результати впровадженої НДДКР, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні; для України цей показник знаходиться на рівні 0,1;

t – період часу (в роках) від моменту отримання чистого прибутку до точки.

Отже, розрахуємо вартість чистого прибутку:

$$ПП = \frac{25265,73}{(1+0,1)^0} + \frac{91500}{(1+0,1)^2} + \frac{141750}{(1+0,1)^3} + \frac{175250}{(1+0,1)^4} = 327082,53 \text{ (грн.)}$$

Тоді розрахуємо $E_{абс}$:

$$Е_{абс} = 327082,53 - 25265,73 = 301816,8 \text{ грн.}$$

Оскільки $E_{абс} > 0$, то вкладання коштів на виконання та впровадження результатів НДДКР буде доцільним.

Розрахуємо відносну (щорічну) ефективність вкладених в наукову розробку інвестицій $Е_в$ за формулою:

$$Е_в = \sqrt[Т]{1 + \frac{Е_{абс}}{PV}} - 1 \qquad (4.12)$$

де $Е_{абс}$ – абсолютна ефективність вкладених інвестицій, грн;

PV – теперішня вартість інвестицій PV = ЗВ, грн;

$Т_ж$ – життєвий цикл наукової розробки, роки.

Тоді будемо мати:

$$Е_в = \sqrt[3]{1 + \frac{301816,8}{25265,73}} - 1 = 1,34 \text{ або } 134\ \%$$

Далі, розраховану величину $E_в$ порівнюємо з мінімальною (бар'єрною) ставкою дисконтування $\tau_{мін}$, яка визначає ту мінімальну дохідність, нижче за яку інвестиції вкладатися не будуть. У загальному вигляді мінімальна (бар'єрна) ставка дисконтування $\tau_{мін}$ визначається за формулою:

$$\tau = d + f,$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2019 році в Україні d = 0,2;

f – показник, що характеризує ризикованість вкладень, величина f = 0,1.

$$\tau = 0{,}2 + 0{,}1 = 0{,}3$$

Оскільки $E_в$ = 134% > $\tau_{мін}$ = 0,3 = 30%, то у інвестор буде зацікавлений вкладати гроші в дану наукову розробку. Термін окупності вкладених у реалізацію наукового проекту інвестицій. Термін окупності вкладених у реалізацію наукового проекту інвестицій $T_{ок}$ розраховується за формулою:

$$T_{ок} = \frac{1}{E_в}$$

$$T_{ок} = \frac{1}{1{,}34} = 0{,}74 \text{ року}$$

Обрахувавши термін окупності даної наукової розробки, можна зробити висновок, що фінансування даної наукової розробки буде доцільним.

4.4 Висновок

В ході економічного обгрунтування розробки проведено оцінювання економічного потенціалу розробки, спрогнозовано витрати на виконання науково-дослідної, дослідно-конструкторської та конструкторськотехнологічної роботи.

ВИСНОВКИ

В ході виконання магістерської кваліфікаційної роботи досліджено предметну область виявлення дублікатів програмного коду.

В рамках дослідження були розглянуті існуючі підходи до виявлення дублікатів. Серед них були виділені ті що дозволяють здійснювати виявлення дублікатів I-III типів. Проаналізовано характеристики точності та повноти виявлення дублікатів в існуючих методах виявлення, що довело доцільність розробки гібридного методу, який підвищує точність виявлення.

Розроблений гібридний метод складається з чотирьох етапів.

Досліджено, що перетворення кодових фрагментів для виявлення дублікатів у абстрактні синтаксичні дерева з подальшою токенізанією оптимальні дані для навчання і роботи нейронної мережі.

Виявлено, що рекурентні нейронні мережі не спроможні виконувати порівняння минулих даних з новими. обґрунтовано застосування сіамської архітектури нейронної мережі для вирішення цієї проблеми.

На основі запропонованого методу була розроблена програмна реалізація інформаційної технології виявлення дублікатів програмногокоду засобами мови C# у середовищі Visual Studio.

Тестування розробленої інформаційної технології проводилось на вибірках різного розміру з набору даних BigCloneBench та OJClone. Тестування показало приріст точності виявлення на 12% та повноти виявлення на 10% в порівнянні з існуючими програмними реалізаціями методів. Тому мету роботи досягнуто.

В ході економічного обґрунтування розробки проведено оцінювання економічного потенціалу розробки, розраховано ефективність вкладених інвестицій та періоду їх окупності.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. А.В. Никуляк, В.І. Месюра Методи виявлення дублікатів програмного коду: Збірник праць X Міжнародної науково-практичної конференції «Інформаційні технології та взаємодії», (м. Київ, Україна 2019 р).

2. E. Juergens, B. Hummel, F. Deissenboeck, and M. Feilkas. Static bug detection through analysis of inconsistent clones. In Workshopband SE Konferenz 2008, LNI. GI, 2008

3. C. Kapser and M. W. Godfrey. “Cloning considered harmful” considered harmful. In Proc. WCRE ’06. IEEE, 2006

4. M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In Proc. ESEC/FSE-13. ACM, 2005

5. Bell, S. and Bala, K. Lerning Visual Similarity for Product Design with Convolutional Neural Networks. In: Proceedings of the 42nd International Conference and Exhibition on Computer Graphics and Interactive Techniques, SIGGRAPH 2015

6. Berlemont, S., Lefebvre, G., Dufiner, S. and Garcia, C. 2015. Siamese Neural Network Based Similarity Metric for Inertial Gesture Classification and Rejection. In: 11th IEEE International Conference and Workshops on Automatic Face and Gesture Recognition, FG 2015

7. Chopra, S., Hadsell, R. and LeCun, Y. 2005. Learning a Similarity Metric Discriminatively, with Application to Face Verification. In: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Patter Recognition, CVPR 2005. San Diego, CA, USA. 20 - 25 June 2005. Washington, DC, USA: IEEE Computer Society

8. Dalal, S., Athavale, V. and Jindal, K. 2011. Case Retrieval Optimisation of Case-Based Reasoning Through Knowledge-Intensive Similarity Measures. International Journal of Computer Applications, Vol 34 pp. 12 – 18

9. R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In Proc. SAS ’01, volume 2126 of LNCS. Springer, 2001

10. R. Koschke. Survey of research on software clones. In Duplication, Redundancy, and Similarity in Software. Dagstuhl Seminar Proceedings, 2007

11. Hopfield John J. Neural networks and physical systems with emergent collective computational abilities // Proceedings of the National Academy of Sciences. — 1982. — Vol. 79, no. 8. — P. 2554–2558.

12. Howard Johnson J. Identifying redundancy in source code using fingerprints. — 1993. — Jan. — p. 171–183.

13. Kapser C., Godfrey M. W. ”Cloning Considered Harmful” Considered Harmful // 2006 13th Working Conference on Reverse Engineering. — 2006. — Oct. — p. 19–28

14. Koschke Rainer, Falke Raimar, Frenzel Pierre. Clone Detection Using Abstract Syntax Suffix Trees // Proceedings of the 13th Working Conference on Reverse Engineering. — WCRE ’06. — 2006. — p. 253– 262.

15. Krinke J. Identifying similar code with program dependence graphs // Proceedings Eighth Working Conference on Reverse Engineering. — 2001. — P. 301–309

16. L. Elman Jeffrey. Finding Structure in Time // Cognitive Science. — 1990. – – Vol. 14, no. 2. — P. 179–211.

17. Mallaiah Sudhamani, Rangarajan Lalitha. Code Similarity Detection by Computing Block Level Feature Metrics // nternational Journal of Computer Application (2250-1797). — 2018. — Vol. 2.

18. Mueller Jonas, Thyagarajan Aditya. Siamese Recurrent Architectures for Learning Sentence Similarity // Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. — AAAI’16. — AAAI Press, 2016. — P. 2786–2792