МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 17.00.00.000 ПЗ

Група ІПм-23-2

Олексяк Назар

2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Олексяк Назар Тарасович

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

МАГІСТЕРСЬКА РОБОТА

Методи класифікації узгодженості та послідовності коду засобами мовних моделей

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Олексяк Н.Т.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Вовк Роман Богданович, к.т.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2024 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Олексяку Назару Тарасовичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “ Методи класифікації узгодженості та послідовності коду засобами мовних моделей”

керівник проекту (роботи) Вовк Роман Богданович, к.т.н., доцент

затверджені наказом закладу вищої освіти від “ 22 ” листопада 2024 р. № 781/7

2. Строк подання студентом проекту (роботи) 15 грудня 2024 р.

3. Вихідні дані до проекту (роботи) Теоретичні концепції та формальні моделі побудови та функціонування інформаційних технологій мовних моделей

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Аналіз предметної області методів класифікації узгодженості програмного коду
2. Моделі та алгоритми класифікації узгодженості коду з використанням мовних моделей
3. Дослідження та аналіз процесів узгодженості коментарів коду
4. Методологія класифікації узгодженості та послідовності коду засобами мовних моделей

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Типи мовних моделей (рис. 1.1)
2. Виявлення узгодженості коментарів коду (рис. 2.1)
3. Приклад перевіреного запиту на отримання на GitHub (рис. 2.2)
4. Кількість репозитріїв програмного забезпечення за мовами на GitHub (рис. 2.3)
5. Хмари слів за мовами програмування (рис. 2.4)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц., к.т.н. Вовк Р.Б.	

7. Дата видачі завдання 04 вересня 2024 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі магістерської роботи	15.09.2024	виконано
2	Аналіз концепцій та алгоритмів предметної області	29.09.2024	виконано
3	Аналіз предметної області методів класифікації узгодженості програмного коду	15.10.2024	виконано
4	Моделі та алгоритми класифікації узгодженості коду з використанням мовних моделей	08.11.2024	виконано
5	Дослідження та аналіз процесів узгодженості коментарів коду	20.11.2024	виконано
6	Методологія класифікації узгодженості та послідовності коду засобами мовних моделей	01.12.2024	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2024	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 80 с., 18 рис., 12 табл., 52 джерела.

Тема: Методи класифікації узгодженості та послідовності коду засобами мовних моделей

Об'єкт дослідження: процеси підтримки та забезпечення узгодженості коментарів і коду в системах з відкритим вихідним кодом.

Мета роботи: розробка методології автоматизованої класифікації узгодженості коментарів і коду за допомогою мовних моделей, а також оцінка її ефективності на проєктах з відкритим вихідним кодом.

Предмет дослідження: методи та моделі класифікації узгодженості коментарів з кодом, зокрема, застосування великих мовних моделей (наприклад, CodeBERT) для автоматизованого аналізу та оцінки якості коментарів.

Результати дослідження

Розроблено методологію класифікації узгодженості коментарів із кодом за допомогою великих мовних моделей, що дозволяє автоматизувати цей процес на великих наборах даних і виявлено аномалії в існуючих наборах даних для мов програмування.

Висновок

Запропоновані методи можуть бути використані для автоматизованого аналізу та класифікації коментарів у проєктах з відкритим кодом, що значно спрощує процеси ревізії коду і підвищує якість програмного забезпечення.

МОВНІ МОДЕЛІ, КЛАСИФІКАЦІЯ КОМЕНТАРІВ, УЗГОДЖЕНІСТЬ КОМЕНТАРІВ І КОДУ, МАШИННЕ НАВЧАННЯ, РЕВІЗІЯ КОДУ, GITHUB, ВІДКРИТИЙ КОД, АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

ABSTRACT

Master Thesis: 80 pp., 18 fig., 12 tab., 52 sources.

Thesis Subject: Methods of classification of consistency and componentity of code using language models

Research object: processes for maintaining and ensuring the suitability of comments and code in open source systems.

The purpose of the work: development of a methodology for automated classification of the consistency of comments and code using language models, as well as evaluation of its effectiveness on open source projects.

Research subject: methods and models for classification of comments consistency with code, in particular, application of large language models (for example, CodeBERT) for automated analysis and evaluation of the quality of comments.

Research results

A methodology for classifying the consistency of comments with code using large language models has been developed, allowing to automate this process on large datasets, and anomalies in existing language programming datasets have been identified.

Conclusion

The proposed methods can be used for automated analysis and classification of comments in open source projects, which greatly simplifies code review processes and software quality.

LANGUAGE MODELS, COMMENT CLASSIFICATION, COMMENT AND CODE CONSOLIDATION, MACHINE LEARNING, CODE REVIEW, GITHUB, OPEN SOURCE, SOFTWARE ANALYSIS.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

GNN - Graph Neural Network

MTPB - Multi-Turn Programming Benchmark

LLM - Large Language Model

AST - Abstract Syntax Tree

NLP - Natural Language Processing

ML - Machine Learning

DL - Deep Learning

PLM - Pre-trained Language Model

GPT - Generative Pre-trained Transformer

BERT - Bidirectional Encoder Representations from Transformers

NL2Code - Natural Language to Code

CLM - Code Language Model

CS - Code Summarization

CCL - Consistency Checking in Large codebases

CoT - Chain of Thought (method for reasoning in LLMs)

MRT - Model Reuse in Testing

RNN - Recurrent Neural Network

CNN - Convolutional Neural Network

TL - Transfer Learning

POS - Part of Speech (used in code tokenization)

ВСТУП

Актуальність теми.

Актуальність дослідження класифікації узгодженості коментарів та коду обумовлена швидким зростанням обсягів і складності сучасного програмного забезпечення, особливо в контексті проєктів з відкритим вихідним кодом, де велика кількість розробників взаємодіє у децентралізованих середовищах. Коментарі в коді відіграють критично важливу роль у забезпеченні його зрозумілості, підтримки та подальшого розвитку, але часто вони є неповними, застарілими або невідповідними до функцій, які пояснюють. Це створює серйозні проблеми для команд розробників, оскільки некоректні або непослідовні коментарі можуть стати джерелом помилок, знижувати продуктивність роботи і ускладнювати розуміння коду, що, в свою чергу, впливає на загальну якість програмного продукту.

З появою великих проєктів з відкритим вихідним кодом, таких як проєкти на GitHub, обсяги коду та коментарів до нього зростають експоненціально. Підтримка та ревізія таких коментарів стають надзвичайно трудомісткими завданнями, що вимагають автоматизації. Застосування сучасних методів машинного навчання та великих мовних моделей, таких як CodeBERT, відкриває нові можливості для автоматизації процесів перевірки коментарів. Використання цих моделей дозволяє автоматично виявляти невідповідності між коментарями та кодом, що зменшує ризик людських помилок і прискорює процес ревізії. Це особливо важливо для великих та активних проєктів, де значна кількість коментарів та їх узгодженість з кодом потребують постійного контролю.

Окрім того, автоматизація аналізу коментарів є важливою з точки зору якості програмного забезпечення, адже невідповідність коментарів може спричиняти труднощі для нових членів команд, студентів чи співробітників, які долучаються до проєктів. Це також актуально для розробників

інструментів для автоматизації процесів ревізії коду та перевірки запитів на внесення змін (pull request'ів), які активно використовуються в проєктах з відкритим вихідним кодом. Тому створення нових інструментів і методологій, що спрощують процес аналізу коментарів і їх узгодженості з кодом, має значний потенціал для покращення практик розробки програмного забезпечення в цілому.

Таким чином, актуальність теми полягає в необхідності розробки інструментів та методів автоматизованої перевірки коментарів у коді, що може значно підвищити якість програмного забезпечення, спростити процеси ревізії і полегшити підтримку великих проєктів, особливо в умовах відкритого вихідного коду та активної розробки.

Мета дослідження - розробка методології автоматизованої класифікації узгодженості коментарів і коду за допомогою мовних моделей, а також оцінка її ефективності на проєктах з відкритим вихідним кодом.

Об'єкт дослідження - процеси підтримки та забезпечення узгодженості коментарів і коду в системах з відкритим вихідним кодом.

Предмет дослідження - методи та моделі класифікації узгодженості коментарів з кодом, зокрема, застосування великих мовних моделей (наприклад, CodeBERT) для автоматизованого аналізу та оцінки якості коментарів.

Відповідно до мети роботи було сформовано наступні **задачі:**

- Провести аналіз існуючих підходів до класифікації узгодженості коментарів з кодом.
- Розробити методологію автоматизованого вибору та аналізу коментарів на основі мовних моделей.
- Навчити та протестувати мовні моделі, такі як CodeBERT, на запропонованих наборах даних.
- Проаналізувати ефективність моделей на різних мовах програмування та порівняти їх результати з попередніми підходами.
- Виявити можливі аномалії в існуючих наборах даних.

- Оцінити можливість застосування моделей для автоматизації ревізії коментарів у сучасних проєктах з відкритим кодом.

Методи дослідження.

У роботі застосовуються такі методи дослідження:

- Машинне навчання для навчання моделей класифікації узгодженості коментарів.

- Методи видобування даних (data mining) для збору та обробки наборів даних з відкритих джерел на GitHub.

- Методи лінгвістичного аналізу для аналізу коментарів у коді.

- Методи кількісного та якісного аналізу для оцінки ефективності моделей і виявлення аномалій у наборах даних.

Наукова новизна отриманих результатів

Розроблено методологію класифікації узгодженості коментарів із кодом за допомогою великих мовних моделей, що дозволяє автоматизувати цей процес на великих наборах даних і виявлено аномалії в існуючих наборах даних для мов програмування, зокрема для Java, що дозволило підвищити точність оцінки моделей.

Практичне значення магістерської роботи

Запропоновані методи можуть бути використані для автоматизованого аналізу та класифікації коментарів у проєктах з відкритим кодом, що значно спрощує процеси ревізії коду і підвищує якість програмного забезпечення. Створені набори даних можуть бути використані для навчання нових моделей генерації коментарів або автоматизації ревізії pull request'ів.

Структура магістерської роботи. Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 80 сторінок, і містить 18 рисунків, 12 таблиць, список використаних джерел із 52 позицій.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ МЕТОДІВ КЛАСИФІКАЦІЇ УЗГОДЖЕНОСТІ ПРОГРАМНОГО КОДУ

1.1. Опис проблемної області дослідження

Коментарі в програмному коді призначені для того, щоб допомагати розробникам розуміти код. На відміну від самого коду, який виконується машиною, коментарі в першу чергу пишуться для людей. Попередні дослідження показали, що розуміння коду домінує в часі професійного розробника, що підкреслює важливість вивчення коментарів. Вивчення узгодженості коментарів коду включає оцінку того, чи написані коментарі узгоджені з кодом, з яким вони пов'язані. У цій роботі ми досліджуємо та застосовуємо великі мовні моделі до проблеми виявлення несумісних коментарів. Наш проект вирішує основну прогалину в попередніх роботах, оскільки 90% попередніх досліджень були зосереджені на Java. Відтворюючи минулі результати, ми також виявляємо значний артефакт анотації, який, ймовірно, призвів до вводячих в оману результатів у минулому. Ми виявляємо, що простий класифікатор, який прогнозує лише на основі кількості нових рядків у кінці прикладу, досягне 80% точності.

Для цього, ми створюємо новий набір даних із понад 80 000 прикладів пар коду та коментарів у 4 мовах програмування: Java, Python, Go та JavaScript. Ми виявляємо, що точно налаштовані версії CodeBERT і Codegen перевершують раніше розроблені глибокі нейронні мережі у всіх вивчених мовах. Крім того, ми вносимо новий вручну відібраний набір з 100 прикладів несумісних коментарів з реальних pull-requests і проводимо дослідження, подаючи 20 pull-requests, які виправляють виявлені проблеми. Цей набір бенчмарків було побудовано шляхом аналізу нововидобутого набору даних з понад 13 мільйонів коментарів з GitHub pull-requests. На цьому створеному вручну наборі оцінки бенчмарків ми спостерігаємо нижчу продуктивність усіх моделей, ніж спостерігалася під час навчання, що підкреслює

необхідність враховувати реальні умови під час побудови моделей. Щоб вивчити, як розробники оцінюють вирішення проблеми несумісних коментарів, ми надіслали 20 pull-requests до проектів з відкритим кодом на GitHub. З 90% прийнятих pull-requests ми виявляємо, що розробники сприймають автоматичне виявлення узгодженості коментарів, підтверджуючи його призначення в середовищі з відкритим кодом.

1.2. Особливості коду і невідповідності коментарів до нього

У розробці програмного забезпечення коментарі — це текстові анотації, вбудовані в код для блага людей. У той час як сам код інтерпретується машиною у виконувані інструкції, коментарі у вихідному коді ігноруються машиною і в більшості випадків надають цінність лише іншим людям. Оскільки ці коментарі ігноруються комп'ютером і тому не потребують дотримання жодних структурних чи синтаксичних правил, вони можуть значно відрізнятися за призначенням і стилем.

Лістинг 1.1. Приклад документаційного коментарю до Python

```
def gcd(a, b):
    """Find the greatest common divisor of integers (a,b).
    """
    m = min(a, b)
    for i in range(m, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
```

Лістинг 1.1 показано приклад коментарю на мові Python. Цей тип коментаря відомий як коментар до документації, який загалом описує загальну функціональність функції, не вдаючись у конкретні деталі її реалізації. Переважно цей тип коментарів зазвичай розміщується безпосередньо перед визначенням функції або класу і містить інформацію про її призначення, параметри, повертаєме значення та інші деталі, які

можуть бути корисними для інших розробників, які будуть використовувати цей код.

Лістинг 1.2, також написаний на Python, представляє вбудований коментар. Цей тип коментаря зазвичай пишеться, щоб надати більш детальну документацію щодо конкретних рядків коду, наприклад, щоб пояснити причини певного вибору або надати більш зрозумілу альтернативу особливо складному розділу. У цьому прикладі математична формула, що включає віднімання та оператор модуля, пояснюється словами.

Лістинг 1.2. Приклад вбудованого коментарю до Python

```
def pad_with_zeros(str_num):
    '''
    Return a string with zeros inserted at the start of the string str_num
    such that its length is divisible by 3.
    '''
    if len(str_num) % 3 == 0:
        return str_num
    else:
        # Insert either 1 or 2 '0' characters by taking the remainder of division by 3
        # and turning 1 into 2, 2 into 1.
        return '0'*(3 - len(str_num) % 3) + str_num
```

Ключове розуміння, яке лежить в основі мотивації цього проекту, полягає в тому, що в реальних промислових програмах розробники витрачають набагато більше часу на читання коду, ніж на його написання. Насправді до 83% програмного часу професійного розробника витрачається на навігацію та розуміння коду. У компанії кожен рядок коду пишеться один раз, але його повинен розуміти кожен розробник, який його використовуватиме. Щоб прискорити процес розуміння, дисципліновані програмісти додадуть коментарі до свого коду. Коментарі в коді є операторами природної мови, які ігноруються під час виконання програми та призначені виключно для вигоди читача. Тому добре написаний і оновлений коментар може дозволити читачеві зрозуміти призначення сусіднього коду без необхідності повністю відстежувати кожен рядок, який буде виконувати комп’ютер.

З іншого боку, погано написані, помилкові або застарілі коментарі можуть призвести до величезного розчарування та накопичення технічної роботи. Коли інші розробники читають такі коментарі, вони припускають, що поведінка відповідає письмовому опису, але потім здивуються, коли програма виконує щось інше. У деяких випадках вони можуть навіть не помітити спочатку, що призведе до ще більших труднощів усунення помилок у майбутньому, оскільки все більше і більше шарів коду будується на ньому, перш ніж помилку врешті-решт буде виявлено.

У сучасній індустрії програмного забезпечення компанії обирають використовувати мови програмування через їхні сильні та слабкі сторони. Така спеціалізація інструментів означає, що навіть в одній компанії кілька мов програмування можуть використовуватися для різних цілей. Однак існуючі дослідження в галузі аналізу коментарів до програмного забезпечення зосереджені на мові Java, де специфічний стиль коментування та шаблони розробки можуть не переноситись на інші мови. Наприклад, коментарі методів у Java часто частково генеруються інтегрованими середовищами розробки (IDE), які забезпечують єдиний стиль і послідовні стандарти навіть для різних проектів. Java також має тенденцію мати довші імена змінних, ніж інші мови, які мають більшу увагу на стислість.

Наскільки ми могли визначити, не існує доступного та позначеного набору даних пар коду та коментаря для мов, окрім Java. Створення таких наборів даних може заохотити майбутню роботу над темою вивчення узгодженості коментарів коду в більш широкому діапазоні мов.

1.3. Можливість використання мовних моделей для коментування програмного коду

Використання великих мовних моделей (LLMs), таких як GPT або BERT, для коментування програмного коду є перспективним напрямком розвитку автоматизації в програмній інженерії. Такі моделі здатні

аналізувати код, генерувати зрозумілі пояснення для його функцій, а також створювати відповідні коментарі, що можуть покращити підтримку і зрозумілість програмних проектів.

Розглянемо основні можливості використання мовних моделей для коментування коду:

1. Автоматична генерація коментарів. LLMs здатні генерувати пояснення для функцій або блоків коду на основі їх синтаксичної та семантичної структури. Це допомагає створювати коментарі, які описують призначення функцій, змінних, умовних операторів та інших конструкцій.

2. Покращення розуміння коду. Лінгвістичні моделі можуть аналізувати не лише окремі функції, але й узгоджувати їх з архітектурою всієї програми. Це дозволяє створювати комплексні коментарі, які пов’язують окремі частини коду та пояснюють їх взаємодію в рамках проекту.

3. Виявлення неявних залежностей. Моделі можуть знаходити приховані взаємозв’язки між функціями або бібліотеками, які можуть не бути очевидними на перший погляд, і додавати коментарі, що вказують на ці залежності. Це допомагає розробникам уникати помилок під час рефакторингу або оптимізації коду.

4. Аналіз та рефакторинг коду. Мовні моделі можуть оцінювати якість коду, пропонувати можливі покращення та автоматично додавати коментарі, що вказують на необхідні зміни або потенційні помилки. Наприклад, вони можуть запропонувати переписати фрагменти коду для підвищення ефективності.

5. Пояснення складного коду. Для нових розробників або тих, хто не брав участі у початковій розробці проекту, LLM можуть стати важливим інструментом для пояснення складних або неочевидних алгоритмів. Це значно спрощує процес підтримки програмного забезпечення та підвищує ефективність командної роботи.

6. Підтримка багатомовності. LLM здатні генерувати коментарі різними мовами, що корисно для міжнародних команд розробників. Вони можуть перекладати вже наявні коментарі або створювати нові на основі вихідного коду.

Але, в даному випадку є і ряд обмежень:

- Точність генерації. Мовні моделі не завжди можуть точно зрозуміти контекст складних програмних конструкцій, що може призвести до генерації неправильних або неточних коментарів. Потрібен додатковий контроль з боку розробників.

- Контекстуальність та абстракція. LLMs можуть мати труднощі з правильним узагальненням складних абстракцій у великих програмах, оскільки багато кодових структур мають контекст, який не завжди можна розпізнати лише з аналізу синтаксису.

- Вимоги до обчислювальних ресурсів. Обробка великих кодових баз мовними моделями може вимагати значних обчислювальних потужностей, що робить цей підхід ресурсомістким для великих проектів.

Мовні моделі (LM) є важливим підходом до покращення мовного інтелекту машин. Загалом, LM включає моделювання ймовірності послідовностей слів для прогнозування ймовірності майбутнього.

Як видно з рис. 1.1, дослідження LM отримали широку увагу і пройшли чотири значні етапи розвитку, наступним чином: Першим основним розвитком у LM були Статистичні Мовні Моделі (SLMs), такі як n-грамові моделі. Ці моделі оцінюють ймовірність наступного слова в послідовності на основі частоти появи попередніх n-грам слів. Наприклад, біграмова модель використовуватиме частоту появи пар слів для оцінки ймовірності наступного слова.

Другий етап розвитку LM включав представлення заснованої на нейронних мережах LM, названої Нейронні Мовні Моделі (NLMs), що також відома як нейронне моделювання мови.

Четвертий етап розвитку LM включав створення великомасштабних моделей передтренування мови, які називаються Великі Мовні Моделі (LLM), які мають здатність виконувати різноманітні завдання обробки природної мови (NLP) з відмінною продуктивністю. Ці моделі, такі як GPT-3 і GPT-4, навчаються на величезних обсягах текстових даних і можуть бути донавчаються для конкретних завдань нижнього рівня, таких як переклад мови або відповіді на запитання.

Підсумовуючи, ці чотири етапи розвитку LM представляють значні досягнення в цій галузі, причому кожен етап будується на попередньому і розширює межі того, що машини можуть досягти в NLP і комп'ютерному зорі. У цьому дослідженні ми головним чином зосереджуємося на LLM. Ми ієрархічно розділили типи LLM на чотири основні категорії (рис. 1.1).

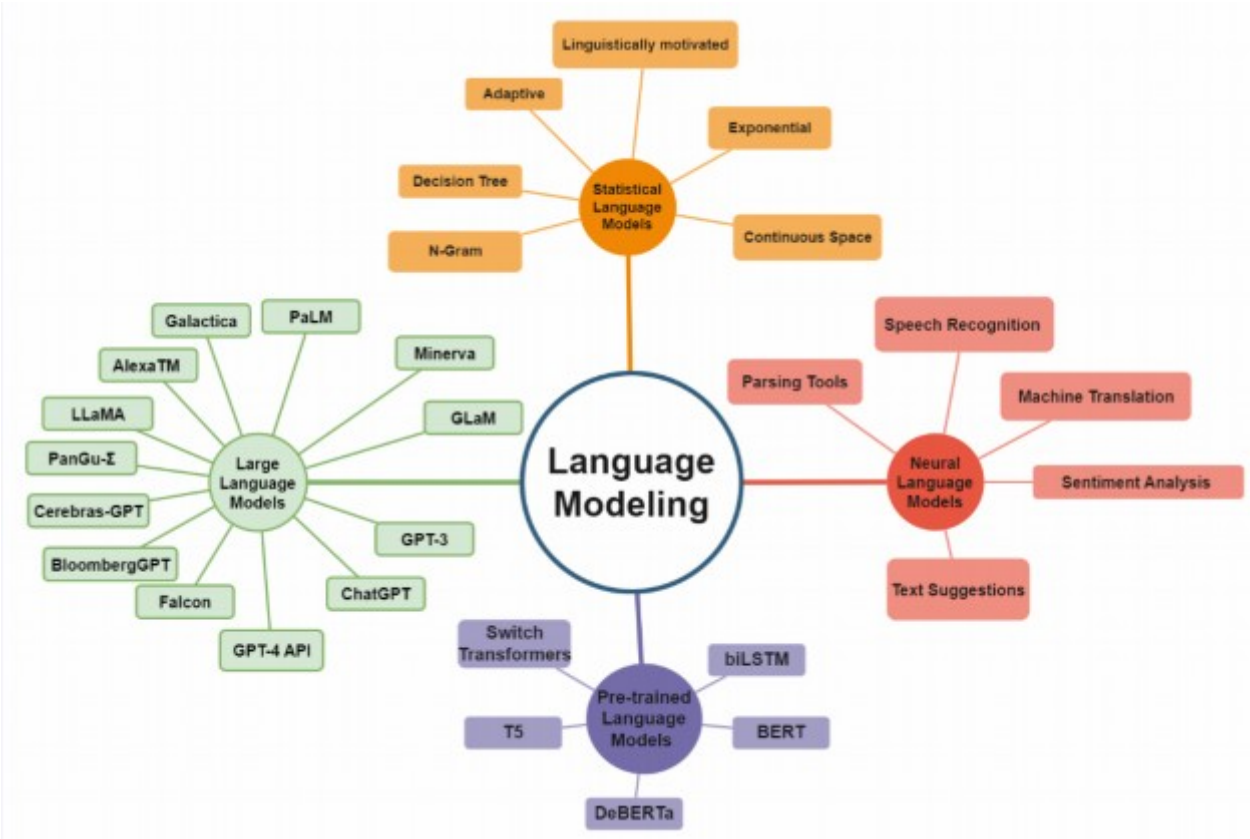


Рис. 1.1. Типи мовних моделей

Сучасна мовна модель під назвою ChatGPT була створена OpenAI. Вона базується на архітектурі GPT-3.5 і була навчена за допомогою значної

кількості текстових даних, отриманих з Інтернету, включаючи книги, статті, вікі та веб-сайти II. ChatGPT є винятковим у створенні людських відповідей і веденні розмов з користувачами. У комп'ютерному зорі (CV) дослідники активно займаються розробкою моделей мови-зору, натхненних можливостями ChatGPT. Ці моделі спеціально розроблені для покращення мультимодальних діалогів, де важлива як візуальна, так і текстова інформація.

Крім того, прогрес у цій галузі призвів до представлення GPT-4, що ще більше розширило можливості мовних моделей шляхом безперешковної інтеграції візуальної інформації як частини введення. Ця інтеграція візуальних даних надає моделі можливість ефективно розуміти та генерувати відповіді, які включають як текстові, так і візуальні підказки, що дозволяє вести більш контекстно насичені та нюансовані розмови в мультимодальних умовах.

Великі мовні моделі — це математичні представлення мови, які здатні робити текстові передбачення, наприклад, наступне слово в реченні або позитивний чи негативний відгук даного тексту. Велике значення в назві означає кількість числових параметрів у представленні, як правило, щонайменше кілька сотень мільйонів значень. Наприклад, ChatGPT є великою мовною моделлю. Враховуючи нещодавні успіхи у застосуванні великих мовних моделей [24] у налаштуваннях аналізу тексту, ми бачимо потенціал у використанні їх для завдання виявлення непослідовних коментарів, таким чином покращуючи якість програмного забезпечення та досвід розробки.

Більшість досліджень у класифікації коментарів у вихідному коді використовували евристичні методи, засновані на таких факторах, як кількість слів, що збігаються між коментарем і функціональним кодом, залишаючи простір для дослідження методів машинного навчання [30]. Хоча використання великої мовної моделі є це не гарантований метод успіху в

будь-якому випадку, ми мотивовані успіхом техніки в інших контекстах і бачимо її потенціал у цій проблемі також.

Отже, використання великих мовних моделей для коментування коду відкриває нові можливості для автоматизації програмної інженерії. Моделі можуть значно покращити процес документації коду, зробити його зрозумілішим для інших розробників, а також допомогти виявляти потенційні проблеми у структурі програмного забезпечення. Однак, для досягнення максимальної ефективності необхідний контроль і корекція з боку розробників, оскільки точність таких моделей не завжди є ідеальною.

Висновки до розділу

У першому розділі було здійснено детальний аналіз проблемної області, пов’язаної з класифікацією узгодженості програмного коду та коментарів до нього. Аналіз показав, що узгодженість між кодом та його коментарями є важливим аспектом для підтримки якості програмного забезпечення, але на практиці часто спостерігається проблема невідповідності між функціональністю коду та коментарями, що описують його роботу.

Розглянуто особливості програмного коду, зокрема його структуру та виклики, пов'язані з підтримкою його зрозумілості й читабельності. Відзначено, що невідповідність між кодом і коментарями може бути результатом таких факторів, як зміни в коді без оновлення документації, нестача чітких стандартів для коментування або недостатня увага до коментарів під час розробки.

У цьому контексті можливість використання великих мовних моделей (LLMs) для автоматичного коментування коду розглянуто як перспективний напрямок. Мовні моделі, такі як GPT, демонструють здатність аналізувати код, генерувати відповідні коментарі та виявляти невідповідності між коментарями та функціональністю коду. Вони можуть значно спростити

процес документації та допомогти підтримувати узгодженість коментарів із фактичним кодом. Однак, вказується на необхідність подальших досліджень для підвищення точності та контекстуальності таких інструментів.

Таким чином, у розділі було визначено, що проблеми узгодженості коду та коментарів є значним викликом для розробників, а використання сучасних інструментів на основі мовних моделей може стати ефективним рішенням цієї проблеми, забезпечуючи більш точну та ефективну документацію програмного забезпечення.

РОЗДІЛ 2. МОДЕЛІ ТА АЛГОРИТМИ КЛАСИФІКАЦІЇ УЗГОДЖЕНОСТІ КОДУ З ВИКОРИСТАННЯМ МОВНИХ МОДЕЛЕЙ

2.1. Особливості процесу документування програмного коду

У цьому розділі ми формулюємо проблему та представляємо питання дослідження. На рисунку малюнку 2.1 ми ілюструємо головну мету виявлення узгодженості коментарів з точки зору його вхідних і вихідних даних.

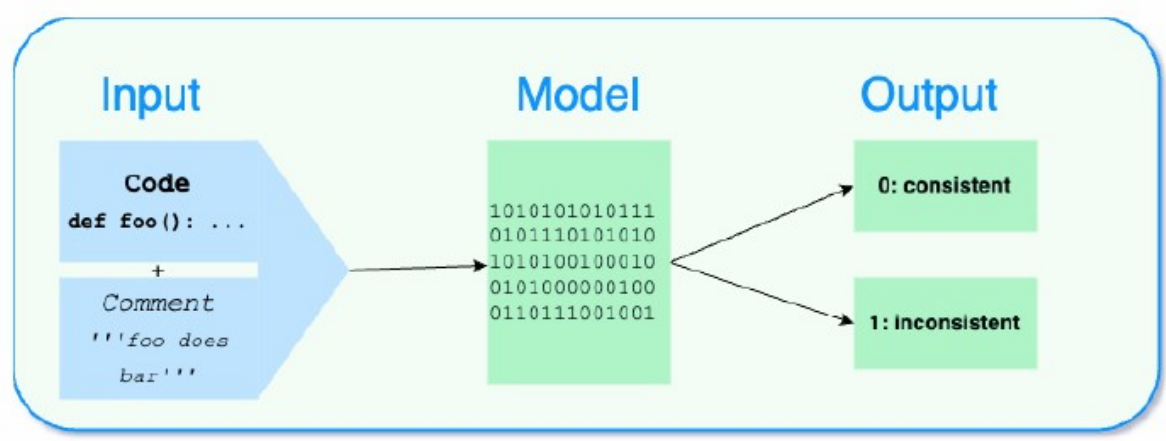


Рис. 2.1. Виявлення узгодженості коментарів коду

Важливим компонентом ефективної розробки програмного забезпечення є документування вихідного коду. Найбільш трудомісткою діяльністю професійного розробника є навігація та розуміння коду, а коментарі в коді безпосередньо допомагають із цим завданням. У міру розвитку проектів програмного забезпечення пишеться новий код, а старий змінюється. Однак, хоча виконання програмного забезпечення перевіряє функцію коду, немає механізму, який би гарантував, що коментарі залишаються узгодженими з кодом, який вони описують.

Проблема, яку намагається вирішити дана робота, полягає в тому, як застосувати великі мовні моделі до реального світу проблеми виявлення невідповідності коментарі в коді на кількох мовах програмування.

Означення 2.1. Коментар – це узгодженим (консистентним) з пов’язаним кодом, якщо кваліфікований рецензент-експерт явно вважає це таким.

Це визначення охоплює широкий спектр потенційних джерел невідповідностей. Кваліфікований рецензент-експерт має бути особою, яка не є автором зміни коду, але також має знання про проект, автором якого була зміна. Поняття консистенції, запропоноване в цьому формулюванні, найбільше відповідає визначенню когерентності введеному в [37]. Хоча кожен рецензент матиме свої власні стандарти, ми очікуємо, що це визначення узгодить кілька загальнодосліджуваних показників оцінки, включаючи зрозумілість, повноту та точність.

Розглянемо приклад різниці між послідовним і непослідовним коментарем у лістингу 2.2. У цьому прикладі ми відтворили правильний приклад лістингу 1.1 разом із версією зі зміненим коментарем, щоб надати приклад як узгодженого, так і непослідовного коментаря. Коментар непослідовний, оскільки код повертає найбільший спільний дільник, але коментар стосується найменшого спільного кратного.

Лістинг 2.1. Приклад послідовного коментарю

```
def gcd(a, b):
    """Find the greatest common divisor
       of integers (a,b).
    """
    m = min(a, b)
    for i in range(m, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
```

Лістинг 2.2. Приклад непослідовного коментарю

```
def gcd(a, b):
    """Find the least common multiple of
       integers (a,b).
    """
    m = min(a, b)
    for i in range(m, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
```

Однак, хоча означення 2.1 дає ідеальний метод визначення того, чи є коментар послідовним, він не є практичним у контексті навчання з учителем. Оскільки це вимагало б звертатися до експерта за їхньою думкою для кожного можливого прикладу, ми не змогли б створити достатньо велику базу даних для навчання моделі. Тому ми вводимо наступне визначення, запропоноване в [28], яке більше підходить для контексту слабкого контролю.

Означення 2.2. Враховуючи зміну в системі контролю версій, яка змінює як код функції, так і її документаційний коментар, ми визначаємо старий коментар як непослідовний з новим кодом, а новий код як послідовний з новим коментарем.

По суті, одним з результатів цього проєкту є вимірювання узгодженості між цими двома визначеннями. Ця робота слідує моделі попередніх досліджень, припускаючи, що означення 2.2 може замінити означення 2.1. У цій роботі ми, таким чином, називаємо означення 2.1 "золотим стандартом", а означення 2.2 — "срібним стандартом".

Ще один важливий термін для уточнення в постановці проблеми — це "реальний світ". Зокрема, у цьому випадку термін "реальний світ" стосується максимально точного співставлення з очікуваним випадком використання розробки програмного забезпечення. Для цього проєкту обраним випадком використання є рецензування коду. Рецензування коду — це стандартна практика в індустрії програмного забезпечення, коли хтось, окрім автора частини коду, переглядає його та вирішує, чи схвалити його чи ні. На рисунку 2.2 ми представляємо приклад pull-запиту, який був прийнятий під час рецензії. У цьому випадку r+ — це скорочений вираз, що означає схвалення. На цьому етапі одним з аспектів, що враховуються, є якість коментарів. На практиці мета цього проєкту полягає в створенні додаткового рівня рецензування коду, який спеціалізується на перевірці коментарів. Це важливо, оскільки, навіть після рецензії коду, дослідники спостерігали приклади комітів коду, які виправляли лише помилкові коментарі. Це явище

свідчить про те, що інструмент, який може надійно розпізнавати послідовні та непослідовні коментарі під час рецензування, покращив би продуктивність розробників, зменшуючи час, витрачений на виправлення попередніх помилок. Таким чином, реальна оцінка перевірки послідовності коментарів передбачала б її тестування на прикладах сценаріїв рецензування коду.

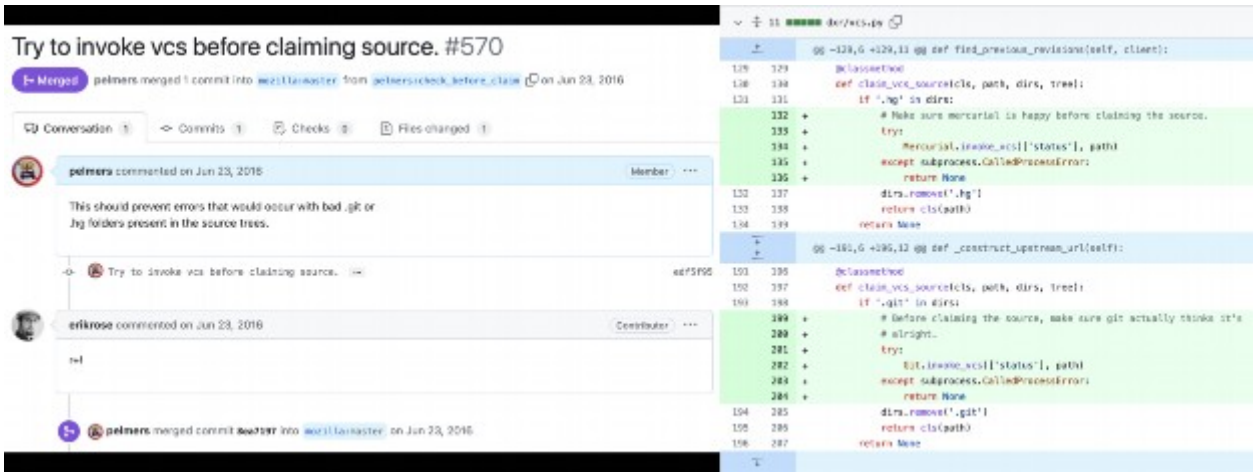


Рис. 2.2. Приклад перевіреного запиту на отримання на GitHub

2.1.1. Вибір мови програмування для дослідження

Кожна мова програмування має власний синтаксис і умовності. Модель, навчена коду Python, швидше за все, не працюватиме так добре, коли вона отримає вхідний код JavaScript. Наприклад, коментарі Java відповідають чітко визначеному стандарту Javadoc [38] і з'являються перед виразом функції, тоді як коментарі функції Python відформатовані по-різному, записуються у вигляді рядка після рядка визначення.

З огляду на попередні роботи над мовою Java, однією з цілей цього проекту є узагальнення моделі на Python, JavaScript і Go. Цей вибір мови зумовлений популярністю в спільноті відкритих кодів. Вибір мов за популярністю обіцяє переваги найширшої застосовності та найбільшої доступності навчальних даних. Попередньо для цього проекту ми видобули метадані для понад 3 мільйонів сховищ GitHub [6], які складаються з усіх загальнодоступних сховищ із принаймні 5 зірками на GitHub з 2009 року до

травня 2023 року. На рисунку 2.3 зображено стовпчасту діаграму кількості репозиторіїв за мовами на GitHub.

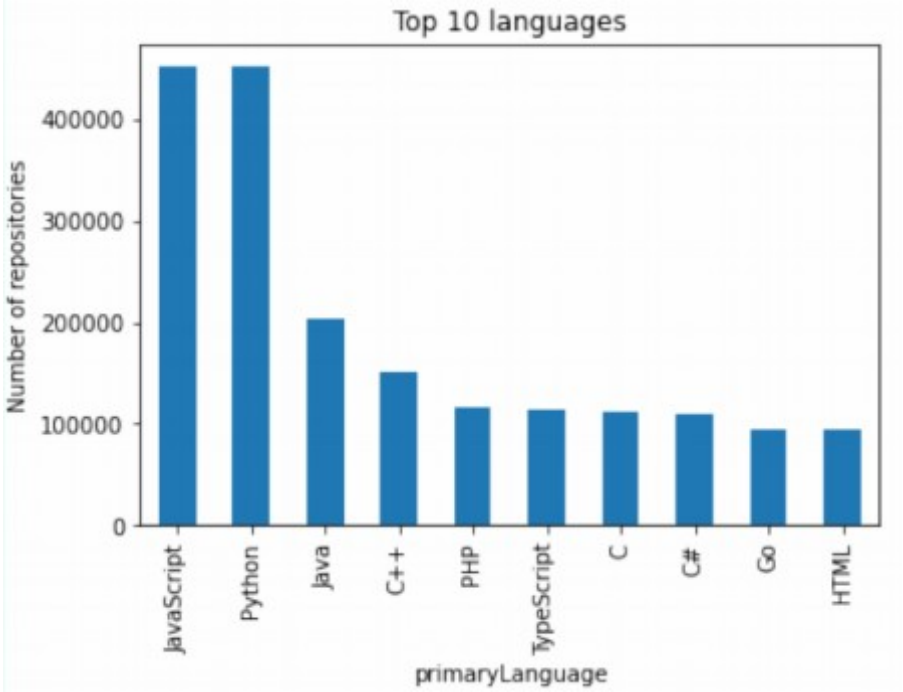


Рис. 2.3. Кількість репозитріїв програмного забезпечення за мовами на GitHub

Зауважимо, що TypeScript є синтаксично дуже схожим надмножиною JavaScript, тому для цілей цього аналізу ми розглядатимемо дві мови разом. Крім того, ми вибрали Go замість PHP або C, яка випередила їх за популярністю, оскільки Go використовує стандартизований формат документації з godoc.

2.1.2. Визначення реального варіанту для дослідження

Друга половина постановки проблеми стосується реального аспекту дослідження, де ми визначаємо реальний варіант використання як розробку з відкритим кодом на GitHub. Ця сфера охоплює два різні контексти.

- Оцінка існуючих прикладів: наявна розробка з відкритим кодом містить багату колекцію прикладів узгодженості коментарів експертів у формі минулого перегляду запитів на витяг. Іншими словами, ми можемо

отримати попередні приклади виправлених неузгоджених коментарів у результаті експертного огляду та перевірити, чи створюють наші методи інструмент, який би також правильно ідентифікував ту саму помилку.

- Людська реакція на нові внески: новий інструмент розробника корисний, лише якщо люди-розробники бажають його використовувати. Тому частиною відповіді на постановку проблеми є з'ясування того, як люди реагують на використання такого інструменту. У цьому випадку ми моделюємо використання інструменту в реальному світі у світі з відкритим вихідним кодом за допомогою запитів на отримання.

Розглядаючи як існуючі, так і нові приклади в контексті відкритих вихідних кодів GitHub, це дослідження має на меті оцінити здатність запропонованого підходу виявляти суперечливі коментарі в реальних проектах.

2.2. Огляд схожих досліджень в даній області

У систематичному огляді літератури, проведеному в [30], виконано огляд досліджень у цій галузі з 2011 по 2020 рік. З 47 вивчених статей автори визначили кілька тенденцій і прогалин, на яких можна зосередити майбутню роботу. Наприклад, 87% існуючих робіт вивчали мову програмування Java. Справді, усі попередні роботи, які найбільше стосуються сеттингу в цьому проекті, були дослідженнями, проведеними на мові Java. В екосистемі багатомовного програмного забезпечення ця єдина зосередженість на одній мові може не мати важливого контексту в інших мовах або призвести до методів оцінювання, які працюють лише на Java.

На рисунку 2.4 зображено найпопулярніші категорії проектів для кожної мови з хмарами слів із самостійно вибраних тем кожного проекту на GitHub, створених на основі аналізу метаданих понад 3 мільйонів сховищ [6]. Хмара слів (або хмара тегів) – це візуальне представлення тексту, яке дозволяє швидко оцінити його зміст. Чим частіше слово зустрічається в

тексті, тим більшим воно буде на хмарі. Крім того, кожна мова має свою власну нішу в екосистема з відкритим кодом із дуже невеликим перекриттям. Це означає, що дослідження, проведені на одній мові, де більшість проектів зосереджено на одному домені, можуть не поширюватися на іншу мову з іншим ринком. Наприклад, ми бачимо, що Java зосереджена на Android і базах даних. Python орієнтований на машинне навчання. Блокчейни та хмарні сервіси дуже популярні в Go, а в JavaScript домінують React і Node.js.

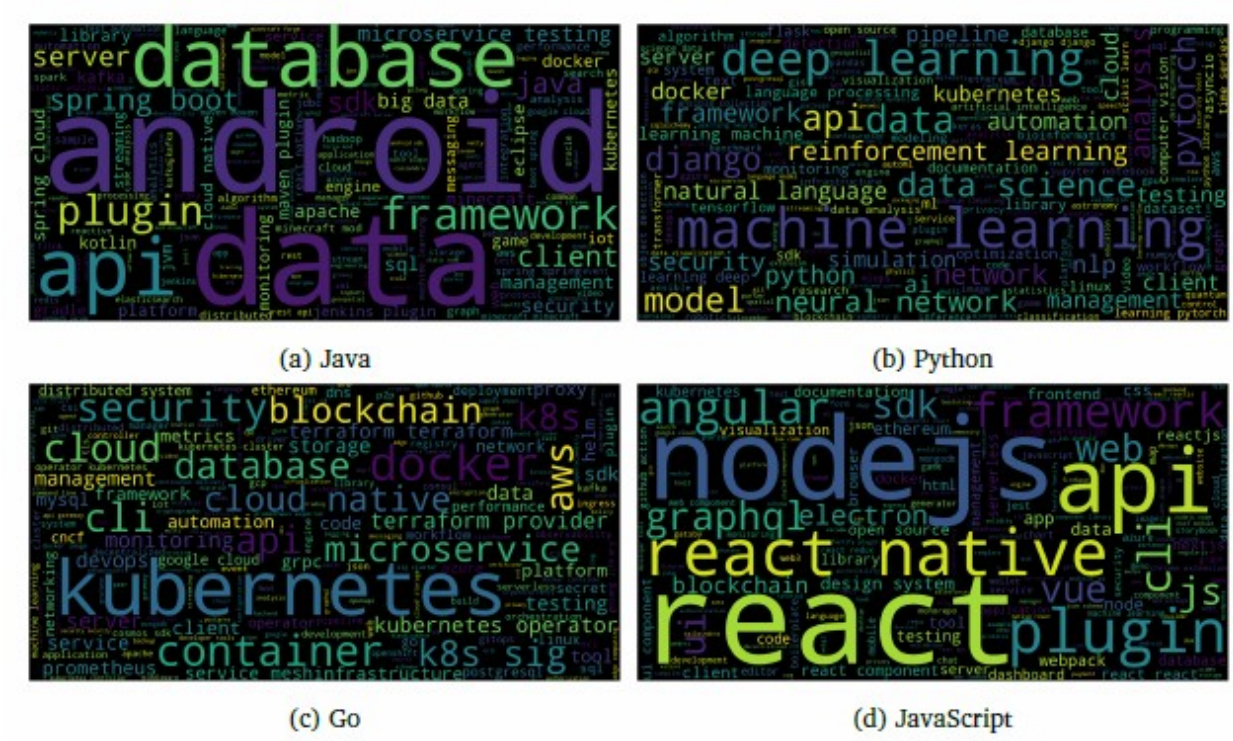


Рис. 2.4. Хмари слів за мовами програмування

Оскільки попередні роботи збирали лише набори даних для Java, вивчення інших мов вимагатиме збору нових наборів даних за допомогою методів інтелектуального аналізу програмного забезпечення.

Попередня робота з вивчення коментарів до коду найчастіше використовувала техніку ручної оцінки для оцінки якості [30]. Хоча повністю ручний підхід має переваги чіткої інтерпретації та високої точності, ця робота вимагає значних зусиль для отримання результатів. Це велике робоче навантаження спричинило брак існуючих інструментів, які могли б

допомогти розробникам у написанні послідовних коментарів і коду. В інших областях, пов’язаних з обробкою природної мови, підходи, засновані на глибокому навчанні, привели до результатів, які перевершують попередні методи, наприклад, у категоризації новин і аналізі настроїв. Зовсім недавно дослідники почали застосовувати мовні моделі для завдання узгодженості коментарів.

У порівнянні з великою історією традиційних методів оцінки коментарів, використання методів глибокого навчання для цього завдання має великий простір для дослідження. Щоб вирішити нашу постановку проблеми, ми виходимо за рамки попередніх підходів, включаючи використання великих мовних моделей, які були спеціально навчені на коді.

Дослідження [30], також виявило недолік в оцінювальній роботі як прогалину в поточній галузі дослідження. Минулі роботи, які пропонували показники якості коментарів, не оцінювали їх у репрезентативному середовищі, або оцінювали їх лише для доступних студентів університету з інформатики. У цьому дослідженні головна мета вивчення полягає в тому, щоб заповнити цю прогалину в поточній літературі на додаток до вирішення реального компоненту постановки проблеми. Наші цілі полягають у тому, щоб відповісти на репрезентативні приклади, наскільки ефективні поточні методи, і як практики реагують на запропоновані внески узгодженості коментарів.

Пов’язано з попереднім пунктом про практичне застосування, ми виявили, що в попередніх роботах не було глибоко досліджено відповіді розробників на системи перевірки узгодженості коментарів. Хоча ми можемо оцінити такі показники, як точність і точність, на прикладах із реальних проектів, ці цифри не висвітлять, чи справді розробники вважають внески корисними. Тому ми включаємо питання щодо того як розробники реагують на результати цієї моделі, щоб дослідити реакцію розробників на внесені виправлення узгодженості коментарів у налаштуваннях перевірки коду.

2.3. Дослідження та аналіз процесів узгодженості коментарів коду

2.3.1. Визначення узгодженості коментарів коду

Необмежена гнучкість коментарів створює величезну площу для вивчення якості коментарів коду. Попередня робота була зосереджена на звуженні масштабу проблеми до конкретних мов програмування або типів коментарів. Наприклад, хоча вбудовані коментарі часто описують щось про поведінку коду, вони також можуть відзначати поведінку, яка ще не реалізована. Деякі мови, такі як Java, накладають структуру на певні типи коментарів, наприклад коментарі на рівні методів у стилі Javadoc. Аналіз коментарів може використовувати знання цієї структури для оцінки якості коментарів, наприклад, перевіряючи, чи ім’я змінної @param відповідає значенню, присутньому в коді.

Сфера оцінки якості коментарів за своєю суттю є суб’єктивною, і існує багато потенційних визначень узгодженості [30]. Окрім узгодженості, дослідження зосереджені на перекриваючих якостях, включаючи релевантність, точність, повноту та точність. Розмитість термінології та збіги між досліджуваними аспектами коду та коментарями історично ускладнювали порівняння результатів різних робіт у сфері узгодженості коментарів коду. Для ілюстрації ми ділимося підбіркою прикладів тісно пов’язаних визначень якості нижче , як зазначили в [42]. Мета цього короткого списку полягає в тому, щоб вказати, що дослідження узгодженості коментарів коду не зупинилися на єдиному визначенні того, що є хорошим або поганим коментарем. Навпаки, це все ще дуже суб’єктивно.

• Правильність: чи правильна інформація в коментарі.

• Точність: Точність або точність коментаря. Наприклад, він не повинен бути занадто абстрактним або розпливчастим.

• Релевантність: наскільки коментар відповідає певній меті.

• Корисність: ступінь, до якої коментар може бути використаний його читачами для досягнення мети.

У цьому проекті ми визначили послідовність у визначенні 2.1 і 2.2. Це визначення поділяється кількома минулими роботами в цій галузі, що дозволяє пряме порівняння результатів, включаючи [20] і [28]. Необхідність збору великих обсягів даних для навчання великих моделей означає, що ми не можемо реально переглянути кожен окремий приклад коду та коментарів, щоб вирішити, чи вони послідовні чи ні. Визначення 2.2 разом із вибірковою попередньою обробкою та ретельним розглядом того, який проект із відкритим вихідним кодом слід включити, надає евристику для визначення того, чи є коментар послідовним.

2.3.2. Вимірювання якості коментарів

У попередньому розділі ми обговорили передумови визначення узгодженості. Ми поділилися прикладами того, як різні минулі роботи визначали правила якості, які вони вивчали. У цьому розділі ми досліджуємо вимірювання якості коментарів. Зауважте, що методи вимірювання можуть не узгоджуватися з визначенням якості. Як правило, визначення якості є суб’єктивним твердженням (коментар корисний), але вимірювання є об’єктивним (у коментарі 0 неправильно написаних слів і 1 помилка форматування).

Евристичні підходи підкреслили важливість визначення релевантних метрик для оцінювання [30]. У попередніх роботах використовувалися різноманітні об’єктивні метрики для вимірювання якості суб’єктивних коментарів. Наприклад, для вимірювання якості зрозумілості було використано оцінку читабельності Флеша-Кінкейда [32]. В іншій статті використовуються пропорції перекриття слів між кодом і коментарем як мірою суб’єктивної якості релевантності [33].

Однак ці показники можуть не використовуватися як показник якості на практиці. Великий збіг між кодом і коментарем може означати, що коментар зайвий. Показник читабельності, який визначається довжиною слів і речень, може бути безглуздим, якщо коментар також містить код.

У найближчому порівнянні з роботою, викладеною в цій було застосовано моделі глибокого навчання до цієї проблеми узгодженості коментарів. У глибокому навчанні виміряна якість коментаря більше не відповідає легко поясненому правилу. Достовірність такого вимірювання залежить від того, наскільки добре воно виконується, і застосовності обставин, за яких модель навчалася.

У їхній роботі проблему узгодженості коментарів на рівні методу JavaDoc вивчали шляхом побудови збалансованого набору даних із 40 688 прикладів, відібраних із 1518 проектів Java з відкритим кодом. Автори побудували набір моделей на основі трансформаторів, які кодували текст коду або з моделлю послідовного тексту, або з графовою нейронною мережею (GNN), яка включала інформацію AST. Огляд цієї архітектури зображено на рисунку 2.5.

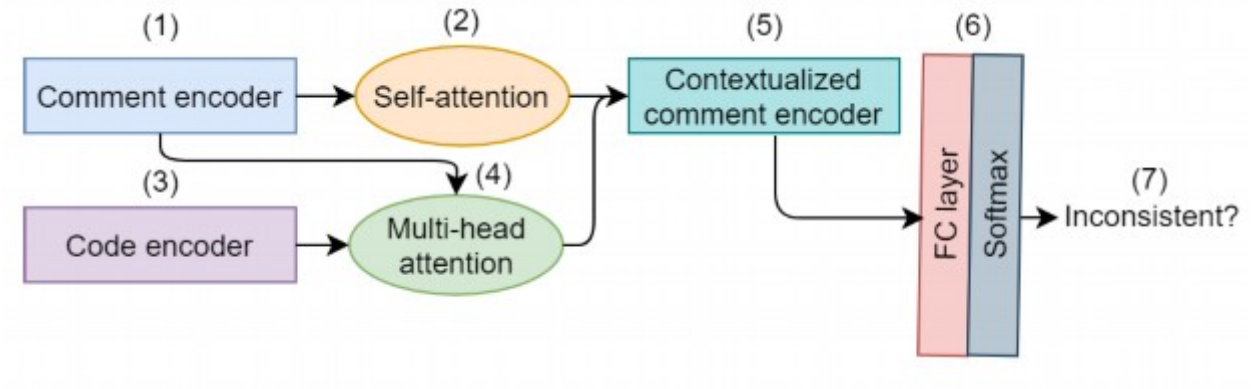


Рис. 2.5. Архітектура моделі з використанням нейронної мережі

Автори оцінили свої методи за двох обставин, визначених як post-hoc і just-in-time. У post-hoc налаштуванні модель отримує лише остаточний коментар і пару коду та має визначити, чи є вони узгодженими. Цей параметр відповідає визначенню проблеми нашої роботи. У налаштуваннях «just-in-time» вхідні дані для моделі також містять попереднє значення коментаря та коду. Цей параметр відповідає випадку використання виявлення невідповідностей під час фіксації або під час редагування. У повному наборі тестових даних найкраща модель (на базі GNN) отримала оцінку F1 0,672 у

пост-гоціальних налаштуваннях і 0,809 у своєчасних налаштуваннях. Однак ці результати включають оцінку за структурованою @param і ©return розділи формату Javadoc. Як обговорювалося вище, наше налаштування вивчає лише неструктуровану частину коментаря, яку не дуже легко проаналізувати за допомогою стандартних засобів розробки. Тому, розглядаючи лише підсумковий рядок коментаря JavaDoc, найкраща модель мала оцінку F1 0,706. Автори повідомили, що для навчання цієї моделі на графічному процесорі Nvidia Titan знадобилося 30 хвилин.

Наступною частиною пов'язаної роботи, яку ми розглядаємо, є покращення цих результатів за допомогою BERT і Longformer. Використовуючи той самий набір даних і метрики оцінки, що в [39] точно налаштували попередньо підготовлену базову модель BERT для цього завдання. У post-hoc налаштуваннях автори досягли найкращого результату F1 0,72. Навчання цієї моделі відбувалося розподіленим способом на 8 графічних процесорах Nvidia RTX A5000 (по 24 ГБ кожен) протягом невизначеного періоду часу.

Окрім побудови нових моделей на основі LLM, наша робота повторює ці моделі для порівняння на нових наборах даних.

2.3.3. Генерація коментарів

Генерація коментарів означає автоматичне створення коментарів для існуючого коду, які допоможуть програмістам зрозуміти, повторно використовувати та підтримувати програмне забезпечення. Хоча наш проект конкретно не націлений на результат генерування коментарів, все ж важливо зрозуміти останній досвід цього завдання. Це пояснюється тим, що можна сформулювати послідовність коментарів як проблему генерації, заявивши, що коментар є послідовним, якщо автоматично згенерований коментар достатньо схожий на оригінал.

У нещодавньому дослідженні технології створення коментарів [35] визначено кілька підходів, подібних до методів виявлення узгодженості

коментарів. Наприклад, методи клонування коду шукають семантично подібний код у кодовій базі та використовують наявні коментарі, щоб заповнити прогалини у відсутніх функціях. Підходи глибокої нейронної мережі навчаються генерувати коментарі шляхом навчання на наборах даних пар коду та коментаря. Ця галузь стикається з подібними проблемами щодо суб'єктивності критеріїв оцінювання та труднощів вимірювання практичного впливу [35]. Хоча робота над створенням коментарів триває, нещодавнє опитування в [36] продемонстрували, що коментарі, написані людиною, були більш корисними, ніж коментарі, згенеровані машиною, у завданнях розуміння програми.

У пропонованому проекті ми вирішили зосередитись на виявленні узгодженості коментарів, а не на генеруванні через головну мету як інструмент для вдосконалення перевірки коду, коли код уже написаний і його потрібно перевіряти на правильність. Ще одна причина, чому ми додаємо коротку довідкову інформацію про генерацію коментарів, полягає в тому, що набори даних, які використовуються для вивчення однієї проблеми, також можуть бути використані для вивчення іншої. В обох випадках набори даних складаються з пар коментарів і коду. Наприклад, набори даних, створені в цій роботі, також можна використовувати для навчання моделей генерації коментарів для мов поза Java.

2.4. Представлення мовних моделей коду

Оскільки комп'ютерний код — це просто текст, мовні моделі є ідеальним кандидатом для виконання завдань на основі вихідного коду. У цьому випадку під великою мовною моделлю (LLM) розуміють глибоку нейронну мережу, навчену виконанню завдань, що включають розуміння природної мови та щонайменше кілька сотень мільйонів параметрів.

Велика мовна модель (LLM, Large Language Model) — це тип штучної нейронної мережі, яка тренується на великому обсязі текстових даних і

здатна розуміти, генерувати та виконувати завдання, пов’язані з природною мовою. Основою таких моделей є технологія глибокого навчання (deep learning), яка дозволяє моделі обробляти та аналізувати складні мовні структури.

Наведемо декілька ключових особливостей великих мовних моделей:

- Масштаб: В LLM використовується величезна кількість параметрів (мільярди або навіть трильйони), що дозволяє їм краще "розуміти" контексти, зв'язки між словами та семантику.

- Навчання на великих даних: Моделі тренуються на масивних текстових корпусах, що включають книги, статті, веб-сайти та інші джерела тексту.

- Обробка природної мови: LLM здатні виконувати різноманітні завдання, такі як генерація тексту, переклад, відповіді на питання, підсумовування текстів, аналіз емоцій тощо.

- Контекстуальне розуміння: Завдяки контекстуальним зв'язкам, великі мовні моделі можуть враховувати зміст цілих абзаців або документів, що покращує якість відповідей або генерованого тексту.

Популярними прикладами LLM є моделі на кшталт GPT (Generative Pre-trained Transformer) від OpenAI, BERT (Bidirectional Encoder Representations from Transformers) від Google, та інші, що використовуються в пошукових системах, чат-ботах, перекладах та багатьох інших додатках.

Переваги LLM:

- Потужність у розумінні складних запитів.

- Можливість автоматизації рутинних завдань на основі тексту.

- Використання в різних галузях: від науки до бізнесу.

2.4.1. Мовні моделі

Загалом мовні моделі є статистичними представленнями мовних відносин. У той час як числові мовні моделі, такі як N-грамні ланцюги Маркова, використовувалися протягом багатьох десятиліть для генерації

тексту [21], останні досягнення в обчислювальній техніці сприяли неймовірному поширенню широкомасштабних текстових моделей на основі трансформаторів, навчених на величезній кількості наборів даних з Інтернету, таких як BERT [5] і сімейство GPT [29].

У сукупності, хоча точне значення великого є нечітким, ці нові моделі відомі як великі мовні моделі, оскільки вони складаються з мільйонів або мільярдів числових параметрів. Ці моделі вивчають мовні зв'язки, перетворюючи слова на числа (крок, відомий як кодування) і оптимізуючи свою здатність передбачати лексеми з великого корпусу тексту за допомогою градієнтного спуску. Оскільки текст також є представленням вихідного коду мови програмування, природним наслідком є використання цих моделей у програмному коді. Такі підходи, як OpenAI Codex [3], уже широко використовуються як основа для інструменту автозаповнення Copilot.

2.4.2. Архітектура мовних моделей

Двома яскравими прикладами великих мовних моделей, які представлені в цьому проекті, є BERT і GPT. Обидва засновані на трансформаторній архітектурі та були навчені передбачати токени у великих корпусах тексту з Інтернету (мільярди слів). Однією з ключових переваг, яка забезпечує масштаб цих моделей, є те, що вони самоконтрольовані, тобто людині не потрібно позначати правильний прогноз під час навчання. Замість цього вони оцінюються щодо передбачених токенів, які вже існують у тексті набору даних, але приховані від вхідних даних моделі.

Розглянемо деталі навчання. Цей проект застосовує одну модель на основі BERT і одну на основі CodeGen для вивчення проблеми узгодженості коментарів. Хоча обидві моделі засновані на трансформаторах і залежності вхідних даних моделі через механізм уваги [43], BERT є кодувальником, а CodeGen (як і GPT) є декодером. Це означає, що результатом BERT є один вектор вбудовування на вхідний маркер, тому вхід і вихід мають однакову довжину. Навпаки, вихід CodeGen є єдиним вектором, що представляє

розподіл ймовірностей для передбачення наступного слова, наступного за даним введенням.

По-перше, ми передамо деякі відповідні деталі навчання та впровадження BERT (на якому базується CodeBERT). Перший вхідний маркер для BERT — це маркер спеціального класу [CLS], за яким слідує послідовність введення, токенізована за допомогою WordPiece, і розмір словника становить 30 000. Це представлення показано на рисунку 2.6.

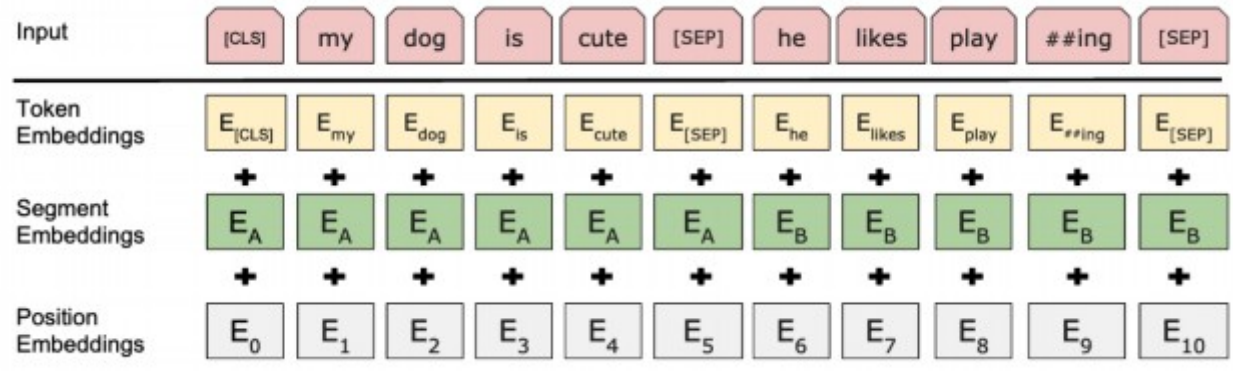


Рис. 2.6. Вхідне представлення BERT

Модель навчається за допомогою завдання моделювання замаскованої мови (MLM) на наборі даних, що складається з BookCorpus та англійської Вікіпедії. У цьому завданні 15% вхідних маркерів приховано спеціальним маркером [MASK], а мета моделі — передбачити приховані маркери. Зауважте, що на цьому етапі до моделі додається один проекційний шар, який створює розподіл ймовірностей розміром словника.

2.4.3. Спеціальне навчання коду

Хоча набори даних, які використовувалися для навчання цих моделей, містили зразки коду, для спеціалізованого використання в завданнях, пов’язаних з кодом, додаткове навчання дає кращу продуктивність. Таким чином, CodeBERT [9] і Codex [3] виникли з BERT і GPT, відповідно, шляхом точного налаштування наборів даних мови програмування.

CodeBERT базується на популярній моделі BERT, яка вже досягла значних успіхів в області NLP. Однак, на відміну від BERT, CodeBERT навчається на величезній кількості пар "код-коментар", що дозволяє їй встановлювати зв'язки між фрагментами коду та відповідними поясненнями.

Ключові можливості CodeBERT:

- Генерація коментарів. Модель може автоматично генерувати зрозумілі коментарі до фрагментів коду, що значно полегшує розуміння та підтримку програмного коду.

- Пошук коду за природною мовою. Ви можете задати запит природною мовою, і CodeBERT знайде відповідні фрагменти коду з великих репозиторіїв.

- Переклад коду. Модель може перекладати код з однієї мови програмування на іншу, зберігаючи при цьому його семантику.

- Автозаповнення коду. CodeBERT може передбачати наступний фрагмент коду, що ви пишете, прискорюючи процес розробки.

CodeBERT використовує ту саму загальну архітектуру, що й база BERT, із 110 мільйонами параметрів, які можна навчати. Однак він використовує кодування пари байтів як токенізер замість WordPiece. Що стосується роботи в цьому проекті, CodeBERT також використовує набір даних із загальнодоступних репозиторіїв GitHub на шести мовах програмування: Java, Python, Go, JavaScript, PHP і Ruby. Його завданням попереднього навчання є бімодальний MLM, де природна мова та відповідний введений код об’єднуються за допомогою спеціального маркера [SEP] і подаються разом у модель. Ці вхідні дані отримані з функцій і коментарів до їх документації. Дійсно, етап попереднього навчання в CodeBERT майже ідентично нагадує вхідний набір даних нашого проекту. Таким чином, ми можемо очікувати, що його можна дуже ефективно адаптувати до завдання виявлення невідповідності.

Друга модель, яку ми досліджуватимемо, CodeGen [26], є загальнодоступним рішенням Codex, яке недоступне для тонкого

налаштування через ексклюзивну угоду між OpenAI і Microsoft. Проект CodeGen публікує кілька розмірів моделей, з яких ми плануємо використовувати лише два найменші для цілей оцінки (350 мільйонів і 2 мільярди параметрів). Вхідними даними для цієї моделі є кодована пара байтів токенізований рядок, а її виходом є розподіл ймовірностей у словнику для наступного токена. На етапі навчання метою моделі є прогнозування наступного токена кожного вхідного зразка. Ця багатомовна модель послідовно навчається на двох наборах даних. Спочатку він навчається на ThePile, наборі даних із 354 мільярдів токенів, що складається з агломерації багатьох менших наборів даних [11]. Потім він навчається на BigQuery, наборі даних, що складається з відкритого коду C, C++, Java, JavaScript, Python, і код Go.

Зауважте, що перетином загальних мов між CodeGen і CodeBERT є саме ті мови, які ми пропонуємо вивчати в цьому проекті.

2.4.4. Точне налаштування за допомогою позначених наборів даних

Для інших програм, окрім передбачення токенів, ми застосовуємо точне налаштування за допомогою позначених наборів даних щодо попередньо навченої моделі. Цей процес створює модель, яка виконує наступні завдання, такі як відповіді на запитання, аналіз настроїв або, у випадку цього проекту, виявлення узгодженості коду та коментарів. Інтуїція, чому ця процедура може працювати, полягає в тому, що завдяки попередньому навчанню модель уже навчилася встановлювати відповідні зв’язки (ваги) між своїми вхідними даними, що дає їй розуміння словникового запасу та граматики базової мови. Таким чином, модель потребує лише відносно невеликої кількості додаткового нагляду, який буде застосований у пов’язаних нижчих налаштуваннях. Оцінки цих точно налаштованих моделей у задачах, подібних до виявлення коментарів, таких як пошук коду, показали схожу продуктивність зі спеціалізованими підходами, розробленими спеціально для цього завдання [9].

2.5. Збір даних програмного забезпечення з відкритих сховищ

Цей розділ охоплює літературу, пов’язану з кодом майнінгу з відкритих сховищ. Хоча це не є основним напрямком цього проекту, розуміння попередньої роботи зі збору даних програмного забезпечення для дослідження є критичною залежністю через відсутність наявних даних для виявлення узгодженості коментарів кількома мовами.

2.5.1. Практики майнінгу GitHub

GitHub — це де-факто онлайн-сховище великої кількості вихідного коду програмного забезпечення, яке можна безкоштовно використовувати для проектів з відкритим кодом. Однак із цією доступністю постає ряд проблем для дослідника, який вивчає практику розробки програмного забезпечення. У обіцянках і ризиках майнінгу GitHub, в [16] представики набір міркувань щодо вибору проектів програмного забезпечення з відкритим кодом для вивчення.

Оскільки метою цього проекту є створення моделі, яка може бути застосована до професійних додатків, слід уважно відфільтрувати набір розглянутих сховищ. Простий вибір найпопулярніших проектів може не призвести до репрезентативного набору даних. Наприклад, найпопулярніший проект на GitHub, freeCodeCamp/freeCodeCamp, є онлайн-курсом програмування. Але, здавалося б, величезна популярність цього проекту зумовлена самою навчальною програмою, де одним із перших завдань є розміщення цього репозиторію на GitHub.

Щоб виконати компонент збору даних цього проекту, нам також потрібно розуміти відповідні методи дослідження програмної інженерії. У [16] описується багато функцій, про які слід знати під час видобутку даних з GitHub. У статті автори пояснюють, що природа GitHub як безкоштовної платформи для будь-якого програмного забезпечення з відкритим кодом призводить до багатьох репозиторіїв, які є неактивними, мають особистий

характер або взагалі не містять програмного забезпечення. Деякі з ключових висновків висвітлено в таблиці 2.1.

Таблиця 2.1.

Висновки щодо аналізу GitHub

Peril Category	Percentage (%)
Inactive projects	46
Non-software development	36
Personal projects	72
Less than 25 total pull requests	95

2.5.2. Емпіричні підходи до оцінки коду

Мета створення набору еталонних показників також заслуговує на певну увагу в дослідженні. У великому емпіричному дослідженні [44] уна основі 500 комітів створено таксономію змін і визначено, які типи комітів найімовірніше призведуть до неузгодженості між кодом і коментарем. Зокрема, автори видобули приклади потенційних проблем, шукаючи повідомлення комітів, що містять ключові слова update або outdate і comment. Цей метод призвів до 27,6% помилкових позитивних результатів серед 500 комітів, проаналізованих вручну, де шаблон пропонував відповідний коментар, але перевірка показала, що він не пов’язаний.

Разом із моделлю [3] представив тест HumanEval, збірку 164 оригінальних програмних задач у стилі запитань для співбесіди розробника програмного забезпечення. У цьому контрольному тесті моделі оцінюються на основі їхньої здатності створювати код із тим самим результатом, що й створене людиною рішення під час виконання.

На додаток до своєї моделі в [26] створено еталонний тест, відомий як Multi-Turn Programming Benchmark (MTPB). Цей еталонний тест схожий на HumanEval тим, що він також оцінює моделі шляхом порівняння виходу згенерованого ними коду з постановкою проблеми. Однак ці проблеми подано покроково, де надаються підказки для кількох кроків загальної

проблеми. Цей тест містить 115 задач. На рисунку 2.7 наведено приклад однієї задачі з Multi-Turn Programming Benchmark (MTPB).

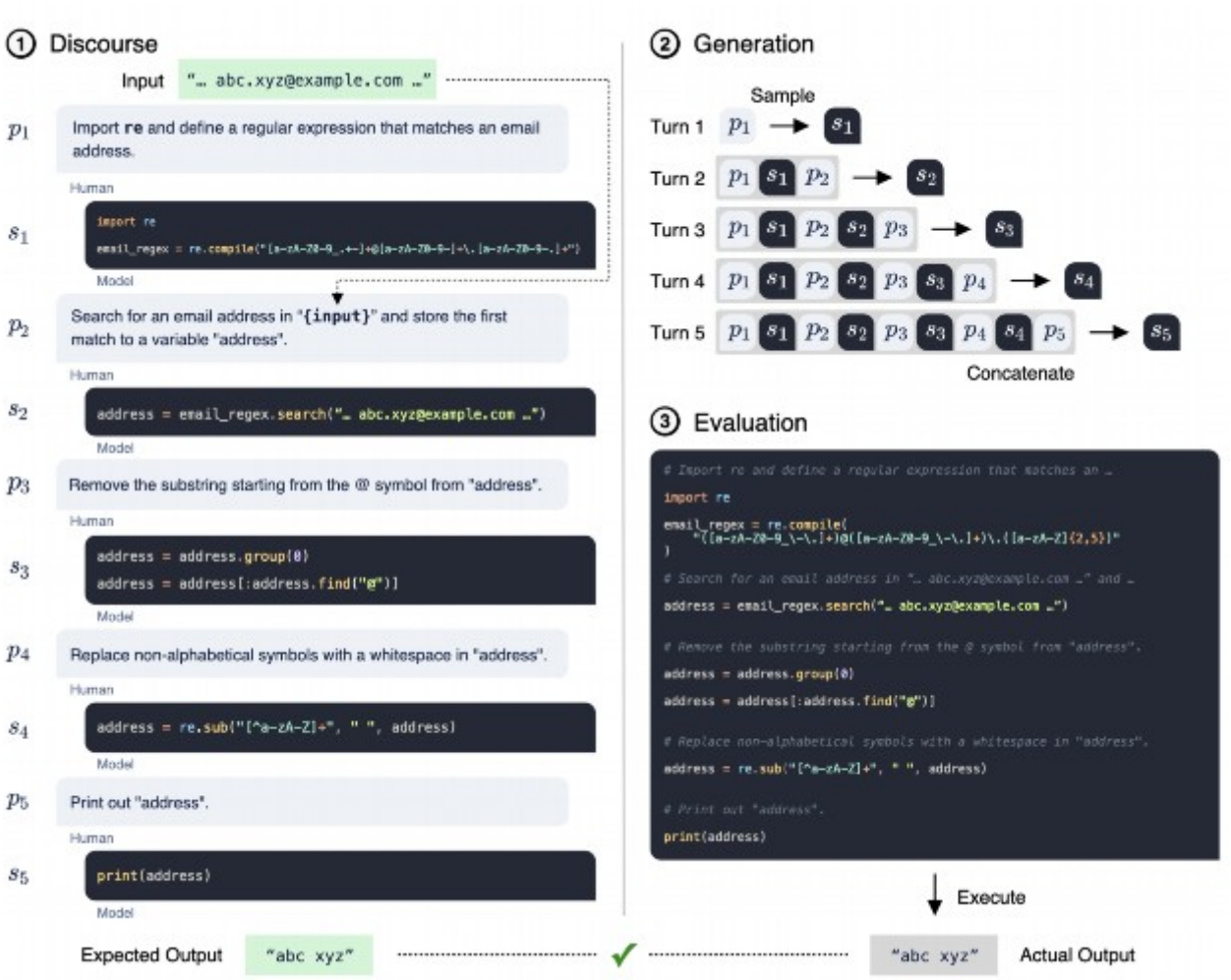


Рис. 2.7. Приклад проблеми з MTPB

Одним із очікуваних результатів дослідження в цьому проекті є розуміння того, як розробники реагують на внески, які виправляють невідповідності коментарів. Хоча цей конкретний тип внеску не вивчався в минулому, ми можемо переглянути інші роботи, які подали виправлення для проектів з відкритим кодом на GitHub. У контексті нашої роботи мета перегляду попередніх досліджень щодо внесків запитів на витягування полягає в тому, щоб повідомити очікування щодо ймовірних рівнів прийняття та оцінити адекватний розмір вибірки.

В [22] розроблено інструмент для виправлення помилок на основі попереджень, виданих популярними інструментами статичного аналізу коду, такими як FindBugs і SonarQube. Автори надіслали 38 запитів на отримання відкритих Java-проектів, включаючи Eclipse IDE та інструменти FindBugs, і виявили, що 84% запропонованих виправлень були прийняті та об'єднані. У подібній роботі [15] видобули історію версій проекту на GitHub, щоб створити REVISAR, інструмент, який визначає шаблони редагування, які часто використовуються. Автори надіслали 16 пул-запитів за допомогою інструменту, з яких 6 було прийнято. Тан та ін. пілотував інноваційний підхід, інтегрувавши процес виправлення помилок у відкритому коді в академічну навчальну програму. Залучаючи внески великого класу з 154 студентів, автори досягли більшого масштабу запитів на вилучення, ніж інші роботи. Внески були спрямовані на вирішення існуючих відкритих проблем у вибраних проектах з відкритим кодом.

Висновки до розділу

У цьому розділі було розглянуто ключові аспекти використання мовних моделей для вирішення задач класифікації узгодженості програмного коду та його коментарів. Аналіз процесу документування коду вказує на важливість ретельного підходу до вибору мови програмування, а також на необхідність орієнтації на реальні сценарії використання в процесі розробки.

Огляд наявних досліджень в цій області підкреслює, що узгодженість коментарів з кодом є критичним фактором для забезпечення якості програмного забезпечення. Особливу увагу приділено методам вимірювання якості коментарів, що дозволяє оцінити їх релевантність та точність. Генерація коментарів за допомогою мовних моделей є перспективним напрямом для автоматизації цього процесу, зокрема для підвищення продуктивності розробників.

У розділі детально представлено мовні моделі, які використовуються для аналізу та генерації коду, з акцентом на їх архітектуру та специфіку навчання. Важливим аспектом є застосування спеціальних наборів даних для точного налаштування моделей, що дозволяє досягти високої точності у розв'язанні завдань узгодженості.

Окремо було висвітлено процес збору даних з відкритих сховищ програмного забезпечення, таких як GitHub, що є основою для емпіричних досліджень та тестування моделей на реальних прикладах коду. Практики майнінгу відкритих репозиторіїв дозволяють отримувати великі обсяги даних, необхідні для навчання мовних моделей, що робить цей підхід ефективним для автоматизації аналізу узгодженості коментарів.

Загалом, розділ демонструє ефективність використання мовних моделей для розв'язання проблем узгодженості коду, що відкриває нові можливості для автоматизації процесів рецензування та покращення якості програмного забезпечення.

РОЗДІЛ 3. МЕТОДОЛОГІЯ КЛАСИФІКАЦІЇ УЗГОДЖЕНОСТІ ТА ПОСЛІДОВНОСТІ КОДУ ЗАСОБАМИ МОВНИХ МОДЕЛЕЙ

3.1. Застосування мовних моделей для класифікації узгодженості коментарів

Пропонований підхід надає можливість застосувати великі мовні моделі (LLM) для класифікації узгодженості коментарів і передбачає навчання раніше введених великих мовних моделей на існуючому наборі даних прикладів Java [28]. Ми обираємо цей підхід з двох причин. По-перше, використання наявного набору даних пропонує пряме порівняння з існуючим сучасним рівнем техніки. По-друге, ми стверджуємо, що цей підхід дійсний, оскільки цей набір даних базується на видобутих проектах з відкритим кодом, які представляють код, який можна зустріти на практиці. Фактично, в [10] виявили, що 90% змін коду також правильно оновлювали коментарі, що свідчить про те, що приклади в наборі даних, швидше за все, є надійними на основі способу збору.

3.1.1. Процес вибору коментаря

Оскільки коментарі не відповідають синтаксичній структурі комп'ютерного коду, їх різноманітність практично безмежна. Таким чином, спроба створити модель, яка розуміє кожен окремий написаний коментар, швидко стане неспроможною. У цьому розділі ми описуємо різні типи коментарів, щоб звузити конкретний тип, який ми розглядаємо в цьому проекті. Лістинг 3.1 демонструє кожен тип коментарів, розглянутий у цьому розділі.

Вбудовані коментарі та коментарі рівня функцій.

У більшості мов програмування коментарі можуть бути написані будь-де в коді або пов'язані з цілими функціями чи методами. Перший тип ми називаємо вбудованими коментарями, а другий — коментарями функцій. У

Java, наприклад, цей другий тип включає коментарі у форматі Javadoc. Ми робимо це розрізнення, оскільки вбудовані коментарі створюють додаткову проблему визначення обсягу коментаря. Вбудовані коментарі часто надають короткі пояснення конкретних фрагментів коду, але можуть містити недостатньо інформації, щоб бути корисною для оцінки узгодженості. Деякі вбудовані коментарі можуть бути навіть закоментованими рядками коду і взагалі не містити природної мови.

Лістинг 3.1. Типи коментарів

```
def foo(arg1):
  '''Unstructured summary comment that explains the usage of the function (the target of
      this project).

  :param arg1: structured information about arg1
  :returns: structured return value explanation
  '''
  bar = arg1 * arg1
  # Inline comment explaining details of the code
  # TODO: comment that describes missing behavior
  return arg1 + arg1 + bar
```

У цьому дослідженні ми зосереджуємось на коментарях рівня функції, оскільки вони завжди знаходяться в одній позиції та пов’язані з усією функцією, навіть у різних проектах і мовах програмування. Це обмеження означає, що наші результати дійсні лише для коментарів на рівні функції. Вбудовані коментарі через їх ще менш обмежену структуру можуть бути складнішими для класифікації.

В [20] застосовано “випадкові ліси” з функціями, створеними вручну, щоб класифікувати змінені вбудовані коментарі як узгоджені чи непослідовні. Ці функції включали, наприклад, чи змінилося оголошення методу, а також кількість змінених операторів у коміті. Однак в [2] виявили, що зміна роздільної здатності внутрішнього коментаря мала значний вплив на результати. Оскільки ми також вивчаємо узгодженість коментарів у кількох мовах, перегляд коментарів функцій дає нам подібні налаштування для порівняння.

Неструктуровані та структуровані коментарі.

У цьому параграфі ми розрізняємо структуровані та неструктуровані частини коментарів функціонального рівня. Структурована частина коментаря стосується використання попередньо визначених анотацій документації, таких як @param у Java та :param у Python. Під неструктурованим ми обговорюємо в першу чергу резюме коментарів до функції, яке з'являється першим. Це резюме, як правило, є довільним описом використання або призначення функції, про яку йдеться, і не потребує певної структури. Це дослідження в першу чергу стосується неструктурованих коментарів, оскільки структуровані коментарі часто вже можна ефективно перевірити за допомогою евристичних підходів та інтегрованих середовищ розробки. Наприклад, такі інструменти, як Javad oc, можуть перевірити, чи збігаються тип, порядок і написання цих структурованих розділів між функцією та її коментарем.

Самопроголошений технічний борг (SATD - Self-admitted technical debt)

SATD (також відомий як коментарі TODO) вказує на завдання, які відсутні або незавершені, і може не надавати корисної інформації про сам код. На відміну від інших типів коментарів, що описують існуючу поведінку в коді, ці коментарі конкретно описують те, чого немає в коді. Тому такі коментарі виключені з дослідження, оскільки їхній критерій узгодженості полягає в тому, що вони описують функціональність, яка не існує, що є протилежним до всіх інших коментарів.

3.1.2 Набір даних

Початковий набір даних містить збалансовану вибірку з 40 688 прикладів пар коду та коментарів, що були позначені та вилучені з 1 518 проектів Java на GitHub [28]. Для нашої роботи, з огляду на фокус на неструктурованих коментарях, ми розглядаємо лише клас Summary у наборі даних, що становить 10 498 прикладів. Ми дотримуємося початкового розподілу 80-10-10 для отримання навчальних, валідаційних і тестових

наборів. Ці набори розподілені таким чином, щоб не було перетину між проектами; всі приклади з будь-якого проекту будуть лише в одному з цих наборів.

Використовуючи токенізатор CodeBERT, який базується на токенізаторі байтно-парного кодування, що використовується в RoBERTa [19], медіанна довжина прикладів з навчального набору становить 131, з валідаційного — 128, а з тестового — 123. Приблизно 13% всіх прикладів перевищують максимальну довжину токенів BERT у 512 (включаючи спеціальний токен [CLS]). Цей токенізатор має розмір словника 30 000 токенів [5].

Згідно з токенізатором Codegen, який має розмір словника 50 000 токенів [26], медіанна довжина відповідно становить 98, 97 і 93. Ми можемо пояснити цю різницю більшим розміром словника токенізатора Codegen, що дозволяє кожному токену включати більше символів у середньому.

Правила класифікації

Для класифікації прикладів у наборі даних автори використовували історії комітів проєктів. Якщо коміт змінює конкретний метод, змінюючи як код, так і його коментар, то старий коментар не є узгодженим з новим кодом. Якщо коміт змінює код без змін у коментарі, то старий коментар позначається як узгоджений з новим кодом.

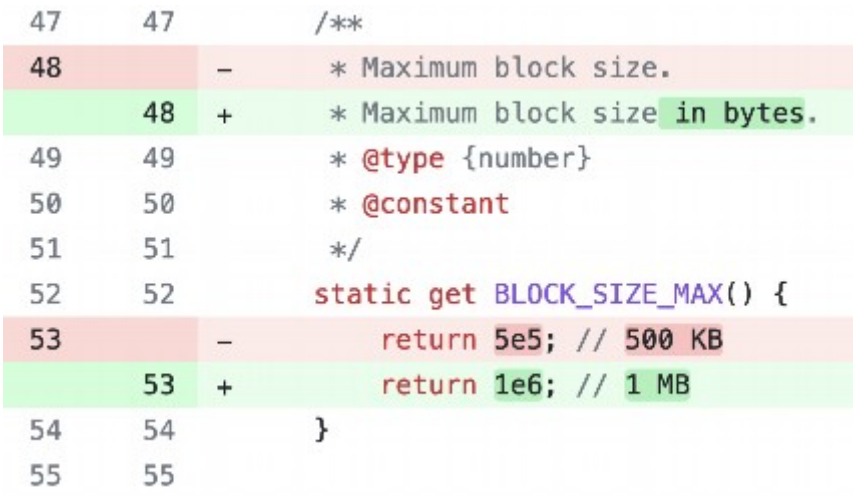


Рис. 3.1. Приклад коміту, який змінює і код, і коментар

На рисунку 3.1 показано приклад коміту, який змінює як код, так і коментар для функції . Модифікація додає текст у байтах до коментаря та змінює повернуте значення з 5e5 на 1e6. Згідно з нашою стратегією класифікації, коментар перед комітом буде позначений як несумісний з новим кодом.

Одним із результатів цієї стратегії маркування став значний дисбаланс в отриманому наборі даних. На практиці було значно частіше, коли код змінювався без зміни коментаря, ніж альтернатива. Таким чином, автори отримали набагато більше прикладів негативної мітки, ніж позитивної. Щоб вирішити цю проблему, було зменшено негативний випадок до кількості позитивних прикладів.

Post-hoc та Ad-hoc оцінка [28] представляє два класи виявлення узгодженості коментарів: Post-hoc і Ad-hoc (або just-in-time). У post-hoc налаштуваннях модель отримує лише остаточний коментар і пару коду та має визначити, чи вони узгоджені. У налаштуванні ad-hoc вхідні дані для моделі також включають попереднє значення коментаря та коду, які вважаються узгодженими, і модель повинна передбачити, чи нова пара все ще є узгодженою. Цей параметр відповідає випадку використання виявлення невідповідностей під час фіксації або під час редагування. У нашому дослідженні ми в основному розглядаємо post-hoc налаштування, оскільки модель, здатна виконувати це завдання, автоматично досягне успіху в ad-hoc налаштуваннях, просто ігноруючи набір змін коду та переглядаючи лише код після змін.

3.1.3. Вибір моделі

Сфера великих мовних моделей швидко розвивається, нові моделі та методи прогресують у сучасному стані прискореними темпами. У цій роботі ми зосереджуємося на двох попередньо підготовлених моделях мови, які представляють широкий спектр поточних досліджень.

• CodeBERT застосовує додаткове спеціальне навчання коду на основі оригінальної моделі мови BERT. Ця модель двонаправленого кодера показала високі результати в наступних завданнях, включаючи класифікацію коду, які тісно пов’язані з метою цього проекту.

• CodeGen — це модель із відкритим вихідним кодом, розроблена для того, щоб запропонувати альтернативу закритій моделі Codex OpenAI. Ця модель, заснована на трансформаторній архітектурі декодера (наприклад, GPT), була навчена на кількох гігабайтах коду та демонструє потужні можливості генерації коду.

CodeGen – це потужна модель машинного навчання, спеціально розроблена для генерації коду. Вона використовує методи глибокого навчання, щоб аналізувати величезні обсяги існуючого коду та навчатися на основі цього досвіду. Завдяки цьому CodeGen здатна автоматично генерувати нові фрагменти коду, що відповідають заданим вимогам.

Алгоритм роботи CodeGen наступний:

- Навчання на даних. Модель CodeGen навчається на великих датасетах, що містять мільйони рядків коду різних мов програмування. Це дозволяє їй засвоїти синтаксис, семантику та стилістику написання коду.

- Генерація коду. Після навчання модель може генерувати код на основі введеного промту або контексту. Наприклад, ви можете задати їй питання природною мовою, і вона спробує написати відповідний фрагмент коду.

- Підтримка різних мов програмування. CodeGen підтримує широкий спектр мов програмування, що робить її універсальним інструментом для розробників.

Щоб застосувати великі мовні моделі (LLM) для класифікації узгодженості коментарів, ми налаштовуємо моделі CodeBERT і CodeGen. Зокрема, для CodeBERT ми отримуємо остаточну класифікацію, прикріплюючи щільний шар із двома виходами до кінцевого вихідного вбудовування унікального токена [CLS], який було додано до вхідних даних.

Це слідує підходу, який автори оригіналу використовували для оцінки наступних завдань [9]. Ця модель представлена на рисунку 3.2.

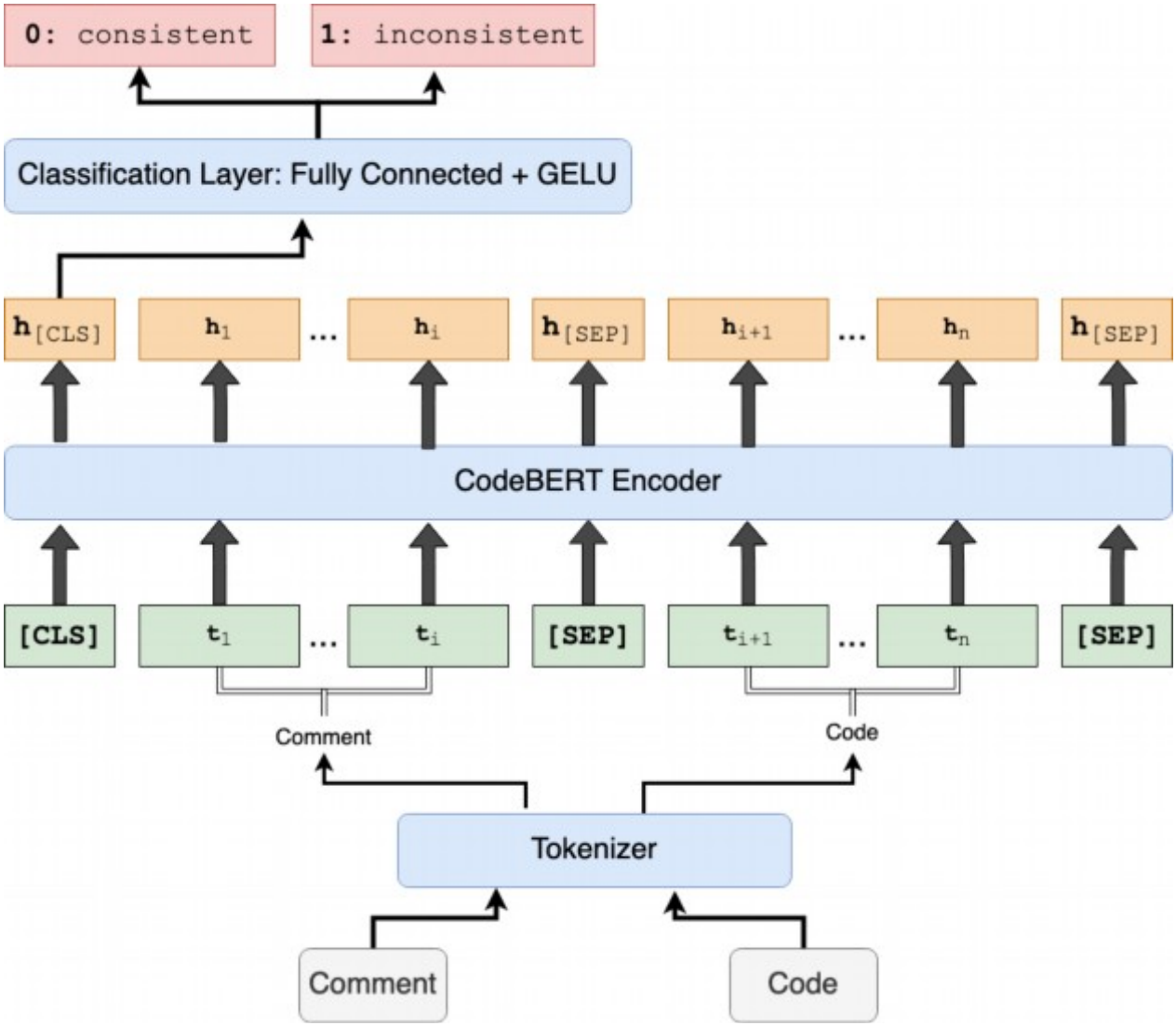


Рис. 3.2. Діаграма моделі CodeBERT

CodeGen, як модель декодера, містить кінцевий вихідний рівень, який створює розподіл ймовірностей по всьому простору маркерів. Для нашого підходу до тонкого налаштування ми замінюємо цей шар бінарним вихідним щільним шаром, що відповідає нашому простору міток двох класів. Ми включаємо діаграму цієї моделі на рисунку 3.3.

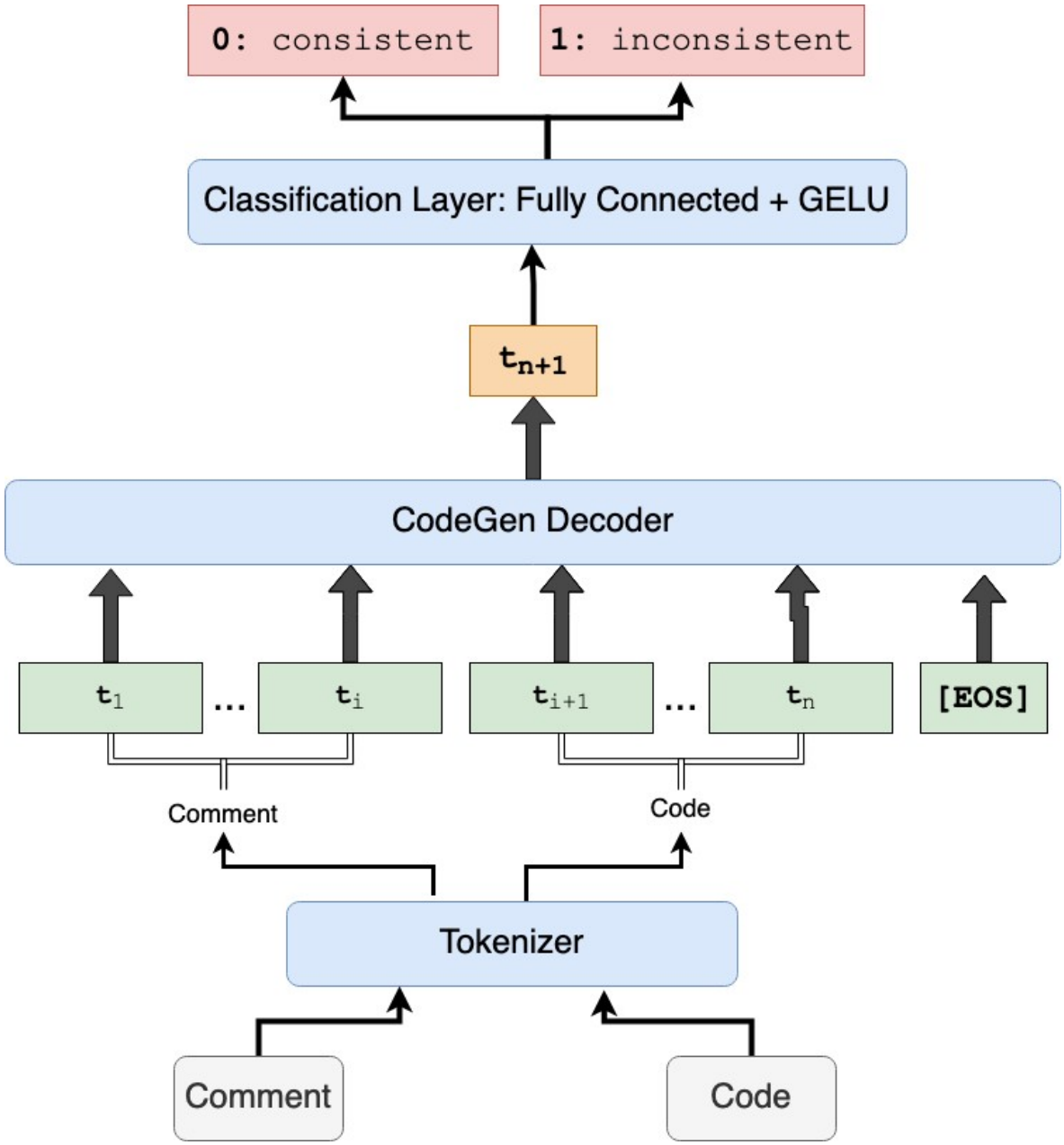


Рис. 3.3. Діаграма моделі CodeGen

Щоб перевірити, як масштабування розміру моделі впливає на результати, ми включили дві версії моделі CodeGen: 350 мільйонів і 2 мільярди параметрів. Ми включили цей тест, оскільки минулі роботи показали, що розмір моделі є значним фактором продуктивності LLM [17]. Однак більші моделі зазвичай вимагають більше даних для досягнення очікуваного покращення продуктивності, умови, якій цей набір даних може не відповідати. Тому ми перевіряємо, чи здатний більший розмір моделі покращити продуктивність цього набору даних. Якщо більша модель не

покращує продуктивність, то користувачі цього набору даних можуть знизити витрати на обчислення шляхом навчання менших моделей.

Деталі навчання

Репліковані моделі, зокрема DeepJIT, BERT та Longformer, були навчені з ідентичними гіперпараметрами до їхніх оригінальних публікацій. Там, де це було можливо, використовувався той самий код через їх опубліковані пакети реплікації. У випадку DeepJIT використовувалася лише базова модель без модуля нейронної мережі на основі графа AST для Java. Ми обрали цей підхід, оскільки цю частину мережі не можна було легко адаптувати для роботи з іншими мовами.

Нові моделі CodeBERT та CodeGen були кожна налаштована на одному процесорі Nvidia V100.

Ми використовували швидкість навчання 10^{-5} для CodeBERT та 10^{-6} для CodeGen. Кожна модель була навчена протягом 20 епох, а модель з найвищим показником F1 на валідаційному наборі, оціненому в кінці кожної епохи, була збережена та протестована.

3.1.4. Оцінка навченої моделі

Ми порівнюємо наші моделі з копіями з робіт [28] (DeepJIT) і [39] (BERT і Longformer). Загалом ми проводимо 6 експериментів: CodeBERT, CodeGen 350M, CodeGen 2B, DeepJit (без AST), BERT і Longformer.

Синтетичні оцінки минулих робіт були зосереджені на оцінці F1, визначеній згідно наступного рівняння:

$$F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Хоча ця оцінка дає збалансоване уявлення про продуктивність моделі, ми припускаємо, що вона не є найточнішим відображенням налаштування визначення узгодженості коментарів.

Через дисбаланс класів, притаманний цій проблемі, ми натомість оцінюємо результати за допомогою зваженої оцінки F1 із вагами класів, отриманими на основі частоти міток класів, знайдених під час створення набору даних. На практиці це призводить до приблизно 19:1 співвідношення ваги класу між негативними (послідовними) і позитивними (непослідовними) мітками. Іншими словами, ця метрика буде більш серйозно карати модель за помилкову позитивну класифікацію, коли вона вирішить, що послідовний приклад є позитивним. Однією з переваг цього підходу є те, що базовий класифікатор, який повертає «1» для кожного вхідного сигналу, більше не досягне сучасної продуктивності. У збалансованому налаштуванні такий класифікатор має ідеальне запам’ятовування 1,0 і точність 0,5 для оцінки F1 0,67, що пропонує сильну конкуренцію порівняно з оцінкою поточного рівня мистецтва 0,72. Використовуючи зважену оцінку, оцінка базового класифікатора зменшується до 0.1.

Продемонструємо цей ефект у таблицях 3.1 і 3.2. Таблиця 3.1 показує розрахунок оцінки F1, коли розміри класів збалансовані, що відповідає результатам, повідомленим у попередній роботі. Таблиця 3.2 демонструє проблему того, що той самий предиктор досягає набагато нижчого результату F1, коли до розрахунку застосовують справжні коефіцієнти класу.

Confusion Matrix – це таблиця, яка використовується для оцінки ефективності алгоритмів класифікації. Вона дозволяє детально проаналізувати, які саме помилки робить модель, і таким чином зрозуміти, які аспекти моделі потребують покращення.

Таблиця 3.1.

Збалансована Confusion Matrix зі 100 зразків

	Predicted Positive	Predicted Negative
Actual Positive	50	0
Actual Negative	50	0
Recall: 1.0	Precision: 0.5	F1: 0.67

Таблиця 3.2.

Confusion Matrix зі 100 зразків. Співвідношення 20:1 негативний: позитивний

	Predicted Positive	Predicted Negative
Actual Positive	5	0
Actual Negative	95	0
Recall: 1.0	Precision: 0.05	F1: 0.1

Ще одна перевага цього показника полягає в тому, що він узгоджує нашу оцінку з ключовими висновками груп інструментів розробки галузевих стандартів. У дослідженні [31], проведеному в Google, виявили, що одним із критеріїв корисного інструменту розробки є те, що він повинен давати менше 10% помилкових спрацьовувань. Цей коефіцієнт хибнопозитивних результатів відповідає точності 0,9. Примітно, що оцінка запам’ятовування здається менш важливою для сприйняття інструменту розробниками. Це розуміння збігається зі зваженим показником F1, який також сильно стимулює моделі, які можуть досягти високого результату точності.

3.1.5. Якісний аналіз визначення поведінки моделей

Щоб отримати уявлення про процес прийняття рішень LLM і краще зрозуміти поведінку моделей, ми виконуємо якісний аналіз, використовуючи карти помітності на вхідних токенах. Використовуючи інструмент візуалізації з відкритим кодом Ecco, ми можемо проаналізувати бали атрибуції для кожного вхідного токена на основі результату [1] Використовуючи техніку зворотного поширення, ми можемо виділити токени у вхідних даних (коментарі та коді), які найбільше сприяють прийняттю LLM класифікаційного рішення.

На рисунку 3.4 показано приклад візуалізації карти помітності з моделі аналізу настроїв. Ця візуалізація підкреслює фразу «найгірший фільм, коли-небудь створений», яка є найважливішою послідовністю токенів у вхідних даних для рішення моделі про класифікацію як негативного почуття.

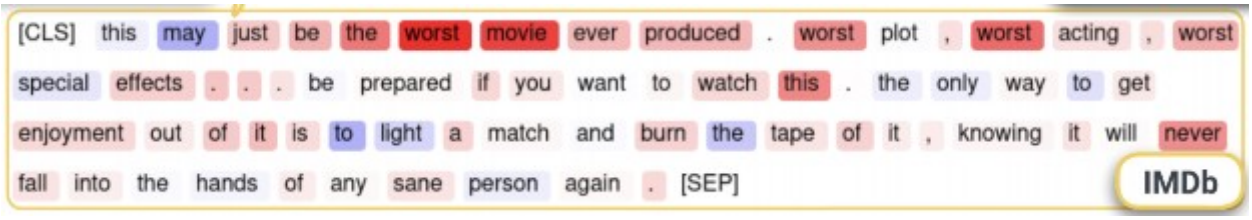


Рис. 3.4. Приклад візуалізації карти помітності з аналізу настроїв [8]

Хоча ми не можемо проаналізувати кожну вагу на кожному шарі цих моделей, ці візуалізації можуть вказувати на ключові шаблони у вхідних даних, отриманих моделлю. Наприклад, у наших даних ми можемо виявити, що найбільша вага надається назвам змінних у Java, коли помилка в коментарі пов’язана з невідповідністю між змінною, зазначеною в коментарі, та її використанням у коді.

Мета цих візуалізацій — зрозуміти, чому моделі зробили ті прогнози, які вони зробили. Оскільки інструмент візуалізації Ecco підходить для моделей трансформаторів декодера, як вхідні дані ми використали модель Codegen 350M, навчену на оригінальному наборі даних. Як незначне технічне зауваження: для сумісності з інструментом ми прикидаємося, що модель є повноцінним декодером із повним простором виводу 51 200 токенів, хоча насправді ми маємо ненульові ймовірності лише для перших двох класів токенів, що відповідають негативним і позитивний.

3.2. Методика оцінки узгодженості в різних мовах програмування

3.2.1 Збір даних для роботи

Наша методологія для цього дослідження передбачає збір нових наборів даних у Java, Python, Go та JavaScript, а потім виконання оцінки моделей для кожної мови.

Ми дотримуємося визначення у [28]. що ми позначаємо коментарі як послідовні чи непослідовні. Однак один ризик такого типу автоматичного маркування полягає в тому, що ми можемо включити помилкові негативи в

набір даних. Іншими словами, метою моделі є виявлення коментарів, які є суперечливими, але немає нічого, що б явно перешкоджало тому, щоб уже суперечливі коментарі були позначені як узгоджені в нашому наборі даних і таким чином забруднили результати. Ми робимо два кроки, щоб зменшити цей потенціал ризику:

1. Вибір проектів має відфільтрувати більшість коду низької якості.

2. Після створення наборів даних ми випадковим чином відбираємо 100 негативних прикладів для перевірки. Хоча ми не є експертами в кожній представленій кодовій базі, ми можемо інтерпретувати очевидні неправильні позначки, такі як коментар, який явно скопійовано з неправильного місця.

Для аналізу необхідних коментарів і коду з кожної мови, яку ми вивчаємо, ми використали інструмент генератора синтаксичного аналізатора tree-sitter, щоб отримати функції та коментарі до їх документації. У нашій постановці проблеми ми аналізуємо коментар документації та визначаємо підсумок як вміст коментаря до першої крапки, першого порожнього рядка або першої структурованої директиви (наприклад, @param у Java). Після попередньої роботи ми також включаємо лише ті функції, у яких принаймні один оператор повернення або тип повернення змінено між старою та новою версіями, оскільки зведення документації часто стосується результату функції. Лістинг 3.2 показує приклад функції з її коментарем до документації перед зміною, а лістинг 3.3 показує ту саму функцію після зміни. Цей приклад буде включено до набору даних, оскільки він задовольняє вимоги, які ми визначили.

Лістинг 3.2. Приклад функції до зміни

```
    /**
     * Returns the sum of the two inputs.
     * @param a the first input
     * @param b the second input
5    * @return the sum of the two inputs
     */
    public int operate(int a, int b) {
        return a + b;
    }
```

Лістинг 3.3. Приклад функції після зміни

```
/**
 * Returns the product of the two
     inputs.
 * @param a the first input
 * @param b the second input
 * @return the product of the two
     inputs
 */
public int operate(int a, int b) {
    return a * b;
}
```

Лістинги 3.4 та 3.5 демонструють іншу пару функцій. Однак, ця пара не буде включена до набору даних, оскільки лише рядки документації параметрів, що починаються з "@param", змінилися в коментарі документації. В нашому дослідженні для неструктурованих коментарів ми розглядаємо лише резюме коментаря, тому ця пара не відповідатиме вимогам.

Лістинг 3.4. Приклад функції до зміни

```
/**
 * Returns the sum of the two inputs.
 * @param a the first input
 * @param b the second input
 * @return the sum of the two inputs
 */
public int operate(int a, int b) {
    return a + b;
}
```

Лістинг 3.5. Приклад функції після зміни

```
/**
 * Returns the sum of the two inputs.
 * @param input1 the first input
 * @param input2 the second input
 * @return the sum of the two inputs
 */
public int operate(int input1, int
    input2) {
    return input1 + input2;
}
```

Щоб покращити якість наборів даних і зменшити шум, який міг би вплинути на результати, ми виконуємо кілька кроків для попередньої обробки та фільтрації:

- Дедуплікація: ми видаляємо всі точні копії того самого коментаря та коду після першого. Приклади цієї проблеми можуть виникати, коли блоки коду копіювали та вставляли з одного місця в інше.
- Визначаємо файл вихідного коду як можливо згенерований, якщо згенероване слово з'являється в перших 100 символах або на шляху до файлу. Наш набір даних видаляє всі приклади з цих файлів. Популярні засоби генерації коду, такі як controller-gen і grpc, залишають згенерований текст або в шляху до файлу, або в перших рядках згенерованого файлу.
- Видаляємо приклади, де з'являється слово deprecated, оскільки це рішення приймається поза кодовою базою і зазвичай не може бути визначено за вмістом функції.
- Видаляємо будь-які приклади, де єдиною відмінністю між старим і новим є пробіли. Це часто трапляється, коли проекти змінюють інструменти форматування або переходять із вкладок на пробіли.
- SATD: як обговорювалося у формулюванні проблеми, ми видаляємо приклади, де коментар, ймовірно, є самовизнаним технічним боргом (наприклад, містить TODO або FIXME). Ми робимо це, оскільки очікується, що узгодженість SATD буде протилежною іншим коментарям. Такий коментар є послідовним, якщо він точно описує щось, чого немає в коді. В іншому випадку, якщо код його реалізує, то коментар є непослідовним.

3.2.2. Моделювання рішення

У цьому дослідженні ми повинні вибрати між двома підходами до навчання кількома мовами. Або ми навчаємо кожну модель на кожній мові окремо (для кожного типу), або ми навчаємо одну модель на прикладах з усіх мов. У нашому тесті ми порівняли модель, навчену всіма мовами, з моделлю, навченою лише на Python, оцінену на тестовому наборі даних Python і дотримуючись деталей навчання, наведених у попередньому розділ. Результати цього тесту показано в таблиця 3.3.

Таблиця 3.3.

CodeBERT навчено на всіх 4 мовах або тільки на Python, оцінено на наборі тестів Python

	Precision	Recall	F1
CodeBERT (trained on Python)	0.62	0.49	0.55
CodeBERT (trained on 4 languages)	0.74	0.59	0.66

Хоча може здатися інтуїтивно зрозумілим, що модель, навчена кількома мовами, пожертвує продуктивністю порівняно зі спеціалізованою моделлю, але у попередніх експериментах на Python ми виявили, що модель, навчена на всіх прикладах, досягає кращого результату. Одне з можливих пояснень такої поведінки полягає в тому, що використання однієї моделі виграє від наявності більшого та більш різноманітного набору даних. Поєднуючи дані з кількох мов, ми збільшуємо загальний обсяг навчальних прикладів, потенційно сприяючи кращому узагальненню та більш надійній моделі. Таким чином, усі результати в цій роботі отримані моделями, які були навчені на всіх 4 мовах.

3.3. Опис продуктивності методики та моделі

Метою даної роботи є виявлення реакції програмістів на використання цієї моделі в соціальному середовищі через pull request'и на GitHub. Згідно з класифікацією в [41], це дослідження можна віднести до категорії пошуку рішень, оскільки воно спрямоване на вирішення практичної проблеми неконсистентних коментарів та вимірювання впливу запропонованого рішення. У цьому контексті основними суб’єктами дослідження є розробники програмного забезпечення, досліджувана поведінка — це реакція на pull request'и, а контекст — це розробка з відкритим вихідним кодом на платформі GitHub.

Ми обрали GitHub як платформу для проведення дослідження через його популярність, що дозволяє залучити підтримку різних організацій для усіх досліджуваних мов програмування. Для виконання цього етапу дослідження ми здійснимо класифікацію на обраних проєктах з відкритим кодом. Проєкти були відібрані на основі таких критеріїв: мінімум 25 зірок і не менше трьох pull request'ів, поданих у досліджуваний період. Порогове значення кількості зірок використовується для зосередження уваги на інженерно орієнтованих проектах [25], а вимога щодо активних pull request'ів дозволяє відфільтрувати тільки активні проєкти. Це підвищує ймовірність отримання відповіді від підтримувачів проєктів.

Після того, як модель виявить потенційно неконсистентний коментар, ми вручну переглядаємо питання. Якщо ми погоджуємось з ідентифікованою проблемою, то намагаємось вручну написати виправлення і подати його через pull request. Нашою метою є отримати приблизно 20 відповідей на наші pull request'и. Через високий рівень складності й необхідність ручного втручання, ми не можемо необмежено подавати запити, оскільки кожен з них вимагає значної кількості ручної роботи для написання відповідного виправлення і подання його у проєкт, з яким ми не маємо попереднього досвіду.

Проте ми вважаємо, що 20 відповідей є достатньою кількістю для формування висновків щодо цього проєкту, оскільки цей процес імітує ідеальний сценарій використання цієї моделі. У такому випадку програміст вводить неконсистентний коментар і одразу отримує сповіщення про це завдяки теоретично досконалій моделі. Така ситуація дозволяє з’ясувати, чи є коментарі корисними і чи вони потрібні, оскільки їх якість у цьому випадку є найвищою можливою. На підтвердження цього підходу ми звертаємось до попередніх досліджень. У дослідженні [42] було подано 16 pull request'ів для виправлення виявлених неконсистентностей, з яких 5 отримали відповідь від розробників. На рисунку 3.5 наведено приклад шаблону pull request'у, який ми використовуємо.

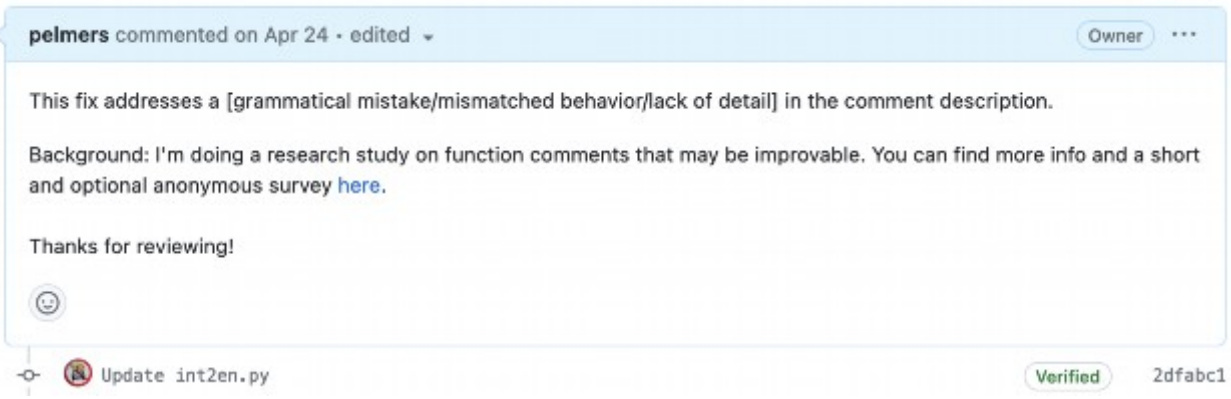


Рис. 3.5. Приклад pull request

Окрім запропонованого виправлення, ми додаємо до запиту анкету, в якій ми просимо пояснити рішення про прийняття або відхилення запиту і включаємо питання для оцінки відносної важливості консистентності коментарів у порівнянні з іншими проблемами, такими як складні ієрархії класів або беззмістовні назви змінних, з точки зору розчарування розробників під час розуміння коду. Точний перелік можливих факторів узято з результатів дослідження [47], де було виявлено, що недостатня кількість коментарів є значним джерелом розчарування.

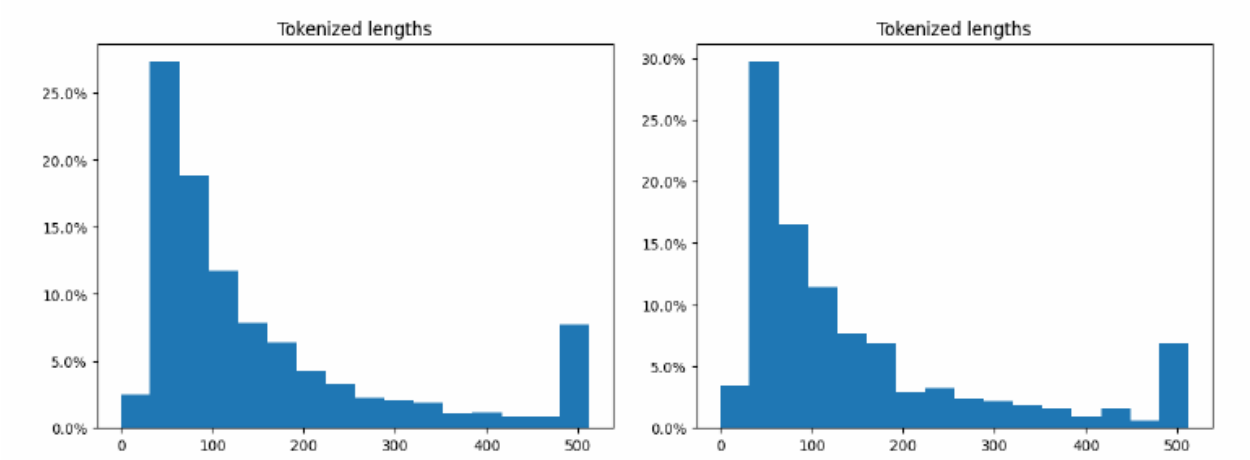
3.4. Навчання моделей машинного навчання на наборі даних Java

Як описано в попередньому розділі про методологію, наш підхід до відповіді на це запитання передбачає навчання моделей машинного навчання на наборі даних Java [28] .

3.4.1. Скорочення потоку вхідних даних

Треба зауважити, що Longformer має максимальну довжину вхідних даних 2048 токенів. Усі інші моделі в тесті використовували довжину вхідних даних щонайбільше 512 токенів, а довші приклади скорочувалися на цьому етапі. Ми оцінили ефект цього скорочення, обчисливши статистичні дані по довжині навчального набору. Наприклад, якщо кількість скорочених

прикладів буде дуже великою, ми очікуємо, що це негативно вплине на продуктивність моделі. Проте з 8398 навчальних зразків лише 585 мали загальну довжину маркерів (сума довжини коду та коментарів) більше 512. Середня довжина в навчальному наборі становить 304 маркери з медіаною 130. Таким чином, більшість зразків не будуть вплинути на обмеження довжини вхідних даних моделі. На рисунку 3.6 показано розподіл довжини прикладів у наборах для навчання та перевірки.



а) Тривалість навчання б) Тривалість перевірки

Рис. 3.6. Гістограми розподілу довжини в наборах для навчання та перевірки даних реплікації

3.4.2. Метрики навчання

Показники ефективності кожної моделі в цьому наборі даних показано в таблиця 3.4.

Таблиця 3.4.

Порівняння продуктивності тесту

	Precision	Recall	Unweighted F1	Weighted F1
DeepJIT Base[28]	0.796	0.527	0.634	0.272
BERT[39]	0.582	0.629	0.605	0.133
Longformer[39]	0.866	0.835	**0.850**	0.409
CodeBERT	**1.000**	0.704	0.826	**0.827**
Codegen 350M	0.995	**0.734**	0.845	0.816
Codegen 2B	0.997	**0.734**	0.846	0.817

Зважена оцінка F1 розраховується шляхом застосування ваги вибірки 19 до всіх негативних прикладів і ваги вибірки 1 для всіх позитивних випадків. Ці ваги відображають частки появи цих класів у вихідних наборах даних із попередньою дискретизацією, округлених до найближчого цілого числа.

3.4.3. Аналіз отриманих даних навчання для отримання класифікатора

Згідно з таблицею 3.4, нові моделі перевершують попередні моделі за вагою F1-оцінки. Зокрема, ми бачимо, що точність цих моделей є майже ідеальною, можливо, надто хорошою. Виявляється, так і є. Загалом, найкращою моделлю в цьому тесті була доопрацьована CodeBERT. Втрати при навчанні, точність та оцінки валідації F1 під час навчання для CodeBERT показані на рисунку 5.2.

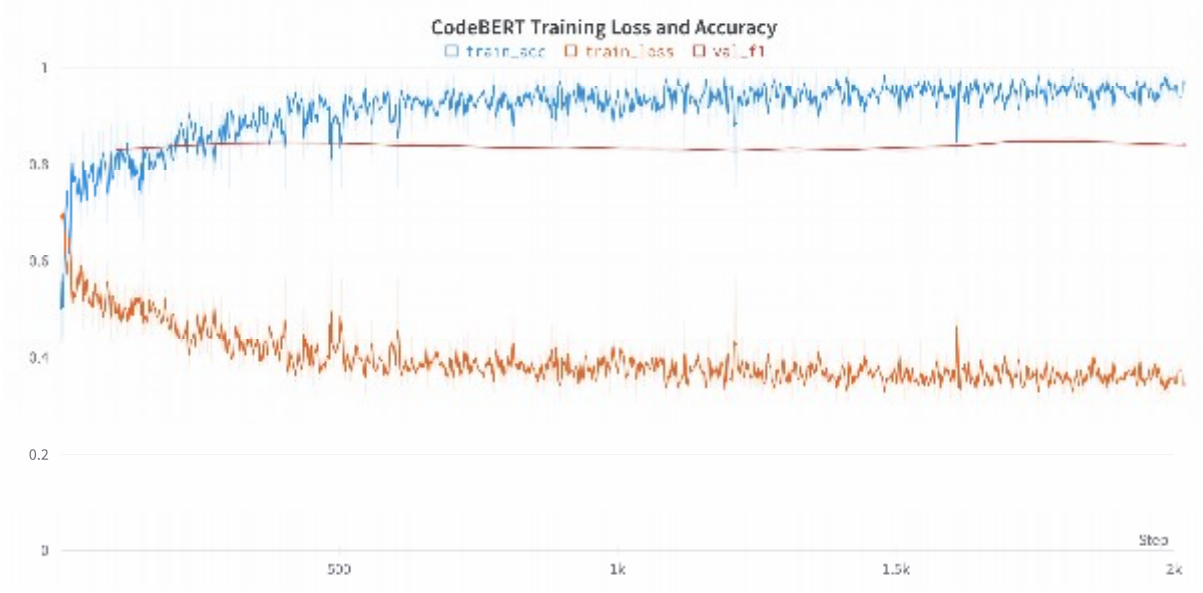


Рис. 3.7. Втрати при навчанні та міра точності F1 моделі CodeBERT на валідаційних даних. Кожна епоха вимірюється в 100 кроках

Цей графік показує, що точність та оцінки F1, здається, досягають плато дуже рано під час навчання. Наші репліковані моделі працюють аналогічно до оригінально опублікованих результатів. Зокрема, в цьому налаштуванні раніше використовувалися незважені оцінки F1.

Щоб пояснити отримані результати, ми використовували бібліотеку Ecco для візуалізації значущості входу моделей на вибраних прикладах введення [1].

Рисунок 3.8 ілюструє нормалізований внесок у кінцеву класифікацію за токеном для прикладу з набору даних реплікації.

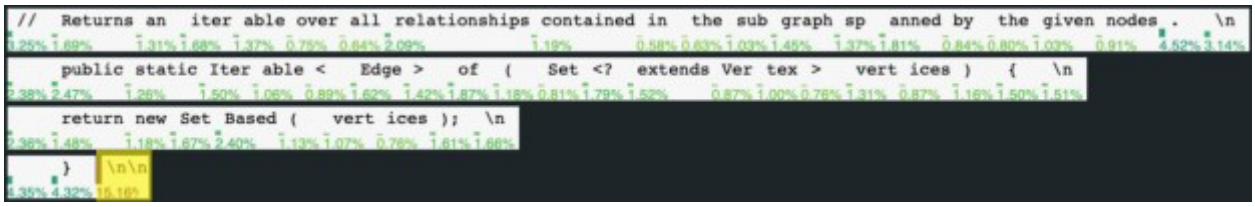


Рис. 3.8. Приклад візуалізації градієнтної норми, кінцеві символи нового рядка виділено

Зауважимо, що найбільший внесок, безсумнівно, надходить від останнього токена, пари символів нового рядка після тіла методу, що складає 15,16% від загальної суми. Щоб перевірити, чи модель була налаштована на фіксацію на цій особливості, ми видалили кінцеві нові рядки та візуалізували результат на рисунку 3.9. Відповідно, ми побачили, що цільовий клас змінився з позитивного на негативний, а також з'явився новий розподіл градієнтів.

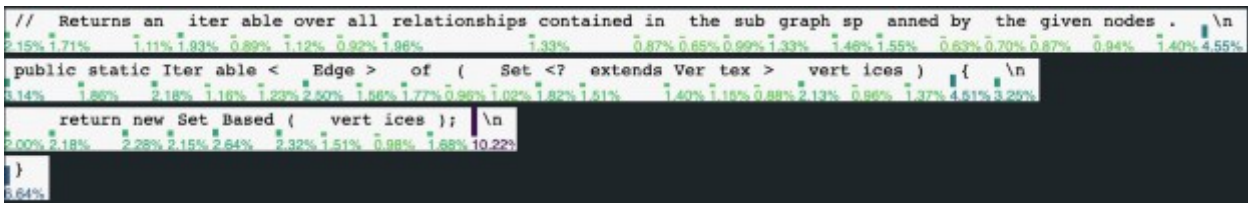


Рис. 3.9. Приклад візуалізації градієнтної норми, кінцеві символи нового рядка видалено

На основі цього ми створили дуже простий класифікатор на основі нового рядка на основі набору даних реплікації. Цей класифікатор просто повертає позитивний клас, якщо він бачить два символи нового рядка в кінці

введення. В іншому випадку він повертає негативний клас. Дане правило класифікації має наступний вигляд:

$$\text{prediction} = \begin{cases} \textit{inconsistent}, & \text{if example has 2 trailing new line characters} \\ \textit{consistent}, & \text{otherwise} \end{cases}$$

Ця проста модель фактично забезпечує ідентичний результат, що й наша 100% точна навчена модель CodeBERT на даних реплікації. Таким чином, ця модель також досягає найвищої точності та F1 у наборі даних реплікації, оскільки всі позитивні приклади в наборі даних містили два кінцевих символи нового рядка, включаючи набори для навчання, перевірки та тестування. Крім того, приблизно 70% негативних (послідовних) прикладів не мали двох завершальних символів нового рядка, що дозволяє моделям вивчати цю різницю під час навчання та використовувати її під час класифікації.

3.5. Застосування навченої моделі на інших мовах

Ще одним завдання пропонованої роботи є дослідження, наскільки добре методи, які досягають значних результатів на оригінальному наборі даних Java, працюватимуть на інших мовах. Щоб відповісти на це запитання, ми спочатку зібрали нові набори даних для чотирьох досліджуваних мов Java, Python, Go та JavaScript. Дотримуючись етапів нашої методології, ми перевірили, наскільки добре навчені на Java моделі оцінюватимуться на нових мовах, а потім навчили нові моделі на нових мовах для порівняння.

Для кожної мови ми видобули приклади з публічних репозиторіїв GitHub. Статистичні дані щодо кількості прикладів і репозиторіїв наведено в таблиці 3.5. Після цього процесу видобутку ми збалансували загальний набір даних, зберігши 22000 прикладів для кожної мови, повторно відібраних, щоб

досягти рівномірного балансу між позитивним і негативним класом. Видобуті проекти були відфільтровані із загального набору 3, 152, 515 проектів за різними критеріями.

Таблиця 3.5.

Статистика набору даних

Language	Number of Projects	Examples	Positive Examples
Java	2257	259,215	15,066
Python	3940	279,440	16,737
Go	2982	573,044	29,779
JavaScript	13,981	296,092	11,059
Total	23,161	1,407,791	72,641

У лістингу 3.6 наведено приклад позитивного випадку для мови Python. У цьому прикладі для даного методу і тіло коду, і текст коментаря були змінені в одному коміті, як показано у лістингу 3.7. Тому ми позначаємо старий коментар як такий, що не відповідає новому коду. Тут, під час перевірки, ми бачимо, що невідповідність полягає в зміні id на key для запиту до бази даних.

Лістинг 3.6. Приклад на Python (до)

```
# Comment: Fetch all tasks from database
    ordered by id.
def get_all_tasks() -> List[Task]:
    with db_session() as session:
        tasks = session.query(Task).
            order_by(Task.id).all()
    return tasks
```

Лістинг 3.7. Приклад на Python (після)

```
# Comment: Fetch all tasks from database
    ordered by key.
def get_all_tasks(include_disabled: bool
    = False) -> List[Task]:
    query = Task.query
    if not include_disabled:
        query = query.filter(~Task.
            disabled)
    return query.order_by(Task.key).all()
```

Таблиця 3.6 і рисунок 3.10 відображають статистичні дані про довжину прикладів у навчальному наборі за мовами в термінах кількості токенів після токенізації за допомогою токенізатора CodeBERT.

Таблиця 3.6.

Середня та медіанна довжини набору даних (кількість токенів після токенізації CodeBERT)

	Mean	Median
Java	170	110
Python	292	201
Go	328	210
JavaScript	293	185

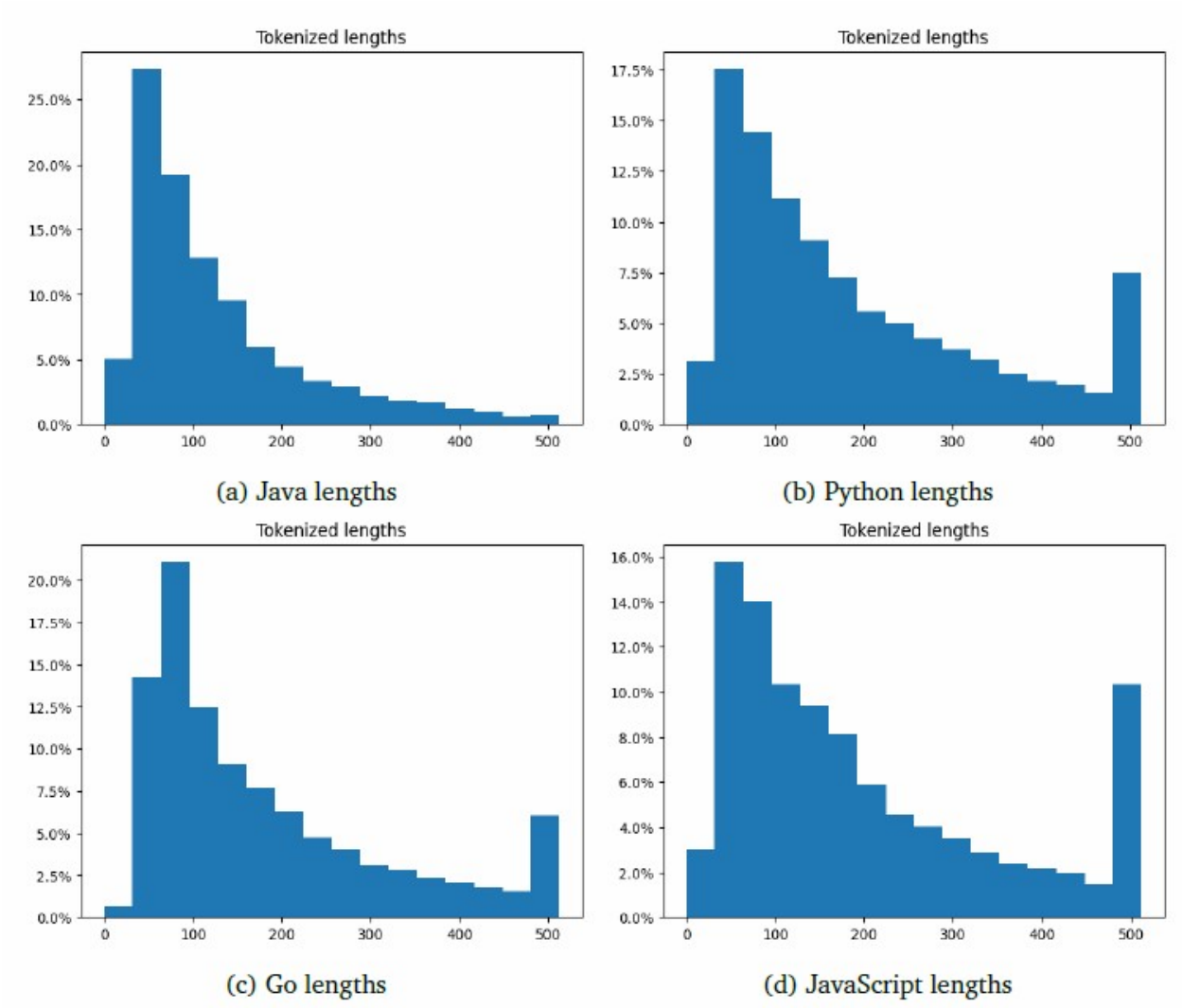


Рис. 3.10. Гістограми розподілу довжини в навчальних наборах за різними мовами програмування

Однорядкові методи Java. Зауважимо, що довжини прикладів на Java були значно коротшими, ніж на інших мовах. Насправді, 7721 з 17600 прикладів (44 відсотки) у тренувальному наборі мали лише одну операцію в тілі методу. Одним із пояснень цього результату є те, що програми на Java сильно покладаються на перенаправлення та перевантаження методів, що призводить до багатьох методів, які просто повертають результат виклику іншого методу. Ми наводимо приклад однорядкового методу в лістингу 3.8.

Лістинг 3.8. Однорядковий приклад засобами Java (до)

```
// Comment: Calls #createConfigStore(
    ServiceSpecification, Collection)
    with an empty list of custom
    deserialization types.
public static ConfigStore<
    ServiceSpecification>
    createConfigStore(
        ServiceSpecification
            serviceSpecification) throws
            ConfigStoreException {
4       return createConfigStore(
            serviceSpecification, Collections
            .emptyList());
}
```

Лістинг 3.9. Однорядковий приклад засобами Java (після)

```
// Comment: Calls #createConfigStore(
    ServiceSpec, Collection) with an
    empty list of custom deserialization
    types.
public static ConfigStore<ServiceSpec>
    createConfigStore(
        ServiceSpec serviceSpec) throws
            ConfigStoreException {
    return createConfigStore(serviceSpec,
        Collections.emptyList());
5 }
```

Це спостереження цікаве, оскільки коротші методи надають менше контексту для моделі, щоб прийняти рішення про класифікацію. Таким чином, ми можемо очікувати, що моделі буде складніше класифікувати приклади на Java, ніж на інших мовах.

Дослідження пробілів. Ми виявили, що всі непослідовні приклади в оригінальному наборі даних мали два пробіли в кінці. Щоб перевірити, чи це було властиво даним і, отже, точним предиктором непослідовності коментаря, ми дослідили розподіл пробілів в кінці в новозібраному тренувальному наборі. У нововидобутому наборі даних GitHub ми перевірили це, також підрахувавши кількість пробілів в кінці після кожного прикладу відповідно до його мітки. Наші результати показано в таблиці 3.7.

Таблиця 3.7.

Результати для нових рядків у Java

Number of trailing newline characters	1	2	3+
Negative	19,653	213,401	10,585
Positive	991	13,581	456

Тут ми не бачимо значної кореляції між міткою та кількістю пробілів після методу. У негативному випадку приблизно 10% прикладів мають один пробіл. У позитивному випадку цей коефіцієнт становить приблизно 8%.

Отже, ми можемо зробити висновок, що значна різниця в ефективності між нововидобутим набором даних Java та оригінальним набором даних Java пов'язана з цим артефактом у процесі збору оригінальних даних.

Незрозуміло, що спричинило цей артефакт анотації в наборі даних. Наша найкраща здогадка полягає в тому, що він виник через помилку в процесі збору даних, коли парсер тіла методу міг бути змінений між збиранням негативних та позитивних прикладів, так що він включав додаткові нові рядки в позитивних прикладах, але не в негативних.

Висновки до розділу

У цьому розділі було детально описано методологію застосування мовних моделей для класифікації узгодженості та послідовності коментарів у коді, а також досліджено ефективність цих підходів для різних мов

програмування. Основними етапами роботи стали вибір коментарів, створення відповідного набору даних, вибір і навчання моделей, а також їх оцінка та аналіз результатів.

Процес вибору коментарів було систематизовано з метою виділення релевантних фрагментів для подальшого аналізу, що стало важливим етапом перед початком моделювання. Для цього був створений набір даних на основі різних проєктів з відкритим кодом, який забезпечив основу для навчання моделей. Було також описано процес моделювання, у якому особливу увагу приділено вибору відповідних моделей та методам їх оцінки.

Застосування навченої моделі до інших мов програмування дало змогу оцінити універсальність запропонованого підходу. Під час оцінки результатів виявлено, що продуктивність моделі може варіюватися залежно від специфіки мови програмування та складності її синтаксису, що потребує додаткового налаштування моделей для кожного випадку.

Також проведено якісний аналіз визначення поведінки моделей, який дав змогу глибше зрозуміти, як моделі реагують на різні типи узгодженості в коментарях, та визначити чинники, які впливають на ефективність класифікації. Це стало основою для подальшого вдосконалення методології та моделей.

Навчання моделей машинного навчання на даних Java продемонструвало, що для досягнення високої точності класифікації необхідно використовувати спеціалізовані підходи до скорочення потоку вхідних даних, а також відповідні метрики для оцінки результатів. Аналіз отриманих даних дав можливість створити точні класифікатори, які забезпечують високу продуктивність моделі.

Загалом, запропонована методологія показала свою ефективність у виявленні неконсистентних коментарів у коді, що робить її корисним інструментом для підтримки якості програмного забезпечення.

ВИСНОВКИ

У магістерській роботі було проведено дослідження методів і методології класифікації узгодженості та послідовності коду за допомогою мовних моделей. В рамках дослідження здійснено аналіз взаємодії мовних моделей, орієнтованих на програмний код, з задачами узгодженості коментарів та їх відповідності коду. Завдання класифікації полягало у визначенні, чи є коментарі узгодженими або ж не відповідають коду, до якого вони відносяться. Особлива увага була приділена коментарям на рівні методів або функцій, які безпосередньо стосуються конкретного методу або функції в коді.

Для забезпечення якісного аналізу було проведено масштабний збір даних з програмного забезпечення на платформі GitHub, що дало змогу створити збалансований набір даних. Окрім того, для перевірки ефективності моделей застосовано вручну підібраний тестовий набір, а сам підхід був протестований на сучасних проєктах з відкритим вихідним кодом, шляхом аналізу коментарів у pull request'ах на GitHub.

Загальні результати демонструють, що модель CodeBERT перевершує попередні підходи як на старих наборах даних, так і на новоствореному. Це підтверджує гіпотезу про те, що великі попередньо навчені мовні моделі здатні значно зменшити необхідність у створенні специфічних для мови функцій. Водночас було виявлено суттєву аномалію у попередньому наборі даних для Java, яка могла спотворити результати попередніх досліджень. Аномалія полягала в артефакті пробілів, які використовувалися моделями для розрізнення позитивних і негативних прикладів, що не відповідало вихідним даним. Усі позитивні приклади в існуючому наборі даних містили додаткові символи нового рядка, що дозволяло тривіальним класифікаторам досягати 100% точності. Проведено перевірку нового набору даних, яка підтвердила, що цей набір не вразливий до такої проблеми, завдяки чому результати оцінювання моделей виявилися об'єктивними.

На етапі тестування було виявлено розрив між результатами моделі на навчальному наборі даних та її продуктивністю під час порівняльного тестування. Це, ймовірно, пояснюється складністю тестового завдання, яке вимагало від моделі більш точного розрізнення між класами порівняно з навчальними завданнями. Дослідження підкреслило складність реальних ситуацій та небезпеку припущення, що висока ефективність на етапі навчання автоматично забезпечує таку ж продуктивність у реальних умовах.

Ми сподіваємось, що результати цього дослідження та нові набори даних, опубліковані разом із роботою, сприятимуть подальшому розвитку галузі. Наші набори даних із мітками щодо коду та коментарів можуть бути використані для навчання моделей генерації коментарів, а наш набір коментарів до pull request'ів може допомогти у створенні ботів для автоматизації перевірки запитів або для лінгвістичного аналізу у проєктах з відкритим кодом.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. J Alammar. Ecco: An open source library for the explainability of transformer language models. In Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: System Demonstrations. Association for Computational Linguistics, 2021
2. Huanchao Chen, Yuan Huang, Zhiyong Liu, Xiangping Chen, Fan Zhou, and Xiaonan Luo. Automatically detecting the scopes of source code comments. Journal of Systems and Software, 153:45–63, 2019
3. Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.
4. Seungtaek Choi, Myeongho Jeong, Hojae Han, and Seung-won Hwang. C2l: Causally contrastive learning for robust text classification. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 36, pages 10526–10534, 2022.
5. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018.
6. Peter Elmers. Github public repository metadata. https://www.kaggle.com/datasets/pelmers/github-repository-metadata-with-5-stars.
7. Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In Proceedings of the 26th Conference on Program Comprehension, pages 286–296, 2018.
8. Nils Feldhus, Leonhard Hennig, Maximilian Nasert, Christopher Ebert, Robert Schwarzenberg, and Sebastian Mller. Saliency map verbalization:

Comparing feature importance representations from model-free and instruction-based methods. In Proceedings of the 1st Workshop on Natural Language Reasoning and Structured Explanations (NLRSE), pages 30–46, 2023.

9. Beat Fluri, MichaelWursch, and Harald C Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In 14th Working Conference on Reverse Engineering (WCRE 2007), pages 70–79. IEEE, 2007.
10. Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800gb dataset of diverse text for language modeling. arXiv preprint arXiv:2101.00027, 2020.
11. Jesús M González-Barahona and Gregorio Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. Empirical Software Engineering, 17:75–89, 2012.
12. Miguel Hoyos Ruge et al. Analysis of software engineering automation tools for go. 2021.
13. Shin Hwei Tan, Chunfeng Hu, Ziqiang Li, Xiaowen Zhang, and Ying Zhou. Github-oss fixit: Fixing bugs at scale in a software engineering course. arXiv e-prints, pages arXiv–2011, 2020.
14. Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In Proceedings of the 11th working conference on mining software repositories, pages 92–101, 2014.
15. Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, JeffreyWu, and Dario Amodei. Scaling laws for neural language models. arXiv preprint arXiv:2001.08361, 2020.
16. Joel Lehman, Jeff Clune, and Dusan Misevic. The surprising creativity of digital evolution. In Artificial Life Conference Proceedings, pages 55–56.

MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info . . . , 2018.

17. Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. CoRR, abs/1907.11692, 2019.
18. Zhiyong Liu, Huanchao Chen, Xiangping Chen, Xiaonan Luo, and Fan Zhou. Automatic detection of outdated comments during code changes. In 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), volume 1, pages 154–163. IEEE, 2018.
19. Christopher Manning and Hinrich Schutze. Foundations of statistical natural language processing. MIT press, 1999.
20. Diego Marcilio, Carlo A Furia, Rodrigo Bonifácio, and Gustavo Pinto. Automatically generating fix suggestions in response to static code analysis warnings. In 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 34–44. IEEE, 2019.
21. Tim Menzies and Martin Shepperd. Special issue on repeatable results in software engineering prediction, 2012.
22. Shervin Minaee, Nal Kalchbrenner, Erik Cambria, Narjes Nikzad, Meysam Chenaghlu, and Jianfeng Gao. Deep learning–based text classification: a comprehensive review. ACM computing surveys (CSUR), 54(3):1–40, 2021.
23. Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. Empirical Software Engineering, 22:3219–3253, 2017.
24. Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint, 2022.

25. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in Neural Information Processing Systems.
26. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
27. Chen, M., Tworek, J., Jun, H., Yuan, Q., Dehghani, M., Abid, A., ... & Ziegler, D. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.
28. Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. W. (2021). Unified pre-training for program understanding and generation. arXiv preprint arXiv:2101.08869.
29. Hellendoorn, V. J., et al. (2020). Global relational models of source code. Proceedings of the ACM on Programming Languages.
30. Feng, Z., Guo, D., Tang, H., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.
31. Svyatkovskiy, A., Deng, S., Fu, S., & Sundaresan, N. (2020). IntelliCode Compose: Code generation using transformer. Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.
32. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR).
33. Gupta, R., Pal, A., Kanade, A., & Shevade, S. (2020). Deepfix: Fixing common C language errors by deep learning. Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation.
34. Bhatia, A., & Singh, S. (2021). Code summarization using transformer: A comprehensive review. Journal of Systems and Software.

35. Kanade, A., Maniatis, P., Balog, M., & Singh, P. (2020). Learning and evaluating contextual embedding of source code. Proceedings of the 37th International Conference on Machine Learning.
36. Puri, R., Kung, D. S., Janssen, G., et al. (2021). Project CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. arXiv preprint arXiv:2105.12655.
37. Liu, F., & Luo, G. (2020). Code completion with neural attention and pointer networks. ACM Transactions on Software Engineering and Methodology (TOSEM).
38. Yin, P., & Neubig, G. (2017). A syntactic neural model for general-purpose code generation. arXiv preprint arXiv:1704.01696.
39. Husain, H., et al. (2019). Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436.
40. Roziere, B., Lachaux, M. A., Chanussot, L., & Lample, G. (2020). Unsupervised translation of programming languages. Advances in Neural Information Processing Systems.
41. Tufano, M., Watson, C., Bavishi, R., & Poshyvanyk, D. (2019). Learning meaningful code changes via neural machine translation. arXiv preprint arXiv:1812.08693.
42. Peng, H., He, S., Liu, Z., Liu, M., Li, H., & Sun, M. (2021). Towards programming language modeling using GPT-2. IEEE Access.
43. Fernandes, P., Leitão, P., & Morgado, E. (2021). Classifying code snippets from machine learning projects using pre-trained language models. arXiv preprint arXiv:2109.06750.
44. Ahamed, M. S., Ahmad, W. U., Chakraborty, S., & Ray, B. (2021). Learning to represent source code as knowledge graphs for developer assistance. arXiv preprint arXiv:2104.00786.
45. Wan, Y., Yang, Y., & Zhao, K. (2019). Improving code summarization with block-level semantic representations using code structure and machine learning. IEEE Transactions on Software Engineering.

46. Fu, X., Li, G., Jin, Z., & Zhu, H. (2020). A code clone detection method based on deep learning for big code. IEEE Transactions on Software Engineering.
47. Lachaux, M. A., Roziere, B., Chanussot, L., & Lample, G. (2021). DOBF: A deobfuscation pre-training objective for programming languages. arXiv preprint arXiv:2102.07492.
48. Hellendoorn, V. J., Devanbu, P., Movshovitz-Attias, Y., & Sutton, C. (2019). Are deep neural networks the best choice for modeling source code? arXiv preprint arXiv:1902.01561.
49. Guo, D., Ren, S., Lu, S., Tang, D., Duan, N., Zhou, M., & Yin, J. (2021). GraphCodeBERT: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.
50. Dam, H. K., et al. (2018). Automatic code completion using neural attention and pointer networks. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.
51. Tabassum, S., & Roy, S. (2021). Source code error classification using deep learning. Proceedings of the IEEE International Conference on Smart Systems and IoT (ICSSI).
52. Nguyen, A. T., & Nguyen, T. N. (2019). A systematic literature review of machine learning techniques for software quality prediction. Information and Software Technology.