

Київський національний університет імені Тараса Шевченка
Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка
Міністерство освіти і науки України

Кваліфікаційна наукова праця на правах рукопису

СЕМЬОНОВ БОГДАН ОЛЕКСАНДРОВИЧ

УДК 004.4:004.8

ДИСЕРТАЦІЯ

ПРОЄКТУВАННЯ І ЗАСТОСУВАННЯ НЕЙРОННИХ МЕРЕЖ ДЛЯ СТВОРЕННЯ МЕТОДІВ ОЦІНЮВАННЯ ЯКОСТІ ТА ГЕНЕРАЦІЇ ПОВІДОМЛЕНЬ ПРО ВНЕСЕНІ ЗМІНИ В СИСТЕМАХ КОНТРОЛЮ ВЕРСІЙ

123 «Комп'ютерна інженерія»

12 «Інформаційні технології»

Подається на здобуття наукового ступеня доктора філософії

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

_____ Семьонов Богдан Олександрович

Науковий керівник: Погорілий Сергій Дем'янович, доктор технічних наук,
професор

Київ - 2026

АНОТАЦІЯ

Семьонов Б.О. Проєктування і застосування нейронних мереж для створення методів оцінювання якості та генерації повідомлень про внесені зміни в системах контролю версій. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 123 «Комп'ютерна інженерія». – Київський національний університет імені Тараса Шевченка, Київ, 2026.

Робота присвячена дослідженню проєктування і застосування нейронних мереж різної архітектури для створення методів оцінювання якості та автоматичної генерації повідомлень про внесені зміни (commit messages) в системах контролю версій.

Повідомлення про внесені зміни є невід'ємним елементом процесу супроводу програмного забезпечення, який становить від 60 % до 80 % загальних витрат протягом життєвого циклу програмного продукту. Якість повідомлень безпосередньо впливає на ефективність аналізу історії змін, локалізацію дефектів, рефакторинг та командну комунікацію. Водночас емпіричні дослідження свідчать, що значна частина повідомлень у реальних проєктах є надто короткими, шаблонними або семантично неінформативними, що обґрунтовує необхідність розроблення засобів автоматизованого аналізу та генерації таких повідомлень.

У роботі здійснено комплексний аналіз існуючих методів формування та оцінювання якості повідомлень про внесені зміни, що охоплює лексичний та текстовий аналіз, статистичні методи (n-грамний аналіз, TF-IDF, тематичне моделювання LDA), методи машинного навчання та обробки природної мови (SVM, Random Forest, LSTM, трансформери) та гібридні підходи. Проведено систематизацію методів автоматичної генерації повідомлень, включаючи евристичні та шаблонні методи, підходи на основі рекурентних нейронних мереж, трансформерні моделі з попереднім навчанням та методи із використанням структурної інформації абстрактних синтаксичних дерев.

Згідно зі сформованими метою та задачами роботи створено методи оцінки якості та автоматичної генерації повідомлень про внесені зміни у системах

контролю версій на основі моделей нейронних мереж різної архітектури; досліджено ефективність застосування рекурентних та трансформерних моделей для задачі бінарної класифікації повідомлень; розроблено методи мовно-специфічного донавчання та семантичного збагачення ідентифікаторів вихідного коду для підвищення якості генерації повідомлень. Нижче наведені **основні результати та наукова новизна**.

В результаті аналізу впливу попередньо навчених моделей векторного представлення слів на якість бінарної класифікації повідомлень показано доцільність:

- застосування методу FastText для технічних текстів завдяки підтримці підслівних n-грам, що дозволяє формувати вектори для незнайомих слів – технічних аббревіатур, назв бібліотек та нестандартних токенів, характерних для повідомлень про внесені зміни;
- використання трансферного навчання з попередньо навченими векторами, що суттєво підвищує якість класифікації: модель SO_vectors_200 (Word2Vec, навчена на технічних текстах Stack Overflow) досягла найвищого Accuracy = 0,861.

Розроблено метод оцінювання якості повідомлень про внесені зміни у системах контролю версій на основі рекурентних нейронних мереж. Побудовано серію із семи моделей-кандидатів зростаючої складності: від базової моделі з повнозв'язним шаром до двошарових двонаправлених рекурентних мереж (BiLSTM, BiGRU). Для всіх моделей використано уніфіковану архітектуру із фіксованою загальною кількістю нейронів рекурентних шарів, що забезпечує коректність порівняльного аналізу. Модель BiGRU з одним шаром досягла оптимального балансу між Accuracy (0,811) та F1-score (0,629).

Розроблено метод оцінювання якості повідомлень на основі трансформерних моделей із застосуванням методології трансферного навчання. Досліджено шістнадцять конфігурацій на базі чотирьох сімейств моделей – BERT, RoBERTa, DistilBERT та XLM-RoBERTa. Найвищу якість продемонструвала модель

roberta_large_en (Accuracy = 0,876, F1-score = 0,719), яка перевершила найкращу RNN-модель на 1,5 відсоткових пункти за Accuracy та на 1,8 за F1-score.

Досліджено застосування трансформерних моделей типу кодувальник-декодувальник для автоматичної генерації повідомлень про внесені зміни. Обґрунтовано вибір та досліджено дев'ять моделей: T5, FLAN-T5, CodeT5, CodeT5+ та BART. Показано, що моделі, орієнтовані на вихідні тексти програм (CodeT5, CodeT5+), суттєво перевершують моделі загального призначення: середній BLEU для CodeT5+ становить 6,55 проти менше 1,0 для T5 та FLAN-T5.

Розроблено метод мовно-специфічного донавчання трансформерних моделей для систем контролю версій. Сформовано навчальні корпуси для шести мов програмування (C/C++, Ruby, Python, Java, JavaScript, PHP) загальним обсягом 271 504 навчальні пари. Метод передбачає незалежне донавчання моделі на корпусі кожної мови, що дозволяє моделі адаптуватися до синтаксичних та семантичних особливостей конкретної мови програмування.

Вперше запропоновано метод семантичного збагачення ідентифікаторів вихідного коду через автоматичне розкриття аббревіатур для підвищення якості автоматичної генерації повідомлень про внесені зміни. Розроблено п'ятиетапний конвеєр збагачення, що включає вилучення ідентифікаторів через побудову абстрактного синтаксичного дерева, сегментацію за конвенціями іменування, генерацію кандидатів розкриття на основі словникового підходу та контекстної моделі CodeBERT, семантичне ранжування кандидатів та формування збагачених текстових представлень.

Виконано експериментальну перевірку ефективності застосування запропонованих методів. Підтверджено, що комбінація мовно-специфічного донавчання та семантичного збагачення забезпечує послідовне накопичення покращень: перехід від базової моделі T5 до донавченої CodeT5+ збільшує BLEU у 10,8 разу, а подальше застосування семантичного збагачення додає ще 35 % приросту. Найкращі результати продемонструвала модель codet5p-770m із мовно-специфічним донавчанням та семантичним збагаченням ідентифікаторів (BLEU = 11,15, ROUGE-1 = 44,62). Найвищі абсолютні результати після збагачення

зафіксовано для JavaScript (BLEU = 16,47, BERTScore = 0,92), а найбільший відносний приріст – для Java (+51,5 % за BLEU) та C/C++ (+49,7 %).

Проведено порівняльний аналіз ефективності застосування технологій паралельних обчислень (GPGPU та TPU) для прискорення навчання нейронних класифікаторів. Встановлено, що GPU-акселерація (NVIDIA Tesla T4) забезпечує середнє прискорення у 45 разів порівняно з CPU та у 15 разів порівняно з TPU. Найвище прискорення GPU/CPU (56×) зафіксовано для моделі LSTM, що вказує на можливість впровадження запропонованих методів у системи загального призначення.

Ключові слова: оцінка якості повідомлень про внесені зміни, автоматична генерація повідомлень, системи контролю версій, рекурентна нейронна мережа, трансформерна модель, BERT, RoBERTa, CodeT5, CodeT5+, бінарна класифікація, трансферне навчання, мовно-специфічне донавчання, семантичне збагачення ідентифікаторів, розкриття аббревіатур, BLEU, BERTScore.

SUMMARY

Semonov B. O. Design and Application of Neural Networks for Developing Methods for Quality Assessment and Generation of Commit Messages in Version Control Systems. – Qualification research work submitted as a manuscript.

Dissertation for the degree of Doctor of Philosophy (PhD) in specialty 123 “Computer Engineering.” – Taras Shevchenko National University of Kyiv, Kyiv, 2026.

The dissertation is devoted to investigating the design and application of neural networks with different architectures for developing methods for quality assessment and automatic generation of commit messages in version control systems.

Commit messages are an integral component of software maintenance, which accounts for 60% to 80% of total costs over the software product life cycle. Message quality directly affects the efficiency of change-history analysis, defect localization, refactoring, and team communication. At the same time, empirical studies indicate that a substantial proportion of messages in real-world projects are too short, template-based, or semantically uninformative, which substantiates the need for automated analysis and generation tools.

The work provides a comprehensive analysis of existing methods for constructing and assessing commit message quality, covering lexical and textual analysis; statistical methods (n-gram analysis, TF-IDF, LDA topic modeling); machine learning and natural language processing methods (SVM, Random Forest, LSTM, transformers); and hybrid approaches. Methods for automatic message generation are systematized, including heuristic and template-based techniques, recurrent neural network approaches, pretrained transformer models, and methods leveraging structural information from abstract syntax trees.

In accordance with the stated goal and objectives, the dissertation develops methods for quality assessment and automatic generation of commit messages in version control systems based on neural models of different architectures; investigates the effectiveness of recurrent and transformer models for binary message classification; and proposes language-specific fine-tuning and semantic enrichment of source-code

identifiers to improve message generation quality. **The main results and scientific novelty** are summarized below.

As a result of analyzing the impact of pretrained word vector representations on binary message classification, the study demonstrates the feasibility of:

- applying FastText to technical texts due to its subword n-gram support, which enables vector construction for unseen words – technical abbreviations, library names, and non-standard tokens typical of commit messages;
- using transfer learning with pretrained vectors, which substantially improves classification performance: the SO_vectors_200 model (Word2Vec trained on Stack Overflow technical texts) achieved the highest Accuracy = 0.861.

A method for assessing commit message quality in version control systems based on recurrent neural networks is developed. A series of seven candidate models of increasing complexity is constructed: from a baseline model with a fully connected layer to two-layer bidirectional recurrent networks (BiLSTM, BiGRU). A unified architecture with a fixed total number of recurrent-layer neurons is used for all models, ensuring a valid comparative analysis. The single-layer BiGRU model achieved the best balance between Accuracy (0.811) and F1-score (0.629).

A method for message quality assessment based on transformer models using transfer learning is proposed. Sixteen configurations across four model families – BERT, RoBERTa, DistilBERT, and XLM-RoBERTa – are investigated. The best performance is demonstrated by roberta_large_en (Accuracy = 0.876, F1-score = 0.719), which outperformed the best RNN model by 1.5 percentage points in Accuracy and 1.8 percentage points in F1-score.

The application of encoder–decoder transformer models to automatic commit message generation is studied. Nine models are justified and evaluated: T5, FLAN-T5, CodeT5, CodeT5+, and BART. It is shown that models oriented toward source code (CodeT5, CodeT5+) substantially outperform general-purpose models: the mean BLEU for CodeT5+ is 6.55, compared with below 1.0 for T5 and FLAN-T5.

A method of language-specific fine-tuning of transformer models for version control systems is developed. Training corpora are compiled for six programming languages (C/C++, Ruby, Python, Java, JavaScript, PHP) with a total of 271,504 training pairs. The method involves independent fine-tuning on each language-specific corpus, enabling the model to adapt to the syntactic and semantic characteristics of a particular programming language.

For the first time, a method of semantic enrichment of source-code identifiers through automatic abbreviation expansion is proposed to improve automatic commit message generation. A five-stage enrichment pipeline is developed, including identifier extraction via abstract syntax tree construction; segmentation according to naming conventions; candidate expansion generation using a dictionary-based approach and the contextual CodeBERT model; semantic ranking of candidates; and formation of enriched textual representations.

Experimental validation of the proposed methods is performed. It is confirmed that combining language-specific fine-tuning and semantic enrichment yields consistent cumulative improvements: transitioning from the baseline T5 model to the fine-tuned CodeT5+ increases BLEU by $10.8\times$, while subsequent semantic enrichment adds an additional 35% gain. The best results are achieved by codet5p-770m with language-specific fine-tuning and identifier semantic enrichment (BLEU = 11.15, ROUGE-1 = 44.62). The highest absolute post-enrichment results are recorded for JavaScript (BLEU = 16.47, BERTScore = 0.92), whereas the largest relative gain is observed for Java (+51.5% in BLEU) and C/C++ (+49.7%).

A comparative analysis of parallel computing technologies (GPGPU and TPU) for accelerating the training of neural classifiers is conducted. It is established that GPU acceleration (NVIDIA Tesla T4) provides an average speedup of $45\times$ compared to CPU and $15\times$ compared to TPU. The highest GPU/CPU speedup ($56\times$) is observed for the LSTM model, indicating the feasibility of deploying the proposed methods in general-purpose systems.

Keywords: commit message quality assessment, automatic message generation, version control systems, recurrent neural network, transformer model, BERT, RoBERTa, CodeT5, CodeT5+, binary classification, transfer learning, language-specific fine-tuning, semantic enrichment of identifiers, abbreviation expansion, BLEU, BERTScore.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Наукові праці, в яких опубліковані основні наукові результати дисертації:

1. Semonov, B. O., & Pogorilyy, S. D. (2025). Research of the application of GPGPU and TPU technologies for ensuring comment quality in version control systems. *Problems in Programming*, (1), 24–37.
<https://doi.org/10.15407/pp2025.01.024>.
2. Semonov, B. O., & Pogorilyy, S. D. (2025). Application of transformer models for generating commit messages in version control systems. *Scientific papers of Donetsk National Technical University. Series: Informatics, Cybernetics and Computer Science*, 2(41), 17–29.
<https://doi.org/10.31474/1996-1588-2025-2-41-17-29>.
3. Семьонов, Б. О., & Погорілий, С. Д. (2025). Створення методу забезпечення якості коментарів у системах контролю версій на основі трансформерних моделей. *Реєстрація, Зберігання І Обробка Даних*, 27(2), 38–51.
<https://doi.org/10.35681/1560-9189.2025.27.2.345503>.

Наукові праці, які засвідчують апробацію матеріалів дисертації:

4. Pogorilyy, S. D., & Semonov, B. O. (2024). The implementation of a commit messages filter for software version control systems. *The 9th International Conference on Control and Optimization with Industrial Applications (COIA-2024)* (pp. 175–179). İstanbul, Türkiye.
5. Semonov, B. O., & Pogorilyy, S. D. (2025). Experimental investigation of CodeTrans T5 and T5 models for commit message generation in version control systems. *Proceedings of the XXI International Scientific Conference “Electronics and Applied Physics” (APHYS 2025)* (pp. 90–91). Kyiv, Ukraine.
6. Semonov, B. O., & Pogorilyy, S. D. (2025). Semantic enrichment of source code identifiers through automatic abbreviation expansion for commit

message generation. *Information Technology and Implementation (Satellite): Conference Proceedings* (pp. 298–301). Kyiv, Ukraine.

7. Pogorilyy, S. D., & Semonov, B. O. (2025). Language-specific fine-tuning of transformer models for commit message generation in version control systems. *XXV International Scientific and Practical Conference “Information Technologies and Security” (ITS-2025)*. Kyiv, Ukraine.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	17
ВСТУП.....	19
РОЗДІЛ 1. АНАЛІЗ МЕТОДІВ ФОРМУВАННЯ ТА ОЦІНЮВАННЯ ЯКОСТІ ПОВІДОМЛЕНЬ ПРО ВНЕСЕНІ ЗМІНИ У СИСТЕМАХ КОНТРОЛЮ ВЕРСІЙ	25
1.1 Роль повідомлень про внесені зміни у життєвому циклі програмного забезпечення	25
1.2 Існуючі підходи до автоматичного аналізу коментарів у системах контролю версій	28
1.2.1 Лексичний та текстовий аналіз.....	28
1.2.2 Статистичні методи	30
1.2.3 Методи машинного навчання та обробки природної мови (NLP).....	32
1.2.4 Гібридні підходи	34
1.3 Застосування методів машинного навчання для оцінки якості внесених змін	36
1.4 Методи автоматичної генерації повідомлень про внесені зміни	42
1.5 Метрики оцінювання якості повідомлень про внесені зміни.....	46
1.6 Висновки до розділу 1 та постановка задач дослідження	51
РОЗДІЛ 2. СТВОРЕННЯ МЕТОДІВ ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПОВІДОМЛЕНЬ ПРО ВНЕСЕНІ ЗМІНИ НА ОСНОВІ НЕЙРОННИХ МЕРЕЖ	54
2.1 Формування навчальної вибірки на основі GitHub REST API та попередня обробка даних	54
2.1.1 Джерело даних та вимоги до навчального корпусу	54
2.1.2 Критерії якості повідомлень про внесені зміни.....	55
2.1.3 Формалізація задачі класифікації.....	56

2.1.4 Методика розмітки даних	57
2.1.5 Попередня обробка текстів	58
2.1.6 Стратегія розподілу вибірки	58
2.1.7 Формат вхідних даних для моделей.....	58
2.2 Метод оцінювання якості повідомлень на основі рекурентних нейронних мереж	59
2.2.1 Архітектура рекурентних моделей для бінарної класифікації повідомлень	60
2.2.2 Навчання та регуляризація моделей	63
2.3 Роль моделей векторного представлення слів (Word2Vec, FastText, GloVe) у класифікації повідомлень про внесені зміни	66
2.3.1 Застосування векторних подань слів у класифікаторах MLP та RNN	68
2.4 Метод оцінювання якості повідомлень на основі трансформерних моделей (BERT, RoBERTa, DistilBERT)	70
2.4.1 Архітектура трансформерних класифікаторів	70
2.4.2 Донавчання попередньо навчених моделей	74
2.5 Застосування технологій GPGPU та TPU для прискорення навчання моделей	75
2.5.1 Архітектура паралельних обчислювальних платформ	75
2.5.2 Ефективність застосування паралельних обчислень для задачі класифікації	77
2.6 Висновки до розділу 2	79
РОЗДІЛ 3. РОЗРОБКА МЕТОДІВ АВТОМАТИЧНОЇ ГЕНЕРАЦІЇ ПОВІДОМЛЕНЬ ПРО ВНЕСЕНІ ЗМІНИ НА ОСНОВІ ТРАНСФОРМЕРНИХ МОДЕЛЕЙ	81

3.1 Формалізація задачі автоматичної генерації повідомлень як задачі умовної генерації тексту	81
3.2 Архітектура моделей генерації повідомлень (T5, FLAN-T5, CodeT5, CodeT5+, BART).....	85
3.2.1 Особливості архітектури encoder-decoder трансформерних моделей	85
3.2.2 Моделі, орієнтовані на вихідні тексти програм.....	88
3.3 Підходи до донавчання моделей	92
3.3.1 Інструкційне навчання (instruction tuning)	94
3.3.2 Мультизадачне навчання (multi-task learning)	96
3.4 Метод мовно-специфічного донавчання для систем контролю версій	98
3.4.1 Формування мовно-специфічних корпусів (C/C++, Ruby, Python, Java, JavaScript, PHP)	99
3.5 Метод семантичного збагачення ідентифікаторів вихідного коду через автоматичне розкриття абревіатур	101
3.5.1 Формалізація методу семантичного збагачення.....	102
3.5.2 Конвеєр автоматичного розкриття абревіатур в ідентифікаторах.....	103
3.6 Висновки до розділу 3	105
РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНИХ МЕТОДІВ	108
4.1 Опис експериментальних корпусів та середовища проведення досліджень.....	108
4.1.1 Корпус повідомлень для задачі класифікації (GitHub REST API).....	109
4.1.2 Корпус пар «змiна-повідомлення» для задачі генерації	114
4.1.3 Програмно-апаратне середовище експериментів	116
4.2 Експериментальне порівняння методів оцінювання якості повідомлень	119
4.2.1 Порівняння RNN-моделей за метриками Accuracy та F1-score	119

4.2.2 Вплив моделей векторного представлення слів на точність класифікації	122
4.2.3 Порівняння трансформерних класифікаторів	126
4.3 Експериментальне порівняння методів генерації повідомлень	128
4.3.1 Оцінювання за лексичними метриками	129
4.3.2 Оцінювання за семантичними метриками	133
4.3.3 Порівняльний аналіз базових та донавчених моделей	134
4.4 Аналіз впливу мовно-специфічного донавчання та семантичного збагачення на якість генерації	139
4.4.1 Вплив архітектури та стратегії попереднього навчання на якість генерації	140
4.4.2 Мовно-специфічні закономірності генерації повідомлень	141
4.4.3 Сукупний ефект донавчання та семантичного збагачення	143
4.5 Аналіз обчислювальної ефективності (CPU, GPGPU, TPU)	145
4.5.1 Порівняльний аналіз часу навчання моделей класифікації	146
4.5.2 Порівняльний аналіз часу навчання моделей генерації	147
4.5.3 Узагальнення результатів та рекомендації щодо вибору платформи	148
4.6 Висновки до розділу 4	149
ВИСНОВКИ	153
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	157
ДОДАТОК А. Оцінювання якості повідомлень на основі рекурентних нейронних мереж	167
ДОДАТОК Б. Оцінювання якості повідомлень з використанням попередньо навчених векторних представлень слів	170
ДОДАТОК В. Оцінювання якості повідомлень на основі трансформерних моделей	173

ДОДАТОК Г. Автоматична генерація повідомлень про внесені зміни на основі моделей кодувальник-декодувальник	177
ДОДАТОК Д. Автоматична розмітка навчальної вибірки	181
ДОДАТОК Е. Сценарії завантаження даних із GitHub REST API.....	184
ДОДАТОК Ж. Мобільний застосунок CmtSelector для розмітки навчальної вибірки	193
ДОДАТОК З. Сертифікат участі в міжнародній конференції COIA-2024 (м. Стамбул, Туреччина, 27–29 серпня 2024 р.).....	198
ДОДАТОК И. Сертифікат участі в міжнародній конференції ITS-2025 (м. Київ, Україна, 11 грудня 2025 р.)	199

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API	інтерфейс прикладного програмування (application programming interface)
AST	абстрактне синтаксичне дерево (abstract syntax tree)
BART	двонаправлений та авторегресійний трансформер (bidirectional and auto-regressive transformer)
BERT	двонаправлені кодувальні представлення з трансформерів (bidirectional encoder representations from transformers)
BiGRU	двонаправлений керований рекурентний блок (bidirectional gated recurrent unit)
BiLSTM	двонаправлена довга короткочасна пам'ять (bidirectional long short-term memory)
BLEU	метрика оцінювання якості машинного перекладу (bilingual evaluation understudy)
BLEURT	метрика оцінювання якості генерації на основі навчених представлень (BLEU with learned representations from transformers)
CodeBERT	попередньо навчена модель для мов програмування та природної мови
CodeT5	попередньо навчена модель типу кодувальник-декодувальник для вихідного коду
CodeT5+	розширена версія моделі CodeT5 з відкритим кодом
CPU	центральний процесор (central processing unit)
CUDA	уніфікована архітектура обчислень на графічних пристроях (compute unified device architecture)
DistilBERT	дистильована версія моделі BERT (distilled BERT)
FastText	метод векторного представлення слів із підслівними n-грамами
FLAN-T5	версія моделі T5, донавчена на колекції інструкцій FLAN

GloVe	глобальні вектори представлення слів (global vectors for word representation)
GPGPU	обчислення загального призначення на графічних процесорах (general-purpose computing on graphics processing units)
GPU	графічний процесор (graphics processing unit)
GRU	керований рекурентний блок (gated recurrent unit)
LDA	латентне розміщення Діріхле (latent Dirichlet allocation)
LSTM	довга короткочасна пам'ять (long short-term memory)
MLP	багатошаровий перцептрон (multilayer perceptron)
NLP	обробка природної мови (natural language processing)
RoBERTa	оптимізований підхід попереднього навчання BERT (robustly optimized BERT pretraining approach)
ROUGE	метрика оцінювання, зорієнтована на повноту (recall-oriented understudy for gisting evaluation)
SVM	метод опорних векторів (support vector machine)
T5	трансформер перетворення тексту в текст (text-to-text transfer transformer)
TF-IDF	частота терміна – обернена частота документа (term frequency – inverse document frequency)
TPU	тензорний процесор (tensor processing unit)
Word2Vec	метод векторного представлення слів (word to vector)
XLM-RoBERTa	багатомовна версія моделі RoBERTa (cross-lingual language model – RoBERTa)
ПЗ	програмне забезпечення

ВСТУП

Обґрунтування вибору теми дослідження. Сучасна індустрія розробки програмного забезпечення характеризується постійним зростанням обсягу та складності програмних систем, що обумовлює необхідність застосування ефективних засобів управління змінами у вихідних текстах програм. Системи контролю версій, зокрема Git, є невід’ємним інструментом колективної розробки програмного забезпечення, забезпечуючи збереження, відстеження та координацію модифікацій вихідного коду. Ключовим елементом процесу фіксації змін є повідомлення про внесені зміни (commit messages), які виконують роль текстового опису модифікацій та є основним джерелом документації еволюції програмного продукту [1–3].

Супровід програмного забезпечення є одним із найдовших та найбільш ресурсомістких етапів його життєвого циклу, сукупні витрати на якому можуть становити від 60 % до 80 % загальних витрат протягом усього періоду експлуатації програмного продукту [4]. Якість повідомлень про внесені зміни безпосередньо впливає на ефективність аналізу історії змін, локалізацію дефектів, рефакторинг та командну комунікацію. Водночас емпіричні дослідження свідчать, що значна частина повідомлень у реальних проєктах є надто короткими, шаблонними або семантично неінформативними, що суттєво знижує ефективність процесів супроводу [5, 6].

Застосування методів штучного інтелекту та машинного навчання для автоматизації процесів аналізу та генерації текстових описів є одним із пріоритетних напрямів сучасної комп’ютерної інженерії. Розвиток глибоких нейронних мереж, зокрема рекурентних архітектур (LSTM, GRU) та трансформерних моделей (BERT, RoBERTa, CodeT5), створив передумови для побудови ефективних методів автоматичної оцінки якості та генерації повідомлень про внесені зміни [7–10]. Проте існуючі підходи мають низку обмежень: недостатнє врахування специфіки мов програмування, обмежена здатність до аналізу семантики ідентифікаторів вихідного коду та відсутність комплексних методів, що поєднують оцінку якості та генерацію повідомлень [11, 12].

Актуальність дослідження підтверджується зростанням інтересу наукової спільноти до зазначеної проблематики, про що свідчать публікації на провідних міжнародних конференціях з програмної інженерії (ICSE, MSR, ASE, FSE) та в журналах IEEE Transactions on Software Engineering, Empirical Software Engineering [3, 13]. Питання автоматичної генерації commit-повідомлень та оцінки їхньої якості досліджуються науковцями провідних університетів та дослідницьких центрів світу. Зазначені чинники обґрунтовують необхідність розроблення нових методів проєктування і застосування нейронних мереж для створення засобів оцінювання якості та генерації повідомлень про внесені зміни в системах контролю версій, що визначає актуальність теми дисертаційного дослідження.

Мета і завдання дослідження. Метою дисертаційного дослідження є створення методів оцінки якості та автоматичної генерації повідомлень про внесені зміни у системах контролю версій на основі проєктування і застосування моделей нейронних мереж різної архітектури; дослідження ефективності рекурентних та трансформерних моделей для задачі бінарної класифікації повідомлень; розроблення методів мовно-специфічного донавчання та семантичного збагачення ідентифікаторів вихідного коду для підвищення якості генерації повідомлень.

Для досягнення поставленої мети сформовано наступні завдання:

1. Виконати комплексний аналіз існуючих методів формування та оцінювання якості повідомлень про внесені зміни у системах контролю версій, здійснити систематизацію методів автоматичної генерації повідомлень та метрик оцінювання їхньої якості.
2. Створити метод оцінювання якості повідомлень про внесені зміни на основі рекурентних нейронних мереж із дослідженням впливу попередньо навчених моделей векторного представлення слів на якість бінарної класифікації.
3. Створити метод оцінювання якості повідомлень про внесені зміни на основі трансформерних моделей із застосуванням методології

трансферного навчання та виконати порівняльний аналіз ефективності різних сімейств моделей.

4. Розробити метод автоматичної генерації повідомлень про внесені зміни на основі трансформерних моделей типу кодувальник-декодувальник із мовно-специфічним донавчанням для шести мов програмування.
5. Розробити метод семантичного збагачення ідентифікаторів вихідного коду через автоматичне розкриття абревіатур для підвищення якості автоматичної генерації повідомлень про внесені зміни.
6. Виконати експериментальну перевірку ефективності запропонованих методів та провести порівняльний аналіз застосування технологій паралельних обчислень (GPGPU та TPU) для прискорення навчання нейронних моделей.

Об’єктом дослідження є методи оцінки якості та автоматичної генерації повідомлень про внесені зміни у системах контролю версій на основі використання різних засобів машинного навчання.

Предметом дослідження є процес проєктування нейронних мереж різної архітектури (рекурентних, трансформерних, кодувальник-декодувальник) для оцінювання якості та генерації повідомлень про внесені зміни; методи мовно-специфічного донавчання та семантичного збагачення ідентифікаторів вихідного коду.

Методи дослідження. У дисертаційній роботі застосовуються методи машинного навчання, глибокого навчання, обробки природної мови, трансферного навчання; багатошарові нейронні мережі різної архітектури (рекурентні мережі LSTM та GRU, двонаправлені рекурентні мережі BiLSTM та BiGRU, трансформерні моделі BERT, RoBERTa, DistilBERT, XLM-RoBERTa, моделі типу кодувальник-декодувальник T5, FLAN-T5, CodeT5, CodeT5+, BART); методи попереднього навчання та донавчання мовних моделей; методи побудови та аналізу

абстрактних синтаксичних дерев. Для програмної реалізації методів та відповідних моделей використовується мова програмування Python із бібліотеками PyTorch, Transformers (Hugging Face), scikit-learn. Обчислювальні експерименти виконано із використанням платформи Google Colaboratory та технологій паралельних обчислень GPGPU (NVIDIA Tesla T4) і TPU.

Наукова новизна отриманих результатів. Протягом вирішення сформованих завдань дослідження отримано наступні результати:

1. Розроблено метод оцінювання якості повідомлень про внесені зміни у системах контролю версій на основі рекурентних нейронних мереж. Побудовано серію із семи моделей-кандидатів зростаючої складності: від базової моделі з повнозв'язним шаром до двохарових двонаправлених рекурентних мереж (BiLSTM, BiGRU). Показано доцільність застосування методу FastText для технічних текстів завдяки підтримці підслівних n-грам та ефективність трансферного навчання з попередньо навченими векторами (SO_vectors_200, Accuracy = 0,861).
2. Розроблено метод оцінювання якості повідомлень на основі трансформерних моделей із застосуванням методології трансферного навчання. Досліджено шістнадцять конфігурацій на базі чотирьох сімейств моделей – BERT, RoBERTa, DistilBERT та XLM-RoBERTa. Найвищу якість продемонструвала модель roberta_large_en (Accuracy = 0,876, F1-score = 0,719), яка перевершила найкращу RNN-модель на 1,5 відсоткових пункти за Accuracy та на 1,8 за F1-score.
3. Розроблено метод мовно-специфічного донавчання трансформерних моделей для систем контролю версій. Сформовано навчальні корпуси для шести мов програмування (C/C++, Ruby, Python, Java, JavaScript, PHP) загальним обсягом 271 504 навчальні пари. Показано, що моделі, орієнтовані на вихідні тексти програм (CodeT5, CodeT5+), суттєво

перевершують моделі загального призначення: середній BLEU для CodeT5+ становить 6,55 проти менше 1,0 для T5 та FLAN-T5.

4. Вперше запропоновано метод семантичного збагачення ідентифікаторів вихідного коду через автоматичне розкриття аббревіатур для підвищення якості автоматичної генерації повідомлень про внесені зміни. Розроблено п'ятиетапний конвеєр збагачення, що включає вилучення ідентифікаторів через побудову абстрактного синтаксичного дерева, сегментацію за конвенціями іменування, генерацію кандидатів розкриття на основі словникового підходу та контекстної моделі CodeBERT, семантичне ранжування кандидатів та формування збагачених текстових представлень. Підтверджено, що комбінація мовно-специфічного донавчання та семантичного збагачення забезпечує послідовне накопичення покращень: найкращі результати продемонструвала модель codet5p-770m (BLEU = 11,15, ROUGE-1 = 44,62).

Особистий внесок здобувача. Дисертаційна робота є результатом самостійних розробок автора. Усі роботи виконано у співавторстві з науковим керівником, здобувачу належать наступні результати: [56] – розроблення методу бінарної класифікації повідомлень про внесені зміни та його програмна реалізація у вигляді фільтра для систем контролю версій; [63] – створення комплексного методу забезпечення якості коментарів у системах контролю версій на основі трансформерних моделей, проєктування та реалізація архітектур нейронних мереж для класифікації та генерації повідомлень; [64] – дослідження та експериментальна перевірка ефективності застосування технологій GPGPU та TPU для прискорення навчання моделей оцінки якості повідомлень; [83] – розроблення методу мовно-специфічного донавчання трансформерних моделей для генерації повідомлень та формування навчальних корпусів для шести мов програмування; [84] – створення методу семантичного збагачення ідентифікаторів вихідного коду через автоматичне розкриття аббревіатур та його інтеграція у конвеєр генерації

повідомлень; [94] – проведення експериментального дослідження моделей CodeTrans T5 та T5 для генерації повідомлень про внесені зміни та порівняльний аналіз результатів; [95] – дослідження ефективності застосування трансформерних моделей типу кодувальник-декодувальник для автоматичної генерації повідомлень, проведення обчислювальних експериментів та аналіз результатів.

Апробація матеріалів дисертації. Основні теоретичні та практичні результати дисертаційної роботи доповідались на науково-технічних конференціях:

8. The 9th International Conference on Control and Optimization with Industrial Applications (м. Стамбул, Туреччина, 27–29 серпня 2024 р.).
9. XXI-й міжнародній науковій конференції «Електроніка та прикладна фізика» (м. Київ, 21–24 жовтня 2025 р.).
10. XII-й міжнародній конференції «Інформаційні технології та впровадження» (м. Київ, 21 листопада 2025 р.).
11. XXV-й міжнародній науково-практичній конференції «Інформаційні технології та безпека» (м. Київ, 11 грудня 2025 р.).

Публікації. За темою дисертаційної роботи опубліковано 7 наукових праць (3 статті та 4 тези): 3 статті – в фахових наукових виданнях України.

Структура та обсяг дисертаційної роботи. Дисертаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та 9 додатків. Загальний обсяг роботи складає 199 сторінок, з яких основний зміст викладено на 138 сторінках. Дисертація містить 14 рисунків, 26 таблиць та список використаних джерел зі 97 найменувань.

РОЗДІЛ 1. АНАЛІЗ МЕТОДІВ ФОРМУВАННЯ ТА ОЦІНЮВАННЯ ЯКОСТІ ПОВІДОМЛЕНЬ ПРО ВНЕСЕНІ ЗМІНИ У СИСТЕМАХ КОНТРОЛЮ ВЕРСІЙ

1.1 Роль повідомлень про внесені зміни у життєвому циклі програмного забезпечення

Упродовж життєвого циклу програмного забезпечення відбуваються постійні зміни, зумовлені як розвитком вимог користувачів, так і еволюцією технологічного середовища. Ефективне управління такими змінами є необхідною умовою забезпечення стабільності, надійності та якості програмного продукту на всіх етапах його експлуатації. Особливу роль у цьому процесі відіграють повідомлення про внесені зміни, які забезпечують збереження, аналіз і контроль модифікацій у вихідних текстах програм та супровідній документації.

Супровід програмного забезпечення (ПЗ) є одним із найдовших та найбільш ресурсомістких етапів його життєвого циклу. Згідно з класичними та сучасними моделями життєвого циклу ПЗ, сукупні витрати на етапі супроводу можуть становити від 60 % до 80 % загальних витрат протягом усього періоду експлуатації програмного продукту. До процесів супроводу належать коригувальний супровід (виправлення помилок), адаптивний супровід (пристосування до змін середовища та вимог), вдосконалювальний супровід (розширення та покращення функціональності) та профілактичний супровід, спрямований на зменшення майбутніх ризиків деградації якості програмного коду [4].

Формально сумарні витрати на супровід програмного забезпечення можна подати у вигляді:

$$C_{maint} = C_{corr} + C_{adapt} + C_{perf} + C_{prev}, \quad (1.1)$$

де C_{corr} – витрати на виправлення помилок;

C_{adapt} – витрати на адаптацію;

C_{perf} – витрати на вдосконалення функціональності;

C_{prev} – витрати на профілактичні заходи.

Якість документації та історії змін безпосередньо впливає на кожен з наведених складових.

Ключову роль у процесі супроводу відіграють системи контролю версій, які забезпечують збереження історії змін у вихідних текстах програм, підтримують паралельну розробку та дозволяють відстежувати еволюцію програмного забезпечення. Системи контролю версій, зокрема Git, є стандартним інструментом сучасної програмної інженерії та широко застосовуються як у промислових, так і в академічних проєктах [1].

Невід’ємним елементом кожної зафіксованої зміни є повідомлення про внесені зміни (commit message), яке слугує текстовим описом виконаних модифікацій. Commit-повідомлення забезпечують семантичний зв’язок між технічними змінами у вихідному коді та причинами їх внесення, виконуючи роль асинхронного комунікаційного механізму між розробниками та командами [2]. У контексті супроводу ПЗ описи внесених змін можуть розглядатися як частина неформальної документації, що накопичується протягом усього життєвого циклу системи.

Інформативність commit-повідомлень істотно впливає на ефективність аналізу історії проєкту. Зокрема, час, необхідний для розуміння причин певної модифікації, можна подати як функцію від якості повідомлень про внесені зміни:

$$T_{analysis} = f(Q_{msg})^{-1}, \quad (1.2)$$

де $T_{analysis}$ – середній час аналізу змін; Q_{msg} – показник якості опису внесених змін. Зі зростанням інформативності та структурованості повідомлень значення $T_{analysis}$ зменшується, що позитивно впливає на продуктивність супроводу.

У наукових дослідженнях якість повідомлень про внесені зміни часто оцінюється за низкою формальних метрик, зокрема довжиною повідомлення, наявністю дієслів дії, посилань на задачі та дефекти, а також відповідністю усталеним шаблонам. Основні функції commit-повідомлень та їх значення для процесу супроводу програмного забезпечення узагальнено в таблиці 1.1.

Таблиця 1.1 – Основні функції повідомлень про внесені зміни в системах контролю версій

Функція повідомлення про внесені зміни	Опис функції	Етап супроводу
Документування змін	Фіксація суті та призначення внесених змін	Аналіз історії
Комунікаційна	Передача контексту між розробниками	Командна розробка
Підтримка відлагодження	Полегшення пошуку причин помилок	Виправлення дефектів
Підтримка рефакторингу	Розуміння мотивації архітектурних змін	Еволюція коду
Автоматизована обробка	Використання в інструментах аналізу та генерації	Використання в інструментах аналізу та генерації

Узагальнений показник якості описів до внесених змін може бути поданий у вигляді зваженої суми окремих характеристик:

$$Q_{msg} = \sum_{i=1}^n w_i \cdot m_i, \quad (1.3)$$

де m_i – нормалізоване значення i -ї метрики якості (наприклад, довжина, семантична насиченість, структурованість); w_i – відповідний ваговий коефіцієнт, що відображає її значущість.

Водночас емпіричні дослідження показують, що значна частина повідомлень про внесені зміни в реальних проєктах є надто короткими, шаблонними або формальними, що ускладнює подальший супровід програмного забезпечення [14]. Такі описи часто не відповідають на запитання «чому» були внесені зміни, знижуючи цінність історії версій як джерела знань про еволюцію програмного продукту. Типові недоліки низькоякісних commit-повідомлень та їхній вплив на

процеси супроводу програмного забезпечення наведено в таблиці 1.2. Наслідком цього є зростання когнітивного навантаження на розробників та збільшення витрат часу на аналіз коду під час супроводу.

Таблиця 1.2 – Типові недоліки повідомлень про внесені зміни та їх вплив на супровід ПЗ

Тип повідомлення	Приклад	Наслідки для супроводу
Надто загальне	«fix», «update»	Втрата семантики
Опис без мотивації	«changed logic»	Без розуміння причин
Технічний шум	«minor changes»	Ускладнення аналізу
Непослідовне	Різні стилі	Неможливість автоматизації

З огляду на зазначене, актуальним є завдання автоматизованого аналізу та забезпечення якості повідомлень про внесені зміни, а також розроблення методів їх автоматичної генерації на основі аналізу змін у вихідних текстах програм. Застосування методів машинного навчання та обробки природної мови дає змогу формалізувати залежність між характеристиками змін у коді та якістю супровідної текстової інформації, що відкриває нові можливості для підвищення ефективності супроводу програмного забезпечення в цілому.

1.2 Існуючі підходи до автоматичного аналізу коментарів у системах контролю версій

Серед існуючих методів автоматичного аналізу описів до внесених змін можна виділити кілька основних підходів, що застосовуються в наукових дослідженнях та практичних інструментах.

1.2.1 Лексичний та текстовий аналіз

Лексичний та текстовий аналіз залишається одним із базових і водночас поширених підходів до автоматичного опрацювання коментарів у системах контролю версій [3]. Основна ідея методу полягає у виявленні ключових слів, словоформ або регулярних виразів у текстах описів внесених змін, що дає змогу

здійснювати попередню класифікацію модифікацій у вихідних текстах програм [13]. Зокрема, коментарі, які містять лексеми «fix», «bug» або «patch», зазвичай інтерпретуються як виправлення помилок, тоді як використання слів «refactor», «optimize» чи «restructure» пов'язується з рефакторингом або реструктуризацією програмного коду [15].

У низці сучасних реалізацій додатково застосовуються регулярні вирази для виявлення складніших шаблонів у коментарях, зокрема посилань на системи відстеження помилок, ідентифікаторів задач та типів змін [5].

Перевагами лексичного підходу є відносна простота реалізації та низькі обчислювальні витрати, що забезпечує його масштабованість під час аналізу великих репозиторіїв програмного коду [16]. Для оцінки ефективності лексичного аналізу використовують метрики точності:

$$P = \frac{TP}{TP + FP}, R = \frac{TP}{TP + FN}, F_1 = 2 \cdot \frac{P \cdot R}{P + R}, \quad (1.4)$$

де TP – кількість правильно класифікованих коментарів; FP – кількість помилково класифікованих; FN – пропущені коментарі.

Водночас численні дослідження відзначають обмеження таких методів, зокрема зниження точності у випадку неформалізованих коментарів, ігнорування семантичного контексту та чутливість до мовних і стилістичних варіацій [17].

Для подолання цих обмежень сучасні роботи пропонують комбінування лексичних правил зі статистичними методами та засобами обробки природної мови. Загальна ймовірність віднесення коментаря c_i до категорії u у комбінованому підході описується так:

$$P(y|c_i) = \alpha \cdot P_{lex}(y|c_i) + (1-\alpha) \cdot P_{stat}(y|c_i), \alpha \in [0,1], \quad (1.5)$$

де $P_{lex}(y|c_i)$ – ймовірність, оцінена за ключовими словами; $P_{stat}(y|c_i)$ – ймовірність, оцінена статистично (наприклад, через ML-модель); α – ваговий коефіцієнт між лексичними та статистичними ознаками.

Таким чином, лексичний та текстовий аналіз може виступати як ефективний початковий інструмент автоматичного опрацювання коментарів, особливо у

сценаріях, де потрібна швидка первинна класифікація змін або підготовка даних для використання більш складних методів аналізу.

1.2.2 Статистичні методи

Статистичний підхід до автоматичного аналізу коментарів у системах контролю версій ґрунтується на застосуванні частотних та ймовірнісних моделей для виявлення закономірностей у текстах описів до внесених змін. До таких методів належать n-грам аналіз, TF-IDF (Term Frequency-Inverse Document Frequency), кластеризація та тематичне моделювання (LDA, Latent Dirichlet Allocation) [18].

Аналіз n-грам передбачає розбиття тексту на послідовності з n слів, що дозволяє оцінити ймовірність появи певних словосполучень та виявляти характерні шаблони для конкретних типів змін у коді [19]. Ймовірність послідовності слів у n-грамній мовній моделі визначається як:

$$P(w_1, w_2, \dots, w_m) \approx \prod_{i=1}^m P(w_i | w_{i-n+1}, \dots, w_{i-1}), \quad (1.6)$$

де w_i – i -те слово тексту; n – розмір n-грамми.

Для біграмної моделі умовна ймовірність оцінюється через частоти:

$$P(w_i | w_{i-1}) = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})}, \quad (1.7)$$

де C – кількість появ відповідної послідовності у корпусі текстів.

Наприклад, часті комбінації «fix bug», «add feature», «update test» можуть слугувати індикаторами певних типів коментарів внесених змін [20, 21].

Метод TF-IDF використовується для визначення значущості окремих термінів у межах всього репозиторію [22]. Він дозволяє виділяти слова та словосполучення, що є специфічними для певного типу змін, і зменшує вплив часто вживаних, але малозначущих слів (наприклад, «update», «change»). Частота терміна у документі визначається як:

$$TF(t, d) = \frac{f_{t,d}}{\sum_k f_{k,d}}, \quad (1.8)$$

де $f_{t,d}$ – кількість входжень терміна t у документі d .

Обернена частота документа визначається формулою:

$$IDF(t) = \log \frac{N}{df_t}, \quad (1.9)$$

де N – загальна кількість документів; df_t – кількість документів, що містять термін t .

Підсумкова вага терміна обчислюється як:

$$TFIDF(t, d) = TF(t, d) \cdot IDF(t), \quad (1.10)$$

Це підвищує ефективність класифікації коментарів на основі лексичного складу тексту.

Кластеризація та тематичне моделювання дозволяють групувати коментарі за схожістю змісту без попередньої розмітки даних. У задачах кластеризації текстів широко застосовується косинусна міра подібності між векторними представленнями документів:

$$sim(A, B) = \frac{A \cdot B}{\|A\| \|B\|}, \quad (1.11)$$

де A, B – вектори ознак документів (наприклад, TF-IDF-представлення).

LDA, зокрема, виділяє приховані теми у текстах описів внесених змін, що може допомогти автоматично ідентифікувати типи змін: виправлення помилок, нові функції, рефакторинг, оновлення документації тощо [23]. У моделі LDA документ розглядається як суміш тем, а кожна тема – як розподіл слів. Ймовірність документа задається як:

$$P(d) = \prod_{n=1}^N \sum_{k=1}^K P(w_n | z_n = k) \cdot P(z_n = k | d), \quad (1.12)$$

де z_n – тема слова; K – кількість тем.

Апріорні розподіли параметрів задаються розподілом Діріхле:

$$\theta \sim Dir(\alpha), \quad \phi \sim Dir(\beta). \quad (1.13)$$

Перевагою статистичного підходу є можливість аналізувати великі масиви коментарів і виявляти загальні закономірності, що відображають тенденції розвитку проєкту та частоту різних типів змін [24]. Статистичні методи також дозволяють підготувати дані для подальшого застосування більш складних

моделей машинного навчання та NLP, що підвищує загальну ефективність автоматизованого аналізу.

Водночас такі підходи мають важливі обмеження. Вони здебільшого ігнорують семантичний та контекстуальний зміст коментарів, що обмежує здатність моделі розрізняти схожі за лексикою, але різні за значенням коментарі [19]. Наприклад, фрази «fix login bug» та «fix documentation for login» можуть мати схожий лексичний склад, але стосуються різних типів змін. Крім того, статистичні методи чутливі до стилю написання та мови коментарів, що ускладнює їх застосування у мультинаціональних командах або при роботі з історією проєкту, яка містить коментарі на різних мовах.

Для підвищення точності класифікації в практичних системах часто поєднують статистичні методи з лексичними правилами та алгоритмами машинного навчання, створюючи гібридні підходи [18]. У загальному вигляді задача класифікації може бути подана через формулу Байєса:

$$P(c | d) = \frac{P(d | c)P(c)}{P(d)}, \quad (1.14)$$

де c – клас коментаря; d – текст коментаря.

Такий підхід дозволяє враховувати як частотні закономірності, так і семантичний контекст текстів.

1.2.3 Методи машинного навчання та обробки природної мови (NLP)

Методи машинного навчання (ML) та обробки природної мови (Natural Language Processing, NLP) дозволяють здійснювати глибокий і семантично точний аналіз коментарів у системах контролю версій. На відміну від лексичних або статистичних підходів, ML/NLP моделі здатні враховувати контекст, синтаксис та семантичне значення тексту, що особливо важливо під час класифікації коментарів, які містять багатозначні або складні фрази [25].

У рамках супервізованого навчання широко застосовуються такі алгоритми:

- SVM (Support Vector Machine) – підходить для класифікації текстів за чітко визначеними ознаками, наприклад, ключовими словами та частотними векторами;

- Random Forest – ансамблевий метод, який дозволяє враховувати складні взаємозв'язки між ознаками та підвищує стабільність класифікації на невеликих і середніх наборах даних;
- нейронні мережі (Feedforward, LSTM, GRU) – забезпечують здатність моделі виявляти залежності у послідовності слів і розпізнавати складні синтаксичні структури, що важливо для довгих або багатослівних коментарів [26].

Упродовж останніх років значну популярність здобули трансформери, зокрема BERT (Bidirectional Encoder Representations from Transformers) та його похідні, які дозволяють здійснювати контекстно залежне розуміння тексту [8]. Такі моделі враховують не лише окремі слова, а й їх взаємозв'язки в межах речень і документів, що дає змогу ефективно класифікувати коментарі, виявляти синоніми, складні конструкції та специфічні шаблони мови розробників.

Сучасні дослідження показують, що трансформерні моделі можуть ефективно застосовуватися і до аналізу програмного коду та описів змін у репозиторіях, формуючи спільні представлення природної мови та вихідних текстів програм [27].

Використання ML/NLP методів дозволяє:

- автоматично класифікувати коментарі за типом змін (виправлення помилок, додавання функцій, рефакторинг, оновлення документації);
- прогнозувати технічний борг та потенційні ризики регресій;
- аналізувати тренди розвитку проєкту на основі семантичного змісту описів внесених змін;
- інтегрувати автоматичні рекомендації щодо стилю та змісту коментарів у CI/CD процесах [28].

Незважаючи на значні переваги, застосування ML/NLP підходів має певні обмеження. Основною вимогою є наявність великого обсягу розмічених даних, необхідних для навчання супервізованих моделей. Для проєктів з невеликою історією внесених змін або нестандартним стилем коментарів може знадобитися ручна розмітка даних, що збільшує витрати часу та ресурсів. Крім того,

трансформери та глибокі нейронні мережі потребують значних обчислювальних ресурсів для навчання та інференсу, що обмежує їх застосування у невеликих командах або при роботі з локальними серверами [7].

Для подолання цих обмежень у практиці часто застосовують гібридні підходи, поєднуючи ML/NLP моделі з лексичними правилами та статистичними методами. Така комбінація дозволяє забезпечити високу точність класифікації, зменшити вплив неформалізованих або неконсистентних коментарів та підвищити ефективність аналізу історії проєкту без необхідності використання надмірних обчислювальних ресурсів.

У підсумку, методи машинного навчання та NLP є ключовим інструментом сучасного автоматичного аналізу коментарів, особливо для великих програмних проєктів із значною історією внесених змін та різноманітними стилями ведення документації, і створюють основу для подальших досліджень та впровадження автоматизованих систем підтримки розробки та супроводу програмного забезпечення [27].

1.2.4 Гібридні підходи

Найбільш ефективними у сучасних дослідженнях та практичних застосуваннях часто є гібридні системи, що поєднують лексичні правила, статистичні методи та моделі машинного навчання. Такі підходи дозволяють одночасно здійснювати автоматичне маркування коментарів та перевірку їх на відповідність стандартам оформлення повідомлень про внесені зміни, наприклад, щодо структури, наявності ключових слів, або згадок пов'язаних завдань у системах відстеження помилок [29]. Комбінація різних стратегій дає змогу поєднати швидкість та простоту лексичних методів з гнучкістю і семантичною точністю моделей ML/NLP, що значно підвищує ефективність класифікації коментарів [28]. Основні підходи до автоматичного аналізу коментарів наведено у таблиці 1.3.

Серед практичних інструментів, що застосовуються для автоматичного аналізу коментарів, можна виділити кілька категорій:

- стандартизація коментарів: наприклад, Commitizen [30], який дозволяє формалізувати повідомлення про внесені зміни відповідно до заздалегідь визначених шаблонів та рекомендацій, забезпечуючи єдину структуру даних для подальшого аналізу;
- збір та попередня обробка даних: інструменти на кшталт PyDriller [31] та GitMiner [32] надають можливість автоматично збирати історію комітів з репозиторіїв Git, витягувати метадані та тексти коментарів, що необхідно для статистичного аналізу та навчання моделей ML;
- аналіз змісту та класифікація: системи, що інтегрують NLP-моделі, дозволяють класифікувати коментарі за типом змін (виправлення помилок, додавання функцій, рефакторинг, оновлення документації) та оцінювати якість написання повідомлень [27].

Таблиця 1.3 – Підходи до автоматичного аналізу коментарів до внесених змін

Підхід	Основна ідея	Переваги	Недоліки
Лексичний аналіз	Пошук ключових слів і шаблонів	Простота, швидкість	Низька семантична точність
Статистичні методи	Аналіз частот і розподілів слів	Масштабованість	Потреба у попередній обробці
ML/NLP	Класифікація на основі моделей	Висока точність	Потреба у розмічених даних
Гібридний підхід	Комбінація методів	Баланс точності та швидкості	Складність реалізації

Порівняльний аналіз таких інструментів показує, що комбінація статистичних методів і NLP-підходів забезпечує оптимальну точність та гнучкість класифікації, проте вимагає достатньої кількості розмічених даних для навчання моделей та значних обчислювальних ресурсів. Це робить їх особливо придатними

для великих проєктів із довгою історією внесених змін та великими командами розробників [33].

Основними проблемами автоматичного аналізу коментарів залишаються:

- нерегулярність написання коментарів, включно зі скороченнями, помилками та жаргонними виразами [34];
- різноманітність стилів та мовних форм, що ускладнює класифікацію у мультинаціональних командах [34];
- необхідність врахування контексту вихідних текстів програм, оскільки зміст коментаря часто не може бути однозначно визначений без аналізу змінених файлів або пов'язаних задач [27].

Незважаючи на ці обмеження, існуючі гібридні підходи дозволяють отримувати цінні аналітичні дані щодо історії змін, тенденцій розвитку проєкту та поведінки команди розробників. Вони також сприяють удосконаленню процесів супроводу програмного забезпечення, допомагаючи своєчасно виявляти проблеми, планувати оновлення та контролювати якість вихідних текстів програм [35].

Таким чином, для ефективного використання коментарів у системах контролю версій доцільно застосовувати комбіновані стратегії, що інтегрують лексичні правила, статистичні методи та сучасні технології машинного навчання. Такий підхід забезпечує баланс між точністю аналізу, практичною застосовністю та обчислювальною ефективністю, створюючи надійну основу для подальших досліджень та розробки автоматизованих систем аналізу змін у вихідних текстах програм, а також для інтеграції аналітичних інструментів у процеси безперервної інтеграції та супроводу програмних продуктів [36].

1.3 Застосування методів машинного навчання для оцінки якості внесених змін

У сучасних процесах розроблення програмного забезпечення системи контролю версій відіграють ключову роль у забезпеченні відстежуваності змін, командної взаємодії та підтримки життєвого циклу програмних продуктів. Одним із важливих елементів таких систем є повідомлення про внесені зміни (commit messages), які містять текстовий опис виконаних модифікацій вихідних текстів

програм. Якість цих повідомлень безпосередньо впливає на ефективність супроводу програмного забезпечення, спрощення аналізу історії змін, локалізацію помилок та повторне використання знань про розвиток проєкту [6].

На практиці значна частина повідомлень формується розробниками без суворих стандартів або автоматичних перевірок, що призводить до появи коротких, неінформативних або семантично некоректних описів. У зв'язку з цим виникає задача автоматизованої оцінки якості повідомлень про внесені зміни. Одним із найбільш ефективних підходів до її розв'язання є використання методів машинного навчання, які дозволяють виявляти приховані закономірності між текстом повідомлення та фактичним змістом змін у вихідних текстах [37].

Методи машинного навчання забезпечують можливість автоматичної класифікації commit-повідомлень на основі набору ознак, сформованих як із текстової інформації, так і зі структурних характеристик змін. Такий підхід дозволяє підвищити об'єктивність оцінювання та мінімізувати вплив суб'єктивного фактору [12].

Одним із ключових аспектів оцінювання якості повідомлень про внесені зміни є його семантична обґрунтованість, тобто відповідність текстового опису фактичним змінам у програмному коді. Семантично обґрунтоване повідомлення повинно відображати:

- тип виконаних змін (виправлення помилки, додавання функціональності, рефакторинг тощо);
- об'єкти змін (класи, модулі, функції);
- характер модифікацій;
- контекст виконаних дій.

Формалізація цього критерію передбачає представлення тексту повідомлення та змін у коді у вигляді векторних моделей ознак [38].

Нехай T – текст повідомлення про внесені зміни, C – множина змін у коді (diff). Векторні представлення визначається як:

$$v_T = \varphi(T), \quad v_C = \psi(C), \quad (1.15)$$

де $\varphi(T)$ – функція перетворення тексту у вектор ознак; $\psi(C)$ – функція перетворення змін у вихідних текстах у вектор ознак.

Семантичну обґрунтованість можна описати через міру подібності між векторами:

$$Q = \text{sim}(v_T, v_C). \quad (1.16)$$

Найчастіше використовується косинусна подібність:

$$\text{sim}(v_T, v_C) = \frac{v_T \cdot v_C}{||v_T|| \cdot ||v_C||}. \quad (1.17)$$

Отримане значення характеризує рівень узгодженості тексту повідомлення з фактичними змінами [38].

Однак лише семантичної подібності недостатньо для повної оцінки якості. Доцільно використовувати комбінований критерій, що враховує декілька характеристик:

$$Q = \alpha S + \beta L + \gamma K, \quad (1.18)$$

де S – семантична подібність; L – нормалізована довжина повідомлення; K – показник наявності ключових слів; α, β, γ – вагові коефіцієнти [37].

Нормалізовану довжину повідомлення можна визначити як:

$$L = \frac{|T|}{|T|_{\max}}, \quad (1.19)$$

де $|T|$ – кількість токенів у повідомленні.

Показник ключових слів може визначатися як:

$$K = \frac{n_{\text{key}}}{n}, \quad (1.20)$$

де n_{key} – кількість ключових слів; n – загальна кількість слів у повідомленні.

До ключових слів зазвичай належать дієслова дії, які характеризують тип змін (add, fix, update, remove, refactor тощо).

Таким чином, семантична обґрунтованість повідомлення може розглядатися як інтегральна характеристика, що враховує текстову інформативність та відповідність структурі змін у коді [38].

Задачу автоматичної оцінки якості описів до внесених змін доцільно формалізувати як задачу бінарної класифікації [37].

Нехай задано навчальну вибірку:

$$D = \{(x_i, y_i)\}_{i=1}^N, \quad (1.21)$$

де x_i – вектор ознак i -го повідомлення про внесені зміни; $y_i \in \{0,1\}$ – мітка класу: $y = 1$ – повідомлення семантично узгоджене, $y = 0$ – повідомлення семантично неузгоджене.

Вектор ознак може включати:

- текстові характеристики повідомлення;
- статистику змін у вихідних текстах;
- структурні параметри опису внесених змін.

Необхідно побудувати класифікатор:

$$f(x): X \rightarrow Y, \quad (1.22)$$

який мінімізує функцію втрат:

$$L = \frac{1}{N} \sum_{i=1}^N \ell(f(x_i), y_i). \quad (1.23)$$

У задачах бінарної класифікації часто використовується логістична функція:

$$P(y = 1|x) = \frac{1}{1 + e^{-w^T x}}, \quad (1.24)$$

де w – вектор параметрів моделі.

Класифікаційне рішення визначається за пороговим значенням:

$$\hat{y} = \begin{cases} 1, & P \geq \tau \\ 0, & P < \tau \end{cases} \quad (1.25)$$

де τ – поріг класифікації.

Ефективність застосування методів машинного навчання значною мірою залежить від якості сформованого набору ознак. Для задачі оцінювання повідомлень про внесені зміни доцільно використовувати комбінований підхід, що включає текстові, структурні та статистичні характеристики [37].

До текстових ознак належать:

- довжина повідомлення;
- кількість слів;
- кількість унікальних токенів;

- частоти термінів;
- TF-IDF представлення;
- частини мови.

Одним із найбільш поширених методів є TF-IDF перетворення:

$$tfidf(t, d) = tf(t, d) \cdot \log \frac{N}{df(t)}, \quad (1.26)$$

де $tf(t, d)$ – частота терміна t у документі d ; $df(t)$ – кількість документів, що містять термін; N – кількість документів.

TF-IDF дозволяє зменшити вплив загальних слів та підсилити вагу інформативних термінів.

Ознаки змін у вихідних текстах програм формуються на основі diff-інформації:

- кількість змінених файлів;
- кількість доданих рядків;
- кількість видалених рядків;
- загальний розмір збережених змін.

Розмір збережених змін можна описати як:

$$C_{size} = add + del, \quad (1.27)$$

де add – кількість доданих рядків; del – кількість видалених рядків.

Також доцільно використовувати співвідношення змін:

$$R = \frac{add}{add + del}, \quad (1.28)$$

Цей показник дозволяє частково характеризувати тип змін.

До структурних ознак можуть належати:

- кількість внесених змін автором;
- часові характеристики;
- частота змін;
- кількість модифікованих модулів.

Такі ознаки дозволяють врахувати контекст розробки програмного забезпечення.

Для розв'язання задачі оцінювання якості commit-повідомлень можуть застосовуватися різні класи моделей машинного навчання. Найпростішими є лінійні моделі, зокрема логістична регресія, які забезпечують швидке навчання та інтерпретованість результатів. Метод опорних векторів дозволяє ефективно працювати з високорозмірними ознаковими просторами, що характерно для текстових даних.

Дерева рішень та ансамблеві методи забезпечують можливість врахування нелінійних залежностей між ознаками. Особливо ефективними є випадкові ліси та градієнтний підсилення [37].

Останніми роками значного поширення набули нейронні мережі, які дозволяють автоматично формувати ознаки на основі текстових представлень. Використання векторних представлень слів та контекстних мовних моделей (зокрема попередньо навчених моделей для коду й природної мови) дає змогу враховувати семантичні зв'язки між словами та структурами коду [39].

У задачах аналізу commit-повідомлень доцільно використовувати гібридні підходи, що поєднують:

- статистичні ознаки;
- векторні представлення тексту (текстові embeddings);
- структурні характеристики, а також (за потреби) історичний контекст репозиторію [40].

Такий підхід забезпечує підвищення точності класифікації.

Отже, застосування методів машинного навчання дозволяє автоматизувати процес оцінювання якості повідомлень про внесені зміни та підвищити ефективність супроводу програмного забезпечення. Запропонований підхід базується на формуванні комплексного ознакового простору, що включає текстові, структурні та статистичні характеристики внесених змін [6].

Формалізація задачі у вигляді бінарної класифікації створює можливість використання широкого спектра алгоритмів машинного навчання та стандартних метрик оцінювання. Важливим етапом є визначення критерію семантичної

обґрунтованості повідомлення, який дозволяє встановити відповідність між текстовим описом та фактичними змінами у коді [40].

Отримані теоретичні положення є основою для подальшого розроблення моделей автоматичного аналізу повідомлень про внесені зміни та проведення експериментальних досліджень їх ефективності.

1.4 Методи автоматичної генерації повідомлень про внесені зміни

Задача автоматичної генерації повідомлень про внесені зміни формально визначається як задача умовної генерації тексту. Нехай D – диференційне представлення змін у вихідному коді (diff), а $M = (m_1, m_2, \dots, m_T)$ – цільове повідомлення довжиною T токенів. Задача полягає у знаходженні повідомлення $\hat{M}$, що максимізує умовну ймовірність:

$$\hat{M} = \operatorname{argmax}_M P(M | D) = \operatorname{argmax}_M \prod_{t=1}^T P(m_t | m_1, \dots, m_{t-1}, D). \quad (1.29)$$

До вхідних даних належать: текстовий опис внесених змін у вихідних текстах, службова інформація проєкту, супровідний контекст (пов'язані завдання, коментарі розробників) та історія попередніх повідомлень. Якість отриманих результатів оцінюється як за допомогою автоматичних метрик – BLEU, METEOR, ROUGE, BERTScore – так і за результатами людської оцінки.

Евристичні та шаблонні методи – перші підходи, які ґрунтувалися на детерміністичних правилах: синтаксичний аналіз внесених змін дозволяв визначити тип зміни (додавання, видалення, модифікація файлу) і підставити відповідний шаблон. Перевагами є детермінованість і відсутність потреби у навчальних даних, проте методи не здатні відображати семантику змін і продукують одноманітні, малоінформативні повідомлення. Дослідження [41] показали, що подібні підходи значно поступаються нейромережевим за всіма метриками якості.

Методи на основі витягування інформації – це підходи, які передбачають пошук у корпусі існуючих повідомлень таких, що семантично або структурно найближчі до поточної зміни у вихідних текстах. Внесені зміни D та кожне

повідомлення з корпусу $\{M_i\}$ векторизуються за допомогою TF-IDF: вага терму t у документі d визначається як:

$$\text{TF-IDF}(t, d) = \frac{f_{t,d}}{\sum_{t'} f_{t',d}} \cdot \log \frac{N}{|\{d': t \in d'\}|}, \quad (1.30)$$

де $f_{t,d}$ – частота терму t у документі d ; N – загальна кількість документів у корпусі. Пошук найближчого сусіда здійснюється за косинусною подібністю:

$$\text{sim}(D, M_i) = \frac{\mathbf{v}_D \cdot \mathbf{v}_{M_i}}{\|\mathbf{v}_D\| \cdot \|\mathbf{v}_{M_i}\|}, \quad (1.31)$$

після чого знайдене повідомлення адаптується за допомогою правил підстановки або реранжується на основі структурних ознак AST [42]. Перевагою є природність результатів, основним недоліком – залежність від якості та розміру корпусу.

Методи на основі рекурентних нейронних мереж: архітектура encoder-decoder на базі LSTM з механізмом уваги дозволила формалізувати задачу як задачу послідовність-у-послідовність. Кодувальник обробляє токенизований текст внесених змін $D = (x_1, \dots, x_n)$ і обчислює приховані стани $h_i = \text{LSTM}(x_i, h_{i-1})$. На кожному кроці декодування вага уваги для i -го токена входу визначається як [11]:

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{j=1}^n \exp(e_{tj})}, \quad e_{ti} = \mathbf{v}^\top \tanh(\mathbf{W}_h h_i + \mathbf{W}_s s_t), \quad (1.32)$$

де s_t – прихований стан декодувальника на кроці t ; $\mathbf{W}_h$, $\mathbf{W}_s$, $\mathbf{v}$ – навчальні параметри. Контекстний вектор $c_t = \sum_i \alpha_{ti} h_i$ використовується для генерації токена m_t . Механізм копіювання додатково дозволяє з імовірністю $p_{\text{gen}} \in [0,1]$ генерувати токен зі словника або безпосередньо копіювати його з вхідної послідовності:

$$P(m_t) = p_{\text{gen}} \cdot P_{\text{vocab}}(m_t) + (1 - p_{\text{gen}}) \sum_{i: x_i=m_t} \alpha_{ti}. \quad (1.33)$$

Обмеженням є складність моделювання довгострокових залежностей і чутливість до довжини вхідної послідовності.

Трансформерні моделі та методи попереднього навчання – це підходи, в основі яких лежить механізм масштабованої скалярної уваги (scaled dot-product attention):

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V, \quad (1.34)$$

де $Q \in \mathbb{R}^{n \times d_k}$, $K \in \mathbb{R}^{m \times d_k}$, $V \in \mathbb{R}^{m \times d_v}$ – матриці запитів, ключів і значень відповідно; d_k – розмірність ключів [7]. Механізм самоуваги (self-attention) дозволяє ефективно моделювати залежності між будь-якими двома позиціями послідовності незалежно від їхньої відстані. Донавчання (fine-tuning) попередньо навченої моделі – T5, BART або CodeT5 – на корпусі пар «внесені зміни (diff) – повідомлення» здійснюється мінімізацією функції перехресної ентропії:

$$\mathcal{L}_{\text{CE}} = - \sum_{t=1}^T \log P_{\theta}(m_t \mid m_{<t}, D), \quad (1.35)$$

де θ – параметри моделі. Дослідження [43] показало, що донавчання T5 на понад 2 мільйони commit-пар забезпечує значне перевищення рекурентних базових ліній. Моделі, орієнтовані на вихідні тексти програм (CodeBERT, GraphCodeBERT, CodeT5), демонструють додаткове покращення показника BERTScore на 4–7% завдяки кращому розумінню семантики програмного коду [38]. Окремим підходом є використання великих мовних моделей без донавчання: задача формулюється як текстовий запит із кількома прикладами у контексті (few-shot prompting), що забезпечує гнучкість адаптації під різні конвенції без необхідності збору тренувальних даних, але пов'язане з проблемами конфіденційності та обчислювальних витрат.

Методи з використанням структурної інформації вихідних текстів аналізують зміни на рівні синтаксичних вузлів абстрактного синтаксичного дерева (AST) замість обробки текстів внесених змін як послідовності рядків. Внесені зміни представляється як граф змін $G = (V, E)$, де вузли V відповідають синтаксичним конструкціям (доданим, видаленим або модифікованим), а ребра E – структурним залежностям між ними. Графова нейронна мережа агрегує ознаки вузлів за схемою передачі повідомлень [44]:

$$h_v^{(l+1)} = \sigma\left(W^{(l)} \cdot \text{AGG}\left(\{h_u^{(l)} : u \in \mathcal{N}(v)\} \cup \{h_v^{(l)}\}\right)\right) \quad (1.36)$$

де $\mathcal{N}(v)$ – множина сусідів вузла v ; $W^{(l)}$ – матриця ваг шару l ; σ – функція активації. Трансформерний декодувальник генерує повідомлення на основі отриманого представлення графа. Підхід демонструє вищу ефективність для складних рефакторингів і змін, що стосуються кілька файлів, проте побудова диференційного AST є обчислювально затратною і залежить від синтаксичного аналізатора конкретної мови програмування.

Методи з підкріплювальним навчанням – традиційне навчання за функцією $\mathcal{L}_{\text{CE}}$, яке не оптимізує цільові метрики якості безпосередньо. У рамках алгоритму SCST (Self-Critical Sequence Training) градієнт функції нагороди обчислюється як [45]:

$$\nabla_{\theta} \mathcal{L}_{\text{RL}} = -\mathbb{E}_{M^s \sim P_{\theta}} \left[\left(r(M^s) - r(\hat{M}) \right) \nabla_{\theta} \log P_{\theta}(M^s | D) \right], \quad (1.37)$$

де M^s – повідомлення, згенероване стохастичним вибором; $\hat{M}$ – базове повідомлення (greedy decoding); $r(\cdot)$ – функція нагороди (наприклад, BLEU або BERTScore). Підкріплювальне навчання у поєднанні з попереднім навчанням з учителем стабільно покращує результати порівняно з виключно навчанням з учителем.

Узагальнений порівняльний аналіз розглянутих підходів наведено в таблиці 1.4.

Таблиця 1.4 – Порівняння методів автоматичної генерації повідомлень про внесені зміни

Метод	Потреба в даних	Якість	Узагальнення	Ефективність
Евристичні шаблони	Відсутня	Низька	Низька	Висока
Витягування інформації	Велика	Середня	Середня	Середня
RNN seq2seq	Велика	Середня–висока	Середня	Середня

Метод	Потреба в даних	Якість	Узагальнення	Ефективність
Трансформер + fine-tuning	Велика	Висока	Висока	Середня
LLM + few-shot prompting	Мала/відсутня	Висока	Дуже висока	Низька
GNN + AST	Велика	Висока	Висока	Низька
RL-оптимізація	Велика	Дуже висока	Висока	Низька

Аналіз показує, що методи на основі донавчання трансформерних моделей, орієнтованих на вихідні тексти програм, забезпечують найкращий баланс між якістю та практичністю застосування. Для промислового використання оптимальним є гібридний підхід: донавчання CodeT5 або аналогічної моделі на репозиторії-специфічних даних із підтримкою механізму копіювання та структурного аналізу AST.

1.5 Метрики оцінювання якості повідомлень про внесені зміни

Оцінювання якості автоматично згенерованих повідомлень про зміни є невід’ємною складовою розробки та порівняння відповідних методів. Оскільки задача належить до класу задач генерації природномовного тексту, для її оцінювання використовують як автоматичні метрики, що порівнюють згенерований текст з еталонним, так і методи експертної оцінки, що залучають людей-оцінювачів. У цьому підрозділі систематизовано основні метрики, описано їхні математичні засади та проаналізовано придатність для оцінювання повідомлень про зміни у вихідному коді.

Метрика **BLEU** (Bilingual Evaluation Understudy), запропонована у роботі [46], первісно розроблялась для оцінювання якості машинного перекладу, проте набула широкого застосування у суміжних задачах генерації тексту. Вона вимірює точність збігу n-грам згенерованого повідомлення із еталонним. Загальна формула BLEU має вигляд:

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^N w_n \cdot \log p_n\right), \quad (1.38)$$

де BP – штраф за надто короткі повідомлення (brevity penalty); p_n – модифікована точність n -грам; w_n – ваги для кожного порядку n -грам (зазвичай $w_n = 1/N$ для $N = 4$). Штраф за стислість визначається як:

$$BP = \begin{cases} 1, & \text{якщо } |c| > |r|, \\ \exp\left(1 - \frac{|r|}{|c|}\right), & \text{якщо } |c| \leq |r|, \end{cases} \quad (1.39)$$

де $|c|$ – довжина згенерованого повідомлення; $|r|$ – довжина еталонного. Модифікована точність n -грам обчислюється як:

$$p_n = \frac{\sum_s \min(C_n(s, h), C_n(s, r))}{\sum_s C_n(s, h)}, \quad (1.40)$$

де $C_n(s, h)$ – кількість входжень n -грами s у згенерованому повідомленні h ; $C_n(s, r)$ – у еталонному r . Попри поширеність, метрика BLEU має суттєві обмеження стосовно задачі генерації повідомлень про зміни: вона чутлива до порядку слів і не враховує семантичну близькість синонімічних виразів, що є критичним для коротких різноманітних повідомлень [47].

Метрика **METEOR** (Metric for Evaluation of Translation with Explicit ORdering) [3] усуває низку недоліків BLEU, враховуючи не лише точні збіги слів, а й стемінг і синонімічні відповідності. Обчислення здійснюється на основі гармонійного середнього точності P та повноти R з більшою вагою для повноти:

$$\text{METEOR} = F_{\text{mean}} \cdot (1 - \text{Pen}), \quad (1.41)$$

де F_{mean} – зважене гармонійне середнє:

$$F_{\text{mean}} = \frac{10PR}{R + 9P}, \quad (1.42)$$

а Pen – штраф за фрагментованість збігів, що враховує порядок слів:

$$\text{Pen} = 0,5 \cdot \left(\frac{ch}{m}\right)^3, \quad (1.43)$$

де ch – кількість груп суміжних збігів (chunks); m – загальна кількість збігів. Вищий показник фрагментованості знижує оцінку, заохочуючи довші неперервні

збіги. METEOR демонструє вищу кореляцію з людською оцінкою порівняно з BLEU для коротких текстів [48].

Метрика **ROUGE** (Recall-Oriented Understudy for Gisting Evaluation) [49], розроблена для оцінювання автоматичного реферування, орієнтована на повноту: вона вимірює частку n-грам еталонного тексту, що присутні у згенерованому. Найпоширенішими варіантами є ROUGE-N та ROUGE-L.

ROUGE-N обчислюється як:

$$\text{ROUGE-N} = \frac{\sum_s \min(C_n(s, h), C_n(s, r))}{\sum_s C_n(s, r)}, \quad (1.44)$$

де підсумовування ведеться по всіх n-грамах еталонного повідомлення. ROUGE-L ґрунтується на найдовшій спільній підпоследовності (Longest Common Subsequence, LCS) та обчислюється як F-міра:

$$\text{ROUGE-L} = \frac{(1 + \beta^2) \cdot R_{\text{lcs}} \cdot P_{\text{lcs}}}{R_{\text{lcs}} + \beta^2 \cdot P_{\text{lcs}}}, \quad (1.45)$$

де $R_{\text{lcs}} = \text{LCS}(h, r)/|r|$, $P_{\text{lcs}} = \text{LCS}(h, r)/|h|$, β – параметр балансу між точністю та повнотою (зазвичай $\beta = 1,2$). Перевагою ROUGE-L є нечутливість до розривів у збігах, що важливо для оцінювання коротких повідомлень довільної структури.

На відміну від попередніх метрик, **BERTScore** [50] використовує контекстні векторні представлення токенів, отримані від попередньо навченої моделі BERT. Це дозволяє вловлювати семантичну близькість між словами навіть за відсутності точних лексичних збігів.

Для кожного токена згенерованого повідомлення $h = (h_1, \dots, h_k)$ та еталонного $r = (r_1, \dots, r_l)$ обчислюється косинусна подібність між їхніми контекстними представленнями. Точність, повнота та F-міра BERTScore визначаються як:

$$P_{\text{BERT}} = \frac{1}{k} \sum_{i=1}^k \max_j \cos(h_i, r_j), \quad (1.46)$$

$$R_{\text{BERT}} = \frac{1}{l} \sum_{j=1}^l \max_i \cos(h_i, r_j), \quad (1.47)$$

$$F_{\text{BERT}} = \frac{2 \cdot P_{\text{BERT}} \cdot R_{\text{BERT}}}{P_{\text{BERT}} + R_{\text{BERT}}}, \quad (1.48)$$

де $\cos(h_i, r_j)$ – косинусна подібність між векторними представленнями tokenів h_i та r_j . Метрика BERTScore демонструє значно вищу кореляцію з людською оцінкою якості порівняно з BLEU та ROUGE [50], і є особливо придатною для задачі генерації повідомлень про внесені зміни, де семантична точність важливіша за точний лексичний збіг.

Метрика **BLEURT** [51] є навченою метрикою оцінювання, що поєднує переваги трансформерних представлень із методологією навчання з учителем. На відміну від BERTScore, яка використовує попередньо навчену модель без додаткового налаштування, BLEURT проходить донавчання на парах (еталонне повідомлення, згенероване повідомлення) з людськими оцінками якості як цільовою змінною.

Формально BLEURT визначається як регресійна модель f_θ , навчена мінімізувати середньоквадратичне відхилення між передбаченою оцінкою та людською:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^N (f_\theta(h_i, r_i) - y_i)^2, \quad (1.49)$$

де h_i – згенероване повідомлення; r_i – еталонне; $y_i \in [0,1]$ – нормалізована людська оцінка; N – кількість навчальних прикладів. Вхідна пара (h, r) кодується моделлю BERT як конкатенація tokenів:

$$[\text{CLS}] \ h_1 \cdots h_k \ [\text{SEP}] \ r_1 \cdots r_l \ [\text{SEP}], \quad (1.50)$$

а вихідне представлення токена $[\text{CLS}]$ подається на лінійний шар для отримання скалярної оцінки якості. Навчання відбувається у два етапи: спочатку модель додатково попередньо навчається на синтетичних парах із автоматичними оцінками (pre-finetuning), а потім донавчається на реальних людських оцінках (finetuning). Завдяки цьому BLEURT демонструє найвищу серед автоматичних

метрик кореляцію з людською оцінкою [51], проте потребує значних обчислювальних ресурсів і залежить від якості навчальних даних із людськими оцінками.

Поряд із зазначеними, у літературі застосовуються й інші метрики. **CIDEr** (Consensus-based Image Description Evaluation) [52] вимірює схожість між згенерованим та еталонним текстом з урахуванням TF-IDF-ваг n -грам, що знижує вплив часто вживаних загальних слів. Для повідомлень про зміни ця властивість є корисною, оскільки дозволяє зосередитися на змістовно значущих термінах. Метрика NIST [53] є модифікацією BLEU, що надає більшої ваги рідкісним n -грамам і таким чином заохочує інформативність повідомлення. Вона обчислюється як:

$$\text{NIST} = \sum_{n=1}^N \left[\frac{\sum_s \text{Info}(s)}{\sum_s C_n(s, h)} \right] \cdot \exp \left(\beta_n \cdot \ln \min \left(\frac{|h|}{|r|}, 1 \right) \right), \quad (1.51)$$

де $\text{Info}(s) = \log_2 \frac{\text{кількість входжень } (n-1)\text{-грами}}{\text{кількість входжень } n\text{-грами } s}$ відображає інформаційну цінність n -грам.

Людська оцінка (human evaluation) залишається «золотим» стандартом оцінювання якості. Зазвичай експерти оцінюють повідомлення за такими критеріями: точність (чи відповідає повідомлення фактичним змінам), повнота (чи охоплює повідомлення всі ключові зміни), стислість (чи є повідомлення лаконічним) та читабельність. Оцінки виставляються за шкалою Лікерта від 1 до 5, а узгодженість між оцінювачами вимірюється за допомогою коефіцієнта Коена κ [54].

Аналіз метрик засвідчує, що жодна з автоматичних метрик не є достатньою для повноцінного оцінювання якості повідомлень про зміни. Найвищу кореляцію з людською оцінкою забезпечують BERTScore та BLEURT, тоді як BLEU та ROUGE залишаються стандартом де-факто для порівняння результатів у літературі завдяки простоті відтворення. Рекомендованою практикою є паралельне застосування кількох метрик, а для підсумкового оцінювання методу – залучення експертів. У роботі для порівняння запропонованого методу з базовими лініями

використовуються метрики BLEU-4, ROUGE, BERTScore та BLEURT як такі, що забезпечують найбільш повну та збалансовану оцінку.

1.6 Висновки до розділу 1 та постановка задач дослідження

У першому розділі дисертаційної роботи проведено комплексний аналіз методів формування та оцінювання якості повідомлень про внесені зміни у системах контролю версій, що дозволив обґрунтувати актуальність дослідження та визначити основні напрями подальшої роботи. За результатами проведеного аналізу сформульовано такі висновки.

1. Встановлено, що повідомлення про внесені зміни (commit messages) є невід’ємним елементом процесу супроводу програмного забезпечення, який становить від 60 % до 80 % загальних витрат протягом життєвого циклу програмного продукту. Показано, що якість повідомлень безпосередньо впливає на ефективність аналізу історії змін, локалізацію дефектів, рефакторинг та командну комунікацію. Водночас емпіричні дослідження свідчать, що значна частина повідомлень у реальних проєктах є надто короткими, шаблонними або семантично неінформативними, що обґрунтовує необхідність розроблення засобів автоматизованого аналізу та генерації таких повідомлень [6].

2. Проведено систематизацію існуючих підходів до автоматичного аналізу коментарів у системах контролю версій, зокрема лексичного та текстового аналізу, статистичних методів (n-грамний аналіз, TF-IDF, тематичне моделювання LDA), методів машинного навчання та обробки природної мови (SVM, Random Forest, LSTM, трансформери) та гібридних підходів. Встановлено, що лексичні та статистичні методи характеризуються простотою реалізації та обчислювальною ефективністю, проте обмежені у здатності враховувати семантичний контекст. Методи ML/NLP забезпечують вищу точність класифікації, однак потребують великих обсягів розмічених даних та значних обчислювальних ресурсів. Найбільш перспективними є гібридні підходи, що поєднують переваги різних методів.

3. Формалізовано задачу автоматичної оцінки якості повідомлень про внесені зміни як задачу бінарної класифікації з комбінованим ознаковим простором, що включає текстові, структурні та статистичні характеристики. Запропоновано

комбінований критерій якості повідомлення, який враховує семантичну обґрунтованість (відповідність тексту фактичним змінам у коді), нормалізовану довжину та наявність ключових слів. Показано, що використання косинусної подібності між векторними представленнями тексту повідомлення та змін у вихідних текстах програм дозволяє кількісно оцінити рівень семантичної узгодженості.

4. Здійснено порівняльний аналіз методів автоматичної генерації повідомлень про внесені зміни, що охоплює евристичні та шаблонні методи, методи витягування інформації, підходи на основі рекурентних нейронних мереж (encoder-decoder з механізмом уваги), трансформерні моделі з попереднім навчанням (T5, CodeT5, BART), методи із використанням структурної інформації абстрактних синтаксичних дерев (AST) та підкріплювального навчання. Встановлено, що трансформерні моделі, орієнтовані на вихідні тексти програм, забезпечують найкращий баланс між якістю генерації та практичністю застосування, а використання підкріплювального навчання у поєднанні з навчанням з учителем додатково покращує результати.

5. Систематизовано метрики оцінювання якості автоматично згенерованих повідомлень, зокрема BLEU, METEOR, ROUGE, BERTScore, BLEURT, CIDEr та NIST. Показано, що метрики на основі контекстних векторних представлень (BERTScore, BLEURT) демонструють найвищу кореляцію з людською оцінкою порівняно з традиційними метриками на основі збігу n-грам. Обґрунтовано доцільність паралельного застосування кількох метрик (BLEU-4, ROUGE, BERTScore, BLEURT) для забезпечення повної та збалансованої оцінки якості.

Наведені висновки дозволяють сформулювати мету дисертаційного дослідження: створення методів оцінки якості та автоматичної генерації повідомлень про внесені зміни у системах контролю версій на основі проектування і застосування моделей нейронних мереж різної архітектури; дослідження ефективності рекурентних та трансформерних моделей для задачі бінарної класифікації повідомлень; розроблення методів мовно-специфічного донавчання та

семантичного збагачення ідентифікаторів вихідного коду для підвищення якості генерації повідомлень.

Для досягнення поставленої мети сформовано наступні задачі:

1. Виконати комплексний аналіз існуючих методів формування та оцінювання якості повідомлень про внесені зміни у системах контролю версій, здійснити систематизацію методів автоматичної генерації повідомлень та метрик оцінювання їхньої якості.
2. Створити метод оцінювання якості повідомлень про внесені зміни на основі рекурентних нейронних мереж із дослідженням впливу попередньо навчених моделей векторного представлення слів на якість бінарної класифікації.
3. Створити метод оцінювання якості повідомлень про внесені зміни на основі трансформерних моделей із застосуванням методології трансферного навчання та виконати порівняльний аналіз ефективності різних сімейств моделей.
4. Розробити метод автоматичної генерації повідомлень про внесені зміни на основі трансформерних моделей типу кодувальник-декодувальник із мовно-специфічним донавчанням для шести мов програмування.
5. Розробити метод семантичного збагачення ідентифікаторів вихідного коду через автоматичне розкриття абревіатур для підвищення якості автоматичної генерації повідомлень про внесені зміни.
6. Виконати експериментальну перевірку ефективності запропонованих методів та провести порівняльний аналіз застосування технологій паралельних обчислень (GPGPU та TPU) для прискорення навчання нейронних моделей.

РОЗДІЛ 2. СТВОРЕННЯ МЕТОДІВ ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПОВІДОМЛЕНЬ ПРО ВНЕСЕНІ ЗМІНИ НА ОСНОВІ НЕЙРОННИХ МЕРЕЖ

2.1 Формування навчальної вибірки на основі GitHub REST API та попередня обробка даних

Ефективність методів машинного навчання для оцінювання якості повідомлень про внесені зміни безпосередньо залежить від якості та репрезентативності навчальної вибірки. У цьому підрозділі обґрунтовано вимоги до навчального корпусу, формалізовано критерії якості повідомлень, описано методику розмітки даних та попередньої обробки текстів, а також визначено стратегію розподілу вибірки для навчання, валідації та тестування моделей.

2.1.1 Джерело даних та вимоги до навчального корпусу

Для формування навчальної вибірки використано вебсервіс GitHub – один із найбільших ресурсів для розміщення ІТ-проектів та спільної розробки програмного забезпечення, в основі якого лежить розподілена система контролю версій Git. GitHub надає спеціальний програмний інтерфейс для застосунків (GitHub REST API), який дозволяє автоматизувати процес збору даних з відкритих репозиторіїв. Вибір GitHub як джерела даних обумовлений низкою чинників: значним обсягом публічних репозиторіїв (понад 200 млн станом на 2023 рік), різноманітністю мов програмування, типів проєктів та стилів написання повідомлень, а також наявністю структурованого програмного інтерфейсу для автоматизованого доступу до історії внесених змін [20, 21].

Для забезпечення репрезентативності навчальної вибірки сформульовано такі вимоги до корпусу даних:

- достатній обсяг для навчання моделей глибокого навчання (не менше кількох тисяч розмічених прикладів);
- збалансованість класів або наявність методів компенсації дисбалансу;
- різноманітність проєктів за предметною сферою, мовою програмування та розміром команди;

- наявність як якісних, так і неякісних повідомлень для забезпечення навчання бінарного класифікатора.

2.1.2 Критерії якості повідомлень про внесені зміни

Незважаючи на відсутність єдиного загальноприйнятого стандарту написання commit-повідомлень, у процесі використання систем контролю версій сформувалась низка загальних рекомендацій, дотримання яких сприяє створенню якісної документації змін, а відхилення від яких свідчить про низьку інформативність або незадовільну структуру опису [55]. На основі аналізу існуючих практик у роботі виокремлено дві групи правил.

1. Правила щодо оформлення повідомлення про внесені зміни:

- опис повинен мати заголовок і може мати тіло, ці частини мають бути розділені порожнім рядком;
- заголовок не має перевищувати 50–70 символів;
- дієслова у заголовку повинні вживатися у доконаній формі.

2. Правила щодо подання інформації в повідомленні про внесені зміни:

- має бути описано, чому були зроблені зміни;
- ефект повідомлення про внесені зміни має бути очевидним;
- якщо внесені зміни виправляють певну проблему, її необхідно вказати;
- подання інформації має бути таким, щоб фахівець, який не має детального розуміння проблеми чи структури коду, зрозумів, що було зроблено і який вплив це має на проєкт.

Слід зазначити, що правила першої групи (а–в) можуть бути перевірені формальними засобами програмування: довжина заголовка, наявність порожнього рядка, граматична форма дієслова. Натомість правило 2.г потребує залучення методів машинного навчання, оскільки визначення того, чи описує повідомлення *мету* та *причину* внесених змін, вимагає семантичного «розуміння» контексту. Саме це правило є ключовим критерієм для задачі бінарної класифікації повідомлень, що розглядається у цьому дослідженні [56].

Приклади повідомлень, що відповідають правилу 2.г, наведено в таблиці 2.1.

Таблиця 2.1 – Приклади повідомлень про внесені зміни, що відповідають критеріям якості

№	Приклад повідомлення	Мета повідомлення
1	Refactored authentication logic to improve code readability and reduce duplication	Покращити читабельність коду та усунути надлишкові фрагменти
2	Fixed race condition in data loader to prevent intermittent test failures	Усунути причину нестабільного проходження тестів
3	Optimized image rendering to improve page load performance on mobile devices	Пришвидшити завантаження на мобільних пристроях
4	Updated password hashing algorithm to enhance security compliance	Підвищити рівень безпеки відповідно до вимог
5	Added error messages to login form to improve user feedback	Забезпечити більш інтуїтивну взаємодію з користувачем
6	Rewrote unit tests to align with updated API behavior	Актуалізувати тести відповідно до оновленої логіки роботи API

2.1.3 Формалізація задачі класифікації

Задачу оцінювання якості повідомлень про внесені зміни формалізовано як задачу бінарної класифікації. Нехай X – множина описів внесених змін, де кожне повідомлення описується m -вимірним вектором слів:

$$\vec{x} = \{x_1, x_2, \dots, x_m\}, \vec{x} \in X, \quad (2.1)$$

де m – розмірність вектора слів; Y – множина можливих відгуків у вигляді M -вимірного вектора:

$$\vec{y} = \{y_1, \dots, y_M\}, \vec{y} \in Y, \quad (2.2)$$

де $M = 2$ – кількість класів: перший клас відповідає повідомленню, яке задовольняє встановленим вимогам якості (зокрема, правилу 2.г), другий –

повідомленню, що не задовольняє цим вимогам. Таким чином, модель фільтра повідомлень описується відображенням:

$$f: X \rightarrow Y \quad (2.3)$$

Зазначене відображення є сюр'єктивним, але не ін'єктивним, оскільки різні повідомлення можуть належати до одного класу, проте кожен клас має хоча б один представник.

2.1.4 Методика розмітки даних

Для навчання моделей бінарної класифікації дані було розмічено вручну відповідно до правила 2.г. Кожному повідомленню присвоєно мітку класу: «1» – якщо повідомлення описує мету та причину внесених змін (тобто відповідає на питання «чому ці зміни було внесено?»), «0» – якщо повідомлення не містить такої інформації. Загальний обсяг розміченої вибірки становив 8 тис. повідомлень.

Аналіз розподілу класів у зібраній вибірці засвідчив наявність дисбалансу: кількість повідомлень, що не відповідають правилу 2.г, суттєво переважала кількість якісних повідомлень. Дисбаланс класів негативно впливає на якість навчання моделей, зокрема призводить до зміщення у бік домінуючого класу [57]. Для подолання цієї проблеми застосовано два підходи (рис. 2.1):

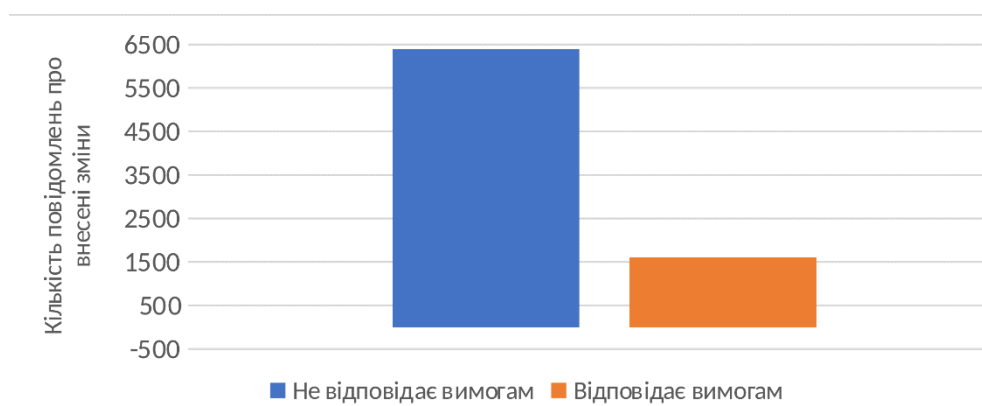


Рисунок 2.1 – Розподіл повідомлень про внесені зміни за класами якості

1. підвибірка (resampling / undersampling) – зменшення кількості прикладів домінуючого класу до рівня мінорного, що забезпечує збалансованість, але зменшує загальний обсяг навчальних даних;
2. зсув порогу класифікації (threshold moving) – налаштування порогового значення ймовірності для прийняття рішення про належність до класу

(коефіцієнт 0,57 замість стандартного 0,5), що дозволяє компенсувати дисбаланс без зменшення обсягу даних.

2.1.5 Попередня обробка текстів

Перед подачею повідомлень на вхід нейронних мереж виконано низку етапів попередньої обробки, спрямованих на нормалізацію та уніфікацію текстових даних [58]:

1. *нормалізація тексту* – приведення всіх літер до нижнього регістру для зменшення розмірності словника та усунення чутливості до регістру;
2. *токенізація* – розбиття текстів повідомлень на послідовності токенів (слів або підслів) за допомогою побудованого на навчальній вибірці токенизатора;
3. *векторизація* – перетворення послідовностей токенів у числові вектори фіксованої довжини. Під час створення об'єкта векторизації довжину вихідної послідовності токенів було зафіксовано, оскільки форма вхідного тензора нейронної мережі має бути сталою. Крім того, розмір словника було обмежено з відкиданням рідковживаних токенів для прискорення обчислень.

2.1.6 Стратегія розподілу вибірки

Для побудови адекватної моделі та запобігання перенавчанню використано метод розподілу на основі тестової підвибірки. Всю сукупність розмічених даних поділено на непересічні частини: 80 % даних використано для навчання моделей-кандидатів, а 20 % – виділено у валідаційну підвибірку для оцінки якості на етапі навчання та підбору гіперпараметрів. Оцінювання фінальної якості моделі виконувалося на окремій тестовій вибірці, яка не використовувалася під час навчання.

2.1.7 Формат вхідних даних для моделей

Підготовлені дані подаються на вхід моделей у вигляді числових послідовностей фіксованої довжини. Для RNN-моделей (підрозділ 2.2) вхідною є послідовність індексів токенів, що проходить через шар Embedding для отримання щільних векторних представлень [59]. Для трансформерних моделей (підрозділ 2.4)

застосовуються спеціалізовані токенизатори відповідних попередньо навчених моделей (BERT, RoBERTa, DistilBERT), які формують вхідні послідовності з урахуванням спеціальних токенів ([CLS], [SEP]) та масок уваги (attention masks) [56].

Таким чином, у підрозділі обґрунтовано методологічні засади формування навчальної вибірки для задачі бінарної класифікації повідомлень про внесені зміни. Визначено критерії якості повідомлень та формалізовано задачу класифікації. Описано методику ручної розмітки даних, підходи до подолання дисбалансу класів та послідовність етапів попередньої обробки текстів [63]. Технічні деталі реалізації збору даних за допомогою GitHub REST API наведено в підрозділі 4.1.

2.2 Метод оцінювання якості повідомлень на основі рекурентних нейронних мереж

У розділі 2.1 сформовано навчальну вибірку розмічених повідомлень про внесені зміни та наведено етапи її попередньої обробки. Подальшим етапом є побудова моделей класифікації, які автоматично визначають, чи відповідає повідомлення критеріям якості, сформульованим у пункті 2.1.2. У цій частині роботи розглянуто метод оцінювання якості повідомлень на основі рекурентних нейронних мереж (RNN), зокрема архітектур LSTM і GRU та їхніх двонаправлених модифікацій (BiLSTM, BiGRU). Ключові результати, представлені нижче, опубліковано в працях [56], [64].

Вибір рекурентних нейронних мереж як базового підходу обумовлений природою вхідних даних: повідомлення про внесені зміни є текстовими послідовностями змінної довжини, в яких порядок слів та контекстні залежності відіграють суттєву роль для визначення інформативності повідомлення. Класичні RNN-архітектури, зокрема LSTM [26] та GRU [65], здатні моделювати довгострокові залежності в послідовностях, що робить їх придатними для задачі бінарної класифікації текстів [59].

2.2.1 Архітектура рекурентних моделей для бінарної класифікації повідомлень

Для вирішення задачі бінарної класифікації повідомлень про внесені зміни побудовано серію моделей зростаючої складності. Кожна модель складається з таких основних компонентів:

- шару векторних представлень (Embedding);
- шарів видобування ознак (Dense або рекурентних);
- вихідного повнозв'язного шару з логістичною сигмоїдною активацією для оцінювання ймовірності належності до кожного з двох класів.

Базова модель. Першу модель побудовано на основі повнозв'язного (Dense) шару без рекурентних компонентів [66]. Архітектура містить два вагових шари: вхідний шар Embedding, що перетворює індекси токенів у щільні векторні представлення розмірністю 128, та вихідний повнозв'язний шар Dense з кількістю нейронів $M = 2$ і функцією активації sigmoid. Між ними розміщено проміжний шар Global Average Pooling, який агрегує послідовність векторів у єдиний вектор фіксованої розмірності. Ця модель слугує базовою (baseline) та використовується для порівняння з рекурентними архітектурами.

Модель Dense з регуляризацією. Для зменшення ефекту перенавчання до базової моделі додано методи регуляризації: шар Dropout [67] з коефіцієнтом $p = 0.3$, який випадково виключає нейрони під час кожної ітерації навчання, та метод ранньої зупинки (early stopping), що припиняє навчання при відсутності покращення метрик на валідаційній вибірці. Шар Dropout розміщено після шару Embedding та після шару Global Average Pooling (рис. 2.2).

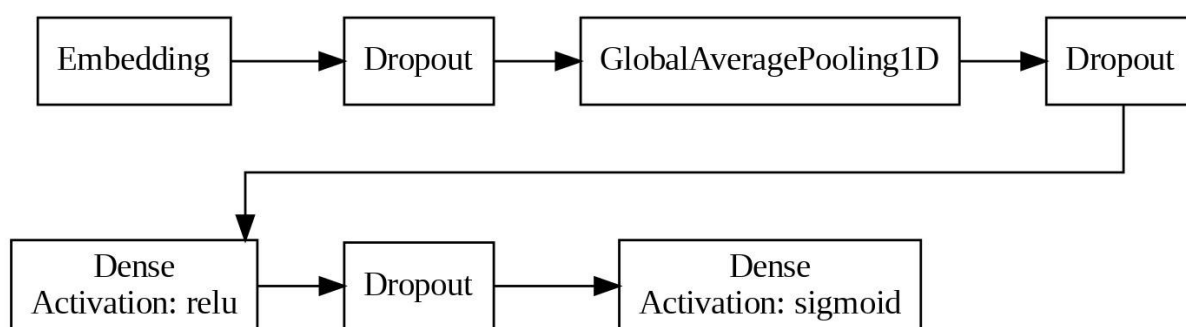


Рисунок 2.2 – Архітектура моделі MLP з методами регуляризації

Модель з LSTM-шаром. Архітектура Long Short-Term Memory (LSTM) [26] є одним із найпоширеніших типів рекурентних нейронних мереж, розроблених для моделювання довгострокових залежностей у послідовних даних. LSTM-комірка містить три механізми керування інформацією (gate): вхідний (input gate), забування (forget gate) та вихідний (output gate), що дозволяє ефективно зберігати та передавати контекстну інформацію на значні відстані в послідовності.

Стан комірки LSTM на кроці t визначається системою рівнянь:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f), \quad (2.4)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i), \quad (2.5)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C), \quad (2.6)$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t, \quad (2.7)$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o), \quad (2.8)$$

$$h_t = o_t \odot \tanh(C_t), \quad (2.9)$$

де f_t, i_t, o_t – вентиля забування, входу та виходу відповідно; C_t – стан комірки; h_t – прихований стан; σ – сигмоїдна функція активації; $\odot$ – поелементне множення; W та b – матриці ваг і вектори зміщення відповідних вентилів [26].

Модель з LSTM-шаром включає:

- шар Embedding (розмірність 128);
- шар Dropout ($p = 0.3$);
- один LSTM-шар з 48 нейронами та функцією активації $\tanh$;
- шар Dropout ($p = 0.3$);
- вихідний Dense-шар з двома нейронами та функцією активації sigmoid.

Рекурентний шар замінює Global Average Pooling, забезпечуючи врахування послідовної природи тексту.

Архітектура GRU. Gated Recurrent Unit (GRU) [65] є спрощеним варіантом LSTM з двома механізмами керування: оновлення (update gate z_t) та скидання (reset gate r_t). Приховані стани GRU обчислюються так:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t]), \quad (2.10)$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t]), \quad (2.11)$$

$$\tilde{\kappa}_t = \tanh(W \cdot [r_t \odot h_{t-1}, x_t]), \quad (2.12)$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{\kappa}_t, \quad (2.13)$$

де z_t – вентиль оновлення; r_t – вентиль скидання. GRU має меншу кількість параметрів порівняно з LSTM (два вентиля замість трьох), що забезпечує швидше навчання при збереженні здатності моделювати довгострокові залежності [65].

Моделі з двонаправленими шарами (BiLSTM, BiGRU). Двонаправлені рекурентні мережі [68] обробляють вхідну послідовність одночасно у прямому та зворотному напрямках, що дозволяє кожному елементу послідовності враховувати як попередній, так і наступний контекст. Вихідне представлення формується шляхом конкатенації прихованих станів обох напрямків:

$$h_t = [h_t^{\rightarrow}; h_t^{\leftarrow}], \quad (2.14)$$

де $h_t^{\rightarrow}$ – прихований стан прямого проходу; $h_t^{\leftarrow}$ – прихований стан зворотного проходу. Це є суттєвою перевагою для задачі класифікації текстів, де зміст повідомлення може залежати від слів, розташованих у різних частинах речення.

Побудовано чотири варіанти моделей з двонаправленими шарами:

- один двонаправлений LSTM-шар з 24 нейронами (загалом 48 нейронів з урахуванням обох напрямків);
- два двонаправлені LSTM-шари з 12 нейронами кожний (загалом 48 нейронів);
- один двонаправлений GRU-шар з 24 нейронами;
- два двонаправлені GRU-шари з 12 нейронами кожний.

Загальна кількість нейронів рекурентних шарів зафіксована на рівні 48 для забезпечення коректності порівняння моделей.

Усі моделі з рекурентними шарами мають спільну структуру: InputLayer $\rightarrow$ Embedding(128) $\rightarrow$ Dropout(0.3) $\rightarrow$ Рекурентний шар (або шари) $\rightarrow$ Dropout(0.3) $\rightarrow$ Dense(2, sigmoid). Для моделей із двома рекурентними шарами між ними додано проміжний шар Dropout. Така уніфікована структура дозволяє ізолювати вплив типу рекурентного шару на якість класифікації. Загалом побудовано сім моделей-

кандидатів: дві моделі на основі Dense-шару (без та з регуляризацією), одну на основі LSTM та по дві моделі BiLSTM і BiGRU (одно- та двошарові варіанти) (рис. 2.3). Детальне експериментальне порівняння цих моделей за метриками Accuracy, F1-score та Recall, а також аналіз кривих навчання наведено в підрозділі 4.2.1.

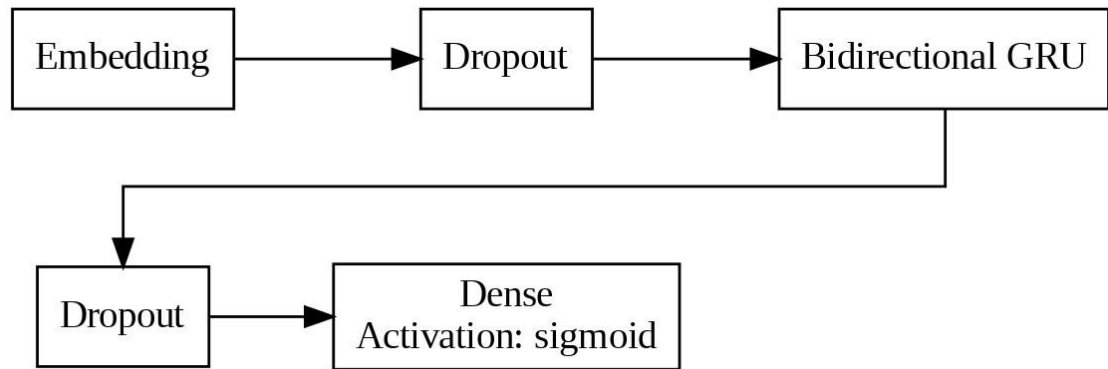


Рисунок 2.3 – Архітектура моделі BiGRU з методами регуляризації

2.2.2 Навчання та регуляризація моделей

Для оптимізації ваг нейронних мереж обрано алгоритм AdamW [69], який є модифікацією класичного алгоритму Adam із застосуванням декомпонованого вагового спаду (decoupled weight decay). На відміну від стандартної L2-регуляризації, де ваговий спад є частиною градієнтного оновлення, в AdamW ваговий спад застосовується окремо від основного процесу оновлення градієнта, що забезпечує коректнішу поведінку регуляризації та краще узагальнення моделі.

Основні кроки алгоритму AdamW полягають у такому. Для кожної ітерації t обчислюється градієнт функції втрат:

$$g_t = \nabla_{\theta} f(\theta_t), \quad (2.15)$$

де $f(\theta)$ – функція втрат; g_t – градієнт для поточних параметрів. Далі оновлюються перший момент (оцінка середнього градієнта) та другий момент (оцінка дисперсії градієнта):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad (2.16)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2. \quad (2.17)$$

Для компенсації зміщення на початкових етапах (оскільки $m_0 = 0$ та $v_0 = 0$) виконується коригування моментів:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}. \quad (2.18)$$

Оновлення параметрів з урахуванням вагового спаду та адаптивної швидкості навчання:

$$\theta_t = \theta_{t-1} - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} + \lambda \theta_{t-1} \right), \quad (2.19)$$

де η – швидкість навчання; β_1, β_2 – коефіцієнти згасання для першого та другого моментів; ε – мале значення для запобігання діленню на нуль; λ – коефіцієнт вагового спаду. Ключова відмінність AdamW від класичного Adam полягає в тому, що доданок $\lambda \theta_{t-1}$ застосовується окремо від адаптивного оновлення на основі градієнтів, що забезпечує стабільніший процес регуляризації [69].

Для запобігання перенавчанню моделей застосовано два основні методи регуляризації. Перший – шар Dropout [67], який під час кожної ітерації навчання випадково обнуляє активації частини нейронів із заданою ймовірністю p . Формально для шару з активаціями h вихідний вектор $\tilde{h}$ визначається як:

$$\tilde{h} = m \odot h, \quad m_i \sim \text{Bernoulli}(1 - p), \quad (2.20)$$

де m – бінарна маска, що генерується незалежно для кожного нейрона на кожній ітерації; $\odot$ – поелементне множення; p – імовірність «вимикання» нейрона (dropout rate).

У побудованих моделях використано коефіцієнт $p = 0.3$. Dropout перешкоджає спільній адаптації (co-adaptation) ваг нейронів, примушуючи мережу формувати більш робастні та незалежні ознаки [67].

Другий – метод ранньої зупинки (early stopping), що припиняє процес навчання, коли значення метрики на валідаційній вибірці перестає покращуватися протягом заданої кількості епох (patience). Цей метод запобігає надмірній спеціалізації моделі на навчальних даних та забезпечує вибір моделі з оптимальним балансом між точністю на навчальній та валідаційній вибірках.

Подолання дисбалансу класів. Як зазначено в підрозділі 2.1.4, навчальна вибірка характеризується нерівномірним розподілом класів: повідомлення, що не відповідають критеріям якості, кількісно переважають якісні повідомлення. Для подолання цієї проблеми досліджено метод зсуву порогу (threshold moving). Замість стандартного порогу 0.5 для бінарної класифікації використано зміщений поріг τ , що дозволяє збалансувати кількість помилок першого та другого роду:

$$\hat{y}_i = \begin{cases} 1, & \text{якщо } P(y = 1|x_i) \geq \tau, \\ 0, & \text{інакше.} \end{cases} \quad (2.21)$$

Значення порогу $\tau = 0.57$ було підібрано емпірично на валідаційній вибірці. Порівняння ефективності цього методу із результатами без його застосування наведено в підрозділі 4.2.1.

Для оцінювання якості класифікації використано метрики Accuracy та F1-score, визначені в підрозділі 1.5. Для задачі фільтрації повідомлень про внесені зміни особливу увагу приділено метриці повноти (Recall), оскільки з практичної точки зору краще відхилити повідомлення та запропонувати розробнику його доповнити, ніж пропустити низькоякісний опис у систему контролю версій. Це відповідає загальному принципу мінімізації помилок другого роду (false negatives) у задачах забезпечення якості програмного забезпечення [56].

Таким чином, у підрозділі запропоновано метод оцінювання якості повідомлень про внесені зміни на основі рекурентних нейронних мереж. Побудовано серію з семи моделей-кандидатів різної архітектури: від базової моделі з повнозв'язним шаром до двошарових двонаправлених рекурентних мереж (BiLSTM, BiGRU). Описано алгоритм оптимізації AdamW з декомпонованим ваговим спадом, методи регуляризації (Dropout, рання зупинка) та підхід до подолання дисбалансу класів через зсув порогу. Уніфікована архітектура моделей із фіксованою загальною кількістю нейронів (48) забезпечує коректність порівняльного аналізу, результати якого детально наведено в підрозділі 4.2.1.

2.3 Роль моделей векторного представлення слів (*Word2Vec*, *FastText*, *GloVe*) у класифікації повідомлень про внесені зміни

Ефективність нейромережових класифікаторів тексту суттєво залежить від якості векторного представлення слів, що подаються на вхід моделі. Шар Embedding, який входить до складу як MLP-, так і RNN-архітектур, перетворює дискретні індекси слів у щільні вектори фіксованої розмірності, зберігаючи семантичні та синтаксичні зв'язки між словами у неперервному векторному просторі. Якість цих представлень безпосередньо впливає на здатність класифікатора узагальнювати навчальні дані та коректно класифікувати нові повідомлення.

Серед методів побудови попередньо навчених векторних представлень слів найширшого застосування в задачах обробки природної мови набули три підходи: Word2Vec, FastText і GloVe. Кожен із них базується на принципово відмінних теоретичних засадах: нейромережевому локальному контексті (Word2Vec, FastText) та глобальній статистиці спільної зустрічальності (GloVe). Нижче наведено математичне формулювання кожного методу та визначено умови їх доцільного застосування для класифікації технічних текстів.

Підхід Word2Vec пропонує два варіанти архітектури: CBOW (Continuous Bag-of-Words) та Skip-gram. У моделі Skip-gram цільове слово w_t використовується для передбачення контекстних слів. Умовна ймовірність контекстного слова w_c за цільовим w_t визначається через нормалізований скалярний добуток відповідних векторів:

$$P(w_c | w_t) = \frac{\exp(u_c^T \cdot v_t)}{\sum_j \exp(u_j^T \cdot v_t)}, \quad (2.22)$$

де v_t – вектор цільового слова; u_c та u_j – контекстні вектори відповідних слів, а підсумовування у знаменнику ведеться по всьому словнику V . Модель навчається шляхом максимізації логарифмічної функції правдоподібності на навчальному корпусі [70]. Основним недоліком є відсутність векторів для слів, що не увійшли до словника під час навчання.

Метод FastText розширює підхід Word2Vec, додаючи до розгляду підслівні n -грами. Вектор слова w будується як середнє арифметичне векторів усіх n -грам з множини G_w , що утворюють це слово:

$$v_w = \frac{1}{|G_w|} \sum_{g \in G_w} z_g, \quad (2.23)$$

де z_g – вектор підслівної n -грами g ; $|G_w|$ – кількість n -грам слова w . Така конструкція дозволяє формувати вектори для незнайомих слів шляхом декомпозиції на відомі підрядки, що особливо важливо для технічних аббревіатур, назв бібліотек та нестандартних токенів, характерних для повідомлень про внесені зміни у вихідних текстах програм [71].

Модель GloVe (Global Vectors for Word Representation) використовує матрицю глобальної спільної зустрічальності слів у корпусі. Метод мінімізує зважену суму квадратів відхилень між скалярним добутком векторів слів та логарифмом їхньої спільної частоти:

$$J = \sum_{i,j=1}^{|V|} f(X_{ij})(w_i^T w_j + b_i + b_j - \log X_{ij})^2, \quad (2.24)$$

де X_{ij} – кількість спільних появ слів i та j у корпусі; w_i, w_j – вектори слів, b_i та b_j – скалярні зсуви; $f(X_{ij})$ – функція зважування, що зменшує вплив надмірно частих пар. Модель GloVe потребує значного обсягу навчальних даних і суттєвих обчислювальних ресурсів, що ускладнює її використання на малих предметно-орієнтованих корпусах [72].

Для підтримки технічної лексики, характерної для повідомлень про внесені зміни, важливим параметром є покриття словника – частка слів навчального корпусу, для яких у попередньо навченій моделі існують вектори:

$$C = \frac{|V_{emb} \cap V_{corpus}|}{|V_{corpus}|}, \quad (2.25)$$

де V_{emb} – словник моделі ембедінгів; V_{corpus} – словник навчального корпусу. Чим вище значення C , тим менша кількість токенів замінюється нульовими векторами під час ініціалізації шару Embedding, що позитивно позначається на

якості навчання класифікатора. Відповідно до цього критерію перевагу слід надавати моделям, навченим на технічних текстах та таким, що підтримують генерацію векторів для незнайомих слів.

Порівняльну характеристику трьох методів за ключовими критеріями якості та застосовності наведено в таблиці 2.2.

Таблиця 2.2 – Порівняльна характеристика методів побудови векторних представлень слів

Характеристика	Word2Vec	FastText	GloVe
Основа методу	Нейромережа (CBOW, Skip-gram)	Нейромережа + n-грами підслів	Матричний факторинг
Тип контексту	Локальний (вікно слів)	Локальний (вікно слів)	Глобальний (матриця зустрічальності)
Врахування морфології	Не враховується	Так (підслівні n-грами)	Не враховується
Підтримка нових слів	Ні (лише слова зі словника)	Так (генерація за n-грамами)	Ні (лише слова зі словника)
Якість на малих корпусах	Достатня	Висока (завдяки n-грамам)	Низька (потребує великих даних)
Обчислювальна складність	Середня	Вища за Word2Vec	Висока (великі матриці)

2.3.1 Застосування векторних подань слів у класифікаторах MLP та RNN

З метою порівняльного аналізу впливу різних методів побудови векторних представлень на точність бінарної класифікації повідомлень про внесені зміни у роботі використано шість попередньо навчених моделей ембедінгів, що охоплюють усі три розглянуті методи та різні корпуси тренування. Ці моделі застосовано для ініціалізації ваг вхідного шару Embedding як у MLP-класифікаторі,

так і в RNN-класифікаторі на основі двонаправленого шару BiGRU, архітектуру яких описано у підрозділі 2.2.

Ваги шару Embedding розмірністю (vocab_size, embedding_dim) ініціалізуються значеннями із попередньо навченої моделі. Для слів, що входять до словника моделі, вектори завантажуються безпосередньо, а для решти токенів застосовується нульова ініціалізація. Під час навчання класифікатора ваги шару Embedding «не заморожуються», що дозволяє моделі адаптувати представлення до специфіки навчального корпусу повідомлень.

Характеристики шести застосованих попередньо навчених моделей векторних подань слів наведено в таблиці 2.3.

Таблиця 2.3 – Характеристики попередньо навчених моделей векторних представлень

Модель	Метод	Розмір вектора	Корпус тренування	Особливості
word2vec-google-news-300	Word2Vec	300	Google News (~100 млрд слів)	Загальна англійська мова
SO_vectors_200	Word2Vec	200	Stack Overflow (технічні тексти)	Спеціалізована для технічних текстів
fasttext-wiki-news-subwords-300	FastText	300	Вікіпедія	Враховує морфологію, підтримка нових слів
glove-twitter-200	GloVe	200	Twitter (~27 млрд токенів)	Соціальні мережі, неформальна лексика
glove-wiki-gigaword-50	GloVe	50	Вікіпедія + Gigaword	Компактна, знижена розмірність
glove-wiki-gigaword-300	GloVe	300	Вікіпедія + Gigaword	Повнорозмірна версія

Оцінювання ефективності побудованих класифікаторів здійснено за двома метриками: часткою правильних відповідей (Accuracy) та середнім гармонійним (F1-score) [73].

2.4 Метод оцінювання якості повідомлень на основі трансформерних моделей (BERT, RoBERTa, DistilBERT)

2.4.1 Архітектура трансформерних класифікаторів

Трансформерні моделі, на відміну від рекурентних мереж, описаних у підрозділі 2.2, опрацьовують усю вхідну послідовність паралельно без збереження прихованого стану. Це зменшує прояви проблеми зникаючого градієнта та забезпечує ефективне моделювання довготривалих залежностей між токенами повідомлення, що є особливо важливим для технічних текстів, у яких термін на початку речення може визначати контекст дієслова наприкінці. Основою трансформерного блоку є механізм масштабованої скалярної уваги, формула якого наведена у підрозділі 1.4, формула (1.34).

Ключовим розширенням є багатоголова самоувага (Multi-Head Self-Attention, МНА), що дозволяє одночасно проєктувати запити, ключі та значення у h незалежних підпросторів і паралельно обчислювати увагу в кожному:

$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W_O, \quad (2.26)$$

де $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$ – i -та голова уваги; $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d \times d_k}$ – проєкційні матриці голови; $W^O \in \mathbb{R}^{(hd_k) \times d}$ – вихідна проєкційна матриця. Розподіл на h голів дозволяє моделі одночасно аналізувати синтаксичні, семантичні та позиційні аспекти тексту в різних підпросторах представлень [74].

Кожен блок кодувальника трансформера також містить пряме з'єднання (Feedforward Network, FFN) – дворівневу повнозв'язну мережу, що застосовується незалежно до кожного токена:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2, \quad (2.27)$$

де W_1, W_2 – матриці ваг; b_1, b_2 – вектори зміщень; d_{ff} – прихований розмір FFN (наприклад, 2048 у BERT-base). Функція $max(0, \cdot)$ є нелінійною активацією ReLU. Виходи МНА та FFN нормалізуються з використанням залишкових зв'язків:

$$h_l = LN(h_{l-1} + MHA(h_{l-1})), \quad (2.28)$$

де $LN(\cdot)$ – нормалізація шару (Layer Normalization). Залишкові зв'язки запобігають деградації градієнта при збільшенні кількості шарів, а нормалізація стабілізує розподіл активацій під час навчання. Повна трансформерна модель складається зі стеку N таких блоків (рис. 2.4).

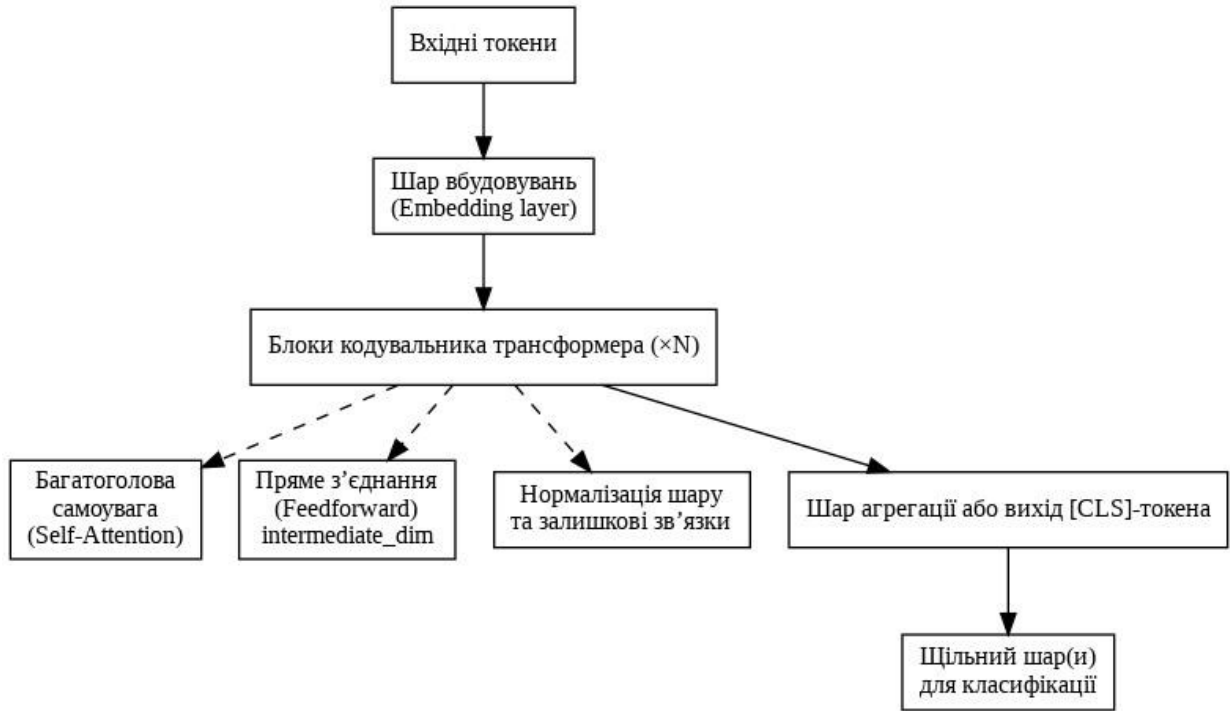


Рисунок 2.4 – Узагальнена архітектура трансформерного класифікатора на основі BERT

Для бінарної класифікації повідомлень використовується агреговане представлення спеціального токена [CLS], доданого на початок кожної вхідної послідовності. Після проходження через N блоків кодувальника вектор $h_{[CLS]}$ передається на вихідний Dense-шар (рис. 2.5):

$$p = \text{softmax}(W_c \cdot h_{[CLS]} + b_c), \quad (2.29)$$

де $W_c \in \mathbb{R}^{d \times 2}$, $b_c \in \mathbb{R}^2$ – параметри, що навчаються, класифікаційної голови; p – вектор ймовірностей двох класів. Для XLM-RoBERTa застосовується Global Average Pooling замість [CLS]-токена.

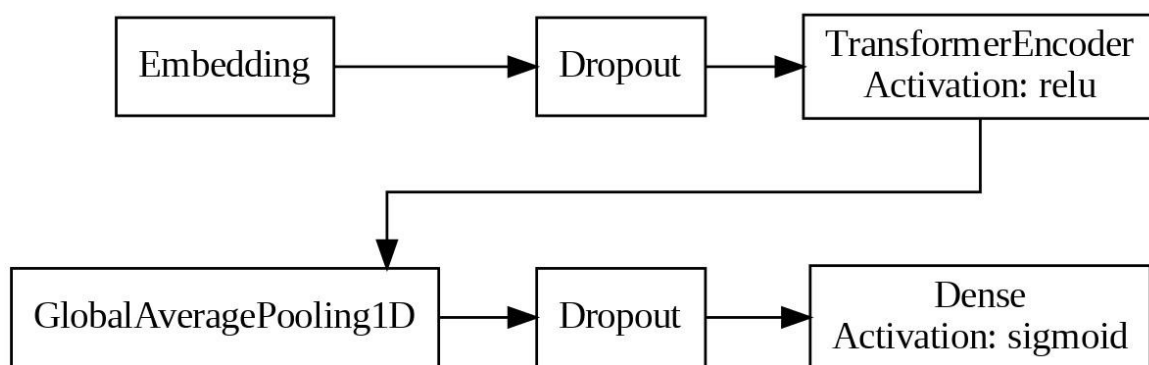


Рисунок 2.5 – Архітектура класифікатора з шаром TransformerEncoder

У дослідженні порівнюються чотири сімейства трансформерних моделей, які відрізняються за стратегією попереднього навчання, ступенем стиснення та мовним охопленням. Їхню загальну характеристику наведено в таблиці 2.4.

Таблиця 2.4 – Порівняльна характеристика трансформерних моделей

Характеристика	BERT	RoBERTa	DistilBERT	XLM-RoBERTa
Архітектура	Двонаправл. Encoder	Покращений BERT	Дистильований BERT	Мультимовний RoBERTa
К-сть параметрів (base)	~110 млн	~125 млн	~66 млн	~270 млн
Мови	Англійська	Англійська	Англійська	Понад 100 мов
Маскування (MLM)	15%, статичне	Динамічне	Статичне	Динамічне
NSP-задача	Так	Ні	Ні	Ні
Метод стиснення	–	–	Knowledge distillation	–
Агрегація виходу	[CLS]-токен	[CLS]-токен	[CLS]-токен	Global Avg Pooling

Модель BERT (Bidirectional Encoder Representations from Transformers) – двонаправлена encoder-модель, що навчається на задачах маскованого мовного моделювання (MLM) та передбачення наступного речення (NSP).

Двонаправленість забезпечується одночасним урахуванням лівого та правого контексту через механізм самоуваги, що принципово відрізняє BERT від авторегресійних моделей [60].

Модель RoBERTa (Robustly Optimized BERT Pretraining Approach) є вдосконаленою версією BERT: застосовується динамічне маскуванню токенів (нова маска генерується для кожної епохи), видалено NSP-задачу та суттєво збільшено корпус попереднього навчання. Ці зміни забезпечують кращу здатність до узагальнення при донавчанні на спеціалізованих корпусах [61].

Модель DistilBERT – стиснена версія BERT, отримана методом дистиляції знань. Архітектурно DistilBERT містить удвічі менше трансформерних блоків (6 замість 12 у BERT-base), що дозволяє зменшити кількість параметрів з ~110 млн до ~66 млн та пришвидшити інференс у 1,6 рази при збереженні 97% якості [62].

Модель XLM-RoBERTa – мультимовна версія RoBERTa, навчена на CommonCrawl більш ніж 100 мовами. Попри те що досліджуваний корпус переважно англomовний, включення XLM-RoBERTa дозволяє оцінити вплив мультимовного попереднього навчання на розпізнавання технічної лексики [75].

Параметри всіх шістнадцяти конфігурацій, що досліджуються у роботі, наведено в таблиці 2.5.

Таблиця 2.5 – Параметри досліджуваних конфігурацій трансформерних моделей

Назва моделі	Пар., млн	Шари	Голови	Вектор	Batch	Регістр
transformer-encoder	–	1	4	512	128	–
bert_tiny_en_uncased	~4	2	2	128	128	Hi
bert_small_en_uncased	~29	4	4	256	128	Hi
bert_medium_en_uncased	~42	8	8	512	128	Hi
bert_base_en_uncased	~110	12	12	768	64	Hi
bert_base_en	~110	12	12	768	64	Так
bert_large_en_uncased	~340	24	16	1024	16	Hi
bert_tiny_en_uncased_sst2	~4	2	2	128	128	Hi

roberta_base_en	~125	12	12	768	64	Так
roberta_large_en	~355	24	16	1024	16	Так
distil_bert_base_en_uncased	~66	6	12	768	128	Ні
distil_bert_base_en	~66	6	12	768	128	Так
xlm_roberta_base_multi	~270	12	12	768	64	Так
xlm_roberta_large_multi	~550	24	16	1024	16	Так

2.4.2 Донавчання попередньо навчених моделей

Запропонований підхід ґрунтується на методології трансферного навчання: трансформерні моделі, попередньо навчені на великих загальних корпусах, донавчаються на задачно-специфічному наборі маркованих повідомлень про внесені зміни. Це дозволяє адаптувати універсальні семантичні представлення до специфіки технічної мови commit-повідомлень за відносно невелику кількість епох.

На відміну від підходів підрозділів 2.2 та 2.3, де адаптувались лише ваги шарів Embedding та рекурентних шарів, при донавчанні трансформерних моделей увесь стек кодувальника залишається тренуваним. Це забезпечує адаптацію контекстно залежних представлень до специфіки домену, однак вимагає значно більших обчислювальних ресурсів. Навчання виконувалось у середовищі Google Colaboratory з використанням GPU NVIDIA A100 (40 ГБ пам'яті), що зумовило необхідність коригування batch size відповідно до обсягу пам'яті (16–128 залежно від розміру моделі).

Для DistilBERT, параметри якого отримано шляхом дистиляції знань, процедура попереднього навчання мінімізувала розбіжність між розподілами виходів моделі-вчителя (BERT) та учня. Завдяки дистиляції DistilBERT успадковує узагальнювальну здатність BERT у компактнішій формі, що є перевагою за обмеженого обчислювального бюджету.

Для всіх трансформерних класифікаторів донавчання здійснюється мінімізацією бінарної крос-ентропійної функції втрат:

$$L_{ft} = -\frac{1}{N} \sum_{i=1}^N (y_i \log p_i + (1 - y_i) \log(1 - p_i)), \quad (2.30)$$

де y_i – мітка класу i -го прикладу (1 – повідомлення, що задовольняє вимогам, 0 – повідомлення, що не задовольняє вимогам); p_i – передбачена ймовірність за формулою (2.29); N – розмір навчальної вибірки. Для оптимізації ваг використовується алгоритм AdamW, описаний у підрозділі 2.2.2 формулами (2.15) – (2.19), із застосуванням Dropout та методу ранньої зупинки. Розподіл даних: 20% маркованих прикладів виділено на перевіірочну підмножину, решта – для тренування.

2.5 Застосування технологій GPGPU та TPU для прискорення навчання моделей

2.5.1 Архітектура паралельних обчислювальних платформ

Навчання нейронних мереж належить до обчислювально інтенсивних задач: кожна ітерація градієнтного спуску вимагає виконання мільярдів операцій множення з накопиченням (MAC-операцій) у паралельному режимі. Для забезпечення прийняттого часу навчання описаних у підрозділах 2.2 – 2.4 моделей у дослідженні здійснено порівняльний аналіз трьох обчислювальних платформ: центрального процесора (CPU), технології обчислень загального призначення на графічних процесорах (GPGPU) та тензорного процесора (TPU). Усі експерименти виконано у хмарному середовищі Google Colaboratory (Google Colab), переваги якого для задач машинного навчання описано у підрозділі 2.4.2 [76].

Технологія GPGPU (General-Purpose computing on Graphics Processing Units) ґрунтується на використанні масиву потокових мультипроцесорів (Streaming Multiprocessors, SM), кожен із яких містить від кількох сотень до кількох тисяч обчислювальних ядер. Графічний процесор Nvidia Tesla T4, задіяний у дослідженні, містить 2560 CUDA-ядер та 320 спеціалізованих тензорних ядер (Tensor Cores), призначених для виконання операцій матричного множення у форматі FP16 із накопиченням у FP32. Ключовою особливістю архітектури GPGPU є паралелізм за даними: кожен потоковий мультипроцесор незалежно обробляє свою підмножину вхідного тензора, що забезпечує ефективне векторизоване обчислення прямого та зворотного поширення помилки.

TPU (Tensor Processing Unit) – спеціалізований інтегральний процесор, розроблений компанією Google та оптимізований для виконання великих матричних операцій у нейронних мережах. Архітектурно TPU базується на систолічному масиві множників (systolic array): дані та ваги послідовно передаються крізь матрицю обчислювальних елементів, що дозволяє мінімізувати звернення до пам'яті та максимально завантажити обчислювальні блоки. TPU v2, що використовувався в роботі, містить 8 ядер із загальним об'ємом пам'яті типу HBM 64 ГБ і оптимізований для форматів із зниженою точністю (FP16, bfloat16, INT8). Порівняльну характеристику технологій GPGPU та TPU наведено в таблиці 2.6.

Таблиця 2.6 – Порівняльна характеристика технологій GPGPU та TPU

Характеристика	TPU (Google TPU v2)	GPGPU (Nvidia Tesla T4)
Призначення	Оптимізовано для тензорних операцій нейронних мереж (TensorFlow)	Паралельні обчислення загального призначення, рендеринг, машинне навчання
Виробник	Google	Nvidia
Архітектура	Систолічний масив матричних множників	Набір потокових мультипроцесорів (SM), 320 тензорних ядер
Тип обчислень	Оптимізовано для операцій із зниженою точністю (FP16, INT8)	Гнучка точність (FP16, FP32, FP64)
Кількість ядер	8 ядер TPU v2	2560 CUDA-ядер + 320 тензорних ядер
Програмне забезпечення	TensorFlow, Google Cloud TPU API	TensorFlow, PyTorch, Keras, CUDA
Пам'ять	HBM (High Bandwidth Memory), 8 ГБ на ядро	GDDR6, 16 ГБ

Доступність	Лише як хмарна послуга Google	Хмарна послуга або локальний пристрій
Енергоспоживання	Оптимізоване для ML-задач	Вище, але більш гнучке
Гнучкість	Спеціалізований для ML	Висока універсальність

Основна відмінність між платформами полягає у ступені спеціалізації: TPU розроблено виключно для операцій глибокого навчання і досягає максимальної ефективності на регулярних матричних обчисленнях, тоді як GPGPU є більш універсальним і може застосовуватись у ширшому спектрі задач. Вибір між платформами зумовлюється вимогами до продуктивності, архітектурою моделі та бюджетом пам'яті.

2.5.2 Ефективність застосування паралельних обчислень для задачі класифікації

Для кількісного оцінювання прискорення навчання моделей при переході з CPU на альтернативні обчислювальні платформи введено коефіцієнт прискорення S_k :

$$S_k = \frac{T_{CPU}}{T_k}, \quad (2.31)$$

де S_k – коефіцієнт прискорення для платформи k ; T_{CPU} – час навчання на CPU; T_k – час навчання на платформі k (TPU або GPU). Чим вище значення S_k , тим ефективнішим є застосування відповідної платформи порівняно зі стандартним процесором.

Для оцінки якості класифікатора у завданні фільтрації повідомлень крім метрик Precision, Recall та F1-score, визначених формулою (1.4), використовується також частка правильних відповідей (Accuracy), яка відображає загальну частку правильно класифікованих прикладів.

Порівняльний аналіз часу навчання рекурентних моделей підрозділу 2.2 на платформах CPU, TPU та GPU наведено в таблиці 2.7. Обчислення проводились у

Google Colab: CPU – Intel Xeon (2 ядра), TPU – Google TPU v2 (8 ядер), GPU – Nvidia Tesla T4 (320 тензорних ядер).

Таблиця 2.7 – Порівняльний аналіз часу навчання моделей на різних апаратних платформах

Модель	CPU (Intel Xeon)	TPU (Google v2)	GPU (Nvidia Tesla T4)	Прискорення GPU/CPU
Dense (без регуляризації)	44 хв 7 с	9 хв 47 с	2 хв 26 с	$\approx 18,2\times$
Dense (з регуляризацією)	57 хв 52 с	7 хв 33 с	1 хв 55 с	$\approx 30,2\times$
LSTM (48 нейронів)	59 хв 34 с	15 хв 56 с	1 хв 4 с	$\approx 56,2\times$
BiLSTM (1 шар, 24 нейрони)	1 год 10 хв 59 с	11 хв 38 с	1 хв 33 с	$\approx 45,5\times$
BiLSTM (2 шари, 12 нейронів)	1 год 14 хв 34 с	16 хв 56 с	2 хв 35 с	$\approx 28,8\times$
BiGRU (1 шар, 24 нейрони)	1 год 3 хв 19 с	10 хв 36 с	1 хв 34 с	$\approx 40,5\times$
BiGRU (2 шари, 12 нейронів)	1 год 5 хв 55 с	18 хв 53 с	2 хв 49 с	$\approx 23,4\times$
Середнє прискорення	$1\times$	$\approx 6\times$	$\approx 45\times$	–

Аналіз результатів таблиці 2.7 демонструє суттєву перевагу GPU над іншими платформами: середнє прискорення GPU порівняно з CPU становить близько 45 разів, а порівняно з TPU – близько 15 разів. TPU перевищує CPU в середньому у 6 разів. Найвище прискорення GPU/CPU спостерігається для моделі LSTM (48 нейронів) – майже $56\times$, що пояснюється особливо ефективним відображенням рекурентних операцій на тензорні ядра CUDA. Для моделей із більшою кількістю шарів (BiLSTM, BiGRU) прискорення дещо нижче через зростання обсягу синхронізаційних витрат між ядрами.

Таким чином, для задачі навчання нейронних класифікаторів повідомлень про внесені зміни GPU-акселерація є оптимальним вибором, що забезпечує скорочення часу одного тренувального циклу з кількох годин (CPU) до лічених хвилин (GPU). Це критично важливо при масштабуванні дослідження на більший корпус повідомлень або при проведенні гіперпараметричного пошуку, де кількість запусків може сягати кількох десятків. Водночас платформа TPU виявляється ефективнішою за CPU при використанні більших batch-розмірів, оскільки систолічний масив досягає максимальної завантаженості саме при великих матричних операціях.

2.6 Висновки до розділу 2

У другому розділі дисертаційної роботи розроблено методи забезпечення якості повідомлень про внесені зміни на основі нейронних мереж різної архітектури. Сформовано навчальну вибірку, побудовано та досліджено моделі бінарної класифікації на основі рекурентних і трансформерних нейронних мереж, проаналізовано вплив попередньо навчених векторних представлень слів та обчислювальних платформ на ефективність навчання. За результатами проведеної роботи сформульовано такі висновки.

1. Сформовано навчальну вибірку обсягом 8 тис. повідомлень про внесені зміни, зібраних з відкритих репозиторіїв вебсервісу GitHub за допомогою GitHub REST API. Визначено критерії якості повідомлень, серед яких ключовим є семантична інформативність – наявність опису мети та причини внесених змін. Формалізовано задачу оцінювання якості як задачу бінарної класифікації. Описано методику ручної розмітки даних, етапи попередньої обробки текстів (нормалізація, токенизація, векторизація) та стратегію розподілу вибірки (80 % – навчальна, 20 % – валідаційна). Для подолання дисбалансу класів запропоновано застосування підвибірки та зсуву порогу класифікації.

2. Запропоновано метод оцінювання якості повідомлень про внесені зміни на основі рекурентних нейронних мереж. Побудовано серію із семи моделей-кандидатів зростаючої складності: від базової моделі з повнозв'язним шаром (Dense) до двошарових двонаправлених рекурентних мереж (BiLSTM, BiGRU). Для

всіх моделей використано уніфіковану архітектуру із фіксованою загальною кількістю нейронів рекурентних шарів (48), що забезпечує коректність порівняльного аналізу. Для оптимізації ваг застосовано алгоритм AdamW із декомпонованим ваговим спадом, а для запобігання перенавчанню – шар Dropout та метод ранньої зупинки. Для компенсації дисбалансу класів використано метод зсуву порогу класифікації з емпірично підібраним значенням порогу.

3. Досліджено вплив попередньо навчених моделей векторного представлення слів (Word2Vec, FastText, GloVe) на якість бінарної класифікації повідомлень. Проведено порівняльний аналіз шести моделей ембедінгів, що відрізняються за методом побудови, розмірністю вектора та корпусом тренування. Показано, що метод FastText має перевагу для технічних текстів завдяки підтримці підслівних n-грам, що дозволяє формувати вектори для незнайомих слів – технічних аббревіатур, назв бібліотек та нестандартних токенів, характерних для повідомлень про внесені зміни.

4. Запропоновано метод оцінювання якості повідомлень на основі трансформерних моделей із застосуванням методології трансферного навчання. Досліджено шістнадцять конфігурацій на базі чотирьох сімейств моделей – BERT, RoBERTa, DistilBERT та XLM-RoBERTa – які відрізняються за кількістю параметрів (від 4 до 550 млн), стратегією попереднього навчання, ступенем стиснення та мовним охопленням. Класифікація виконується на основі агрегованого представлення спеціального токена [CLS] з подальшим донавчанням усього стеку кодувальника на задачно-специфічному наборі маркованих повідомлень. Для оптимізації використано бінарну крос-ентропійну функцію втрат та алгоритм AdamW із регуляризацією.

5. Проведено порівняльний аналіз ефективності застосування технологій паралельних обчислень (GPGPU та TPU) для прискорення навчання нейронних класифікаторів. Встановлено, що GPU-акселерація (Nvidia Tesla T4) забезпечує середнє прискорення у 45 разів порівняно з CPU та у 15 разів порівняно з TPU. Найвище прискорення GPU/CPU (56×) зафіксовано для моделі LSTM, що пояснюється ефективним відображенням рекурентних операцій на тензорні ядра.

Показано, що GPU-акселерація є оптимальним вибором для задачі навчання нейронних класифікаторів повідомлень, оскільки дозволяє скоротити час тренувального циклу з кількох годин до лічених хвилин, що є критично важливим при проведенні гіперпараметричного пошуку.

РОЗДІЛ 3. РОЗРОБКА МЕТОДІВ АВТОМАТИЧНОЇ ГЕНЕРАЦІЇ ПОВІДОМЛЕНЬ ПРО ВНЕСЕНІ ЗМІНИ НА ОСНОВІ ТРАНСФОРМЕРНИХ МОДЕЛЕЙ

3.1 Формалізація задачі автоматичної генерації повідомлень як задачі умовної генерації тексту

Автоматична генерація повідомлень про внесені зміни (commit messages) на основі текстових репрезентацій змін у вихідних текстах програм є задачею умовної генерації тексту (conditional text generation). На відміну від задачі бінарної класифікації повідомлень, розглянутої у розділі 2, де модель оцінює якість існуючого повідомлення, задача генерації передбачає побудову нового текстового опису змін на основі їх структурованої репрезентації. Формалізація цієї задачі дозволяє чітко визначити вхідні та вихідні простори, цільову функцію оптимізації, механізм автоматичного декодування та критерії оцінювання якості згенерованих повідомлень.

Нехай X – множина текстових представлень змін у вихідному коді (наприклад, виходів команди *git diff*), а Y – множина текстових повідомлень про внесені зміни, що описують характер, причину та вплив відповідних модифікацій. Кожен елемент $x \in X$ визначається як послідовність токенів, що відображає додані, видалені та змінені фрагменти коду разом із контекстом оточуючих рядків. Кожен елемент $y \in Y$ є послідовністю токенів природної мови, що формує опис відповідних змін.

Для навчання моделі використовується корпус прикладів $D = \{(x_i, y_i)\}_{i=1}^N$, зібраний з відкритих репозиторіїв, де кожна пара містить текстову репрезентацію змін у коді та відповідне повідомлення. Обсяг корпусу N визначається кількістю зібраних та відфільтрованих пар «зміни-повідомлення».

Генеративна модель визначається як параметризоване відображення $f_\theta: X \rightarrow Y$, де θ – вектор параметрів (ваг) трансформерної моделі. Модель апроксимує умовну ймовірність $P(y|x; \theta)$ – ймовірність генерації повідомлення y за умови спостереження змін x при фіксованих параметрах θ .

Задача навчання полягає у знаходженні оптимальних параметрів моделі, що мінімізують функцію втрат на навчальному корпусі:

$$\theta^{opt} = \arg \min_{\theta} \sum_{(x_i, y_i) \in D} L(f_\theta(x_i), y_i), \quad (3.1)$$

де L – функція втрат (зокрема, крос-ентропія або метрика подібності між згенерованим та еталонним повідомленням).

Після завершення навчання, для нового прикладу $x^{test} \in X$, модель генерує повідомлення:

$$y^{test} = f_{\theta^{opt}}(x^{test}). \quad (3.2)$$

Трансформерні моделі, що застосовуються для генерації тексту, використовують авторегресійний механізм декодування, при якому кожен наступний токен вихідної послідовності генерується з урахуванням усіх попередньо згенерованих токенів та вхідної послідовності. Умовна ймовірність повідомлення факторизується відповідно до ланцюгового правила:

$$P(y|x; \theta) = \prod_{t=1}^T P(y_t | y_{<t}, x; \theta), \quad (3.3)$$

де y_t – токен повідомлення на позиції t ; $y_{<t}$ – послідовність усіх попередніх токенів повідомлення; T – довжина згенерованого повідомлення. Така факторизація дозволяє моделі послідовно генерувати кожне слово, враховуючи як контекст вхідних змін, так і вже згенеровану частину повідомлення.

Для навчання моделі використовується функція крос-ентропійних втрат, яка обчислюється як середнє від'ємне логарифмічне правдоподібність еталонних токенів за всіма прикладами навчального корпусу:

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^N \sum_{t=1}^{|y_i|} \log P_{\theta}(y_{i,t} | y_{i,<t}, x_i), \quad (3.4)$$

де $y_{i,t}$ – t -й токен еталонного повідомлення y_i ; P_{θ} – умовна ймовірність, що визначається моделлю з параметрами θ . Мінімізація цієї функції відповідає максимізації правдоподібності еталонних повідомлень, що спонукає модель генерувати тексти, наближені до людських описів змін.

Під час навчання застосовується стратегія примусового навчання (teacher forcing), за якої на кожному кроці декодування замість попередньо згенерованого токена подається відповідний токен еталонного повідомлення. Це прискорює збіжність та стабілізує процес навчання, хоча може призводити до розходження між умовами навчання та генерації (exposure bias).

Після навчання генерація повідомлення для нового прикладу здійснюється шляхом авторегресійного декодування. На кожному кроці модель обирає токен із найвищою умовною ймовірністю (жадібне декодування) або застосовує метод пошуку пучком (beam search), який зберігає кілька найбільш імовірних гіпотез на кожному кроці для підвищення якості генерації. Формально, при жадібному декодуванні вихідне повідомлення визначається як:

$$\hat{y} = \arg \max_y P(y|x; \theta). \quad (3.5)$$

Для оцінювання якості згенерованих повідомлень використовується набір метрик, що вимірюють подібність між згенерованим та еталонним повідомленням. Загальне значення метрики обчислюється як середнє за тестовою вибіркою:

$$\bar{Q} = \frac{1}{N_{test}} \sum_{i=1}^{N_{test}} M(\hat{y}_i, y_i), \quad (3.6)$$

де M – функція оцінювання якості (наприклад, BLEU, ROUGE, BERTScore або BLEURT) [46, 49–51]; $\hat{y}_i$ – повідомлення, згенероване моделлю для i -го тестового прикладу; y_i – відповідне еталонне повідомлення; N_{test} – обсяг тестової вибірки. Застосування різнотипних метрик (лексичних та семантичних) дозволяє

комплексно оцінити якість генерації як на рівні поверхневих збігів, так і на рівні семантичної відповідності.

Архітектурно задача умовної генерації повідомлень реалізується на основі моделей типу кодувальник-декодувальник (encoder-decoder). Кодувальник обробляє вхідну послідовність токенів змін у вихідних текстах та формує їх контекстне представлення у вигляді послідовності прихованих станів. Декодувальник, у свою чергу, авторегресійно генерує вихідну послідовність токенів повідомлення, використовуючи механізм перехресної уваги (cross-attention) для врахування інформації від кодувальника на кожному кроці генерації. Така архітектура лежить в основі моделей T5, CodeT5, CodeT5+, FLAN-T5 та BART, які застосовуються у цьому дослідженні для генерації повідомлень про внесені зміни.

Суттєвою перевагою трансформерних моделей є можливість застосування трансферного навчання (transfer learning): моделі, попередньо навчені на великих корпусах тексту або вихідного коду, донавчаються (fine-tuning) на спеціалізованому корпусі пар «зміни у коді – повідомлення». Під час донавчання оптимізуються всі параметри моделі або їх частина з використанням алгоритму AdamW та лінійного планувальника швидкості навчання (linear learning rate scheduler), [69] що забезпечує стабільну збіжність та запобігає катастрофічному забуванню попередньо набутих знань.

Таким чином, у підрозділі формалізовано задачу автоматичної генерації повідомлень про внесені зміни як задачу умовної генерації тексту. Визначено вхідний та вихідний простори, корпус навчальних даних, параметризовану трансформерну модель та цільову функцію оптимізації. Описано авторегресійний механізм декодування, функцію крос-ентропійних втрат, стратегію примусового навчання та підходи до декодування. Введено формальний апарат для оцінювання якості згенерованих повідомлень. Наведена формалізація є основою для розроблення конкретних методів генерації, які розглядаються у наступних підрозділах.

3.2 Архітектура моделей генерації повідомлень (T5, FLAN-T5, CodeT5, CodeT5+, BART)

У підрозділі 3.1 задачу автоматичної генерації повідомлень про внесені зміни формалізовано як задачу умовної генерації тексту. Для її розв'язання у дисертаційному дослідженні застосовується сімейство трансформерних моделей з архітектурою кодувальник-декодувальник (encoder-decoder), що включає: T5, FLAN-T5, CodeT5, CodeT5+ та BART. Ці моделі попередньо навчені на великих корпусах текстових або програмних даних і донавчаються на спеціалізованому корпусі пар «зміни у коді – повідомлення». Цей підрозділ містить опис архітектурних особливостей зазначених моделей та обґрунтування їхнього вибору для задачі генерації.

3.2.1 Особливості архітектури encoder-decoder трансформерних моделей

Усі моделі, що застосовуються у дослідженні, ґрунтуються на архітектурі encoder-decoder, запропонованій у роботі [74]. Кодувальник перетворює вхідну послідовність токенів змін у коді на послідовність контекстних представлень, а декодувальник авторегресійно генерує вихідну послідовність токенів повідомлення. Кожен блок кодувальника та декодувальника складається з механізму багатоголової уваги, прямого з'єднання (feedforward network) та нормалізації шару із залишковими зв'язками.

Базовим елементом трансформера є механізм масштабованої скалярної уваги (scaled dot-product attention). Для матриць запитів Q , ключів K та значень V обчислення уваги визначається як:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \quad (3.7)$$

де d_k – розмірність ключів; масштабування на $\sqrt{d_k}$ запобігає зникненню градієнтів при великих значеннях скалярного добутку. Функція softmax перетворює оцінки подібності на нормалізовані ваги уваги, що визначають внесок кожного елемента послідовності до підсумкового представлення.

Механізм багатоголової уваги (Multi-Head Attention, МНА) дозволяє паралельно обчислювати увагу у h незалежних підпросторах представлень, що забезпечує моделювання різних типів залежностей одночасно:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W_O, \quad (3.8)$$

де кожна голова обчислюється як:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V), \quad (3.9)$$

де W_i^Q , W_i^K , W_i^V – проєкційні матриці i -ї голови; W_O – вихідна проєкційна матриця. Кожен блок кодувальника також містить дворівневу повнозв'язну мережу (Feedforward Network, FFN), що застосовується незалежно до кожного токена:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2, \quad (3.10)$$

де W_1 , W_2 – матриці ваг; b_1 , b_2 – вектори зміщень. Виходи механізму уваги та FFN обробляються із використанням залишкових зв'язків та нормалізації шару:

$$h_l = \text{LN}(h_{l-1} + \text{Sublayer}(h_{l-1})), \quad (3.11)$$

де $\text{LN}(\cdot)$ – нормалізація шару (Layer Normalization); Sublayer – підшар МНА або FFN. Залишкові зв'язки запобігають деградації градієнта при збільшенні глибини мережі, а нормалізація стабілізує розподіл активацій.

Ключовою відмінністю архітектури encoder-decoder від encoder-only моделей (BERT, RoBERTa), описаних у підрозділі 2.4, є наявність у декодувальника додаткового шару перехресної уваги (cross-attention). Цей механізм забезпечує передачу інформації від кодувальника до декодувальника на кожному кроці генерації:

$$\text{CrossAttn}(Q_{\text{dec}}, K_{\text{enc}}, V_{\text{enc}}) = \text{softmax}\left(\frac{Q_{\text{dec}}K_{\text{enc}}^T}{\sqrt{d_k}}\right)V_{\text{enc}}, \quad (3.12)$$

де Q_{dec} – запити декодувальника, сформовані з поточного прихованого стану; K_{enc} , V_{enc} – ключі та значення, отримані з виходів кодувальника. Таким чином, при генерації кожного токена повідомлення декодувальник може динамічно зосереджуватись на різних фрагментах вхідних змін у коді, що є критично важливим для формування змістовних описів.

Модель T5 (Text-to-Text Transfer Transformer), розроблена компанією Google, реалізує [77] уніфікований підхід до задач обробки природної мови, формулюючи кожен задачу як перетворення «текст у текст». Під час попереднього навчання T5 використовує стратегію маскування фрагментів (span corruption): випадковий набір послідовних токенів у вхідному тексті замінюється на спеціальний маркер, а модель навчається відновлювати замасковані фрагменти. Цільова функція попереднього навчання визначається як:

$$\mathcal{L}_{\text{span}} = - \sum_{t=1}^T \log P_{\theta}(y_t | y_{<t}, x_{\text{masked}}), \quad (3.13)$$

де x_{masked} – вхідна послідовність із замаскованими фрагментами; y_t – токен цільової послідовності (відновлюваного фрагмента). Така стратегія дозволяє моделі навчитися контекстному передбаченню, що є безпосередньо корисним для задачі генерації повідомлень.

Модель T5 представлена у кількох конфігураціях, що відрізняються кількістю параметрів: t5-base (~220 млн) та t5-large (~770 млн). Обидві конфігурації використовують словник розміром 32 тисячі токенів, побудований за алгоритмом SentencePiece [78].

Модель FLAN-T5 (Fine-tuned Language Net) є вдосконаленою версією T5, додатково донавченою [79] на великій колекції задач, сформульованих у вигляді текстових інструкцій (instruction tuning). На відміну від базової T5, FLAN-T5 навчена виконувати завдання відповідно до текстових вказівок, що забезпечує кращу узагальненість та здатність адаптуватися до нових задач без додаткового донавчання. У контексті генерації повідомлень інструкційне налаштування дозволяє формувати вхідні дані у вигляді шаблонів, наприклад: «Generate a commit message for the following code changes:», що підвищує якість генерації.

Модель BART (Bidirectional and Auto-Regressive Transformers), розроблена компанією Meta, є [80] автокодувальником із шумозаглушенням (denoising autoencoder). На відміну від T5, яка використовує лише маскування фрагментів, BART застосовує різноманітні перетворення зашумлення (спотворення) вхідного тексту, зокрема маскування та видалення токенів, перестановку речень, обертання

документа і заміну фрагментів тексту. Кодувальник BART є двонаправленим, а декодувальник – авторегресійним. Цільова функція попереднього навчання визначається як:

$$\mathcal{L}_{\text{BART}} = - \sum_{t=1}^T \log P_{\theta}(x_t | x_{<t}, g(x)), \quad (3.14)$$

де $g(x)$ – спотворена версія вхідної послідовності, отримана застосуванням стохастичних перетворень. Модель навчається відновлювати оригінальну послідовність x . Конфігурація bart-large містить ~400 млн параметрів та використовує словник розміром 50 тисяч токенів. Широкий набір стратегій пошкодження забезпечує BART стійкість до різних типів шуму у вхідних даних, що є корисним при обробці технічних текстів змін у коді.

3.2.2 Моделі, орієнтовані на вихідні тексти програм

Загальномовні моделі (T5, FLAN-T5, BART), попередньо навчені на корпусах природної мови, не враховують специфіку вихідних текстів програм: структуру ідентифікаторів, синтаксичні конструкції мов програмування, семантику змін у коді. Для подолання цього обмеження компанією Salesforce розроблено спеціалізовані моделі CodeT5 та CodeT5+, архітектура [9, 10] яких адаптована до роботи з програмним кодом.

Модель CodeT5 побудована на основі архітектури T5 та попередньо навчена на великому корпусі вихідних текстів програм, що включає дані з відкритих репозиторіїв на кількох мовах програмування. Ключовою відмінністю CodeT5 від базової T5 є застосування задачі розпізнавання ідентифікаторів (identifier-aware pretraining): під [9] час попереднього навчання модель додатково навчається розрізняти ідентифікатори у вихідному коді (назви змінних, функцій, класів) та передбачати їх на основі оточуючого контексту. Це забезпечує глибше розуміння семантики програмного коду порівняно з моделями, навченими лише на текстах природної мови.

Модель CodeT5 представлена у трьох конфігураціях: codet5-small (~60 млн параметрів), codet5-base (~220 млн) та codet5-large (~770 млн). Усі конфігурації

використовують словник розміром 32 тисячі токенів, оптимізований для токенізації як природної мови, так і вихідних текстів програм.

Модель CodeT5+ є вдосконаленою версією CodeT5, що поєднує переваги попереднього навчання на вихідних текстах програм з інструкційним налаштуванням (instruction tuning) в умовах багатозадачності. На відміну від CodeT5, де модель навчається на єдиній задачі маскованого моделювання, CodeT5+ додатково навчена на кількох задачах одночасно: генерація коду, сумаризація коду, переклад між мовами програмування та відповіді на запитання за кодом. Таке мультизадачне навчання забезпечує формування більш універсальних внутрішніх представлень, що покращує якість генерації при донавчанні на спеціалізованих задачах. У дослідженні застосовуються конфігурації codet5p-220m та codet5p-770m.

Основні параметри всіх моделей, що застосовуються у дослідженні для генерації повідомлень про внесені зміни, наведено у таблиці 3.1, порівняння архітектур – на рис. 3.1.

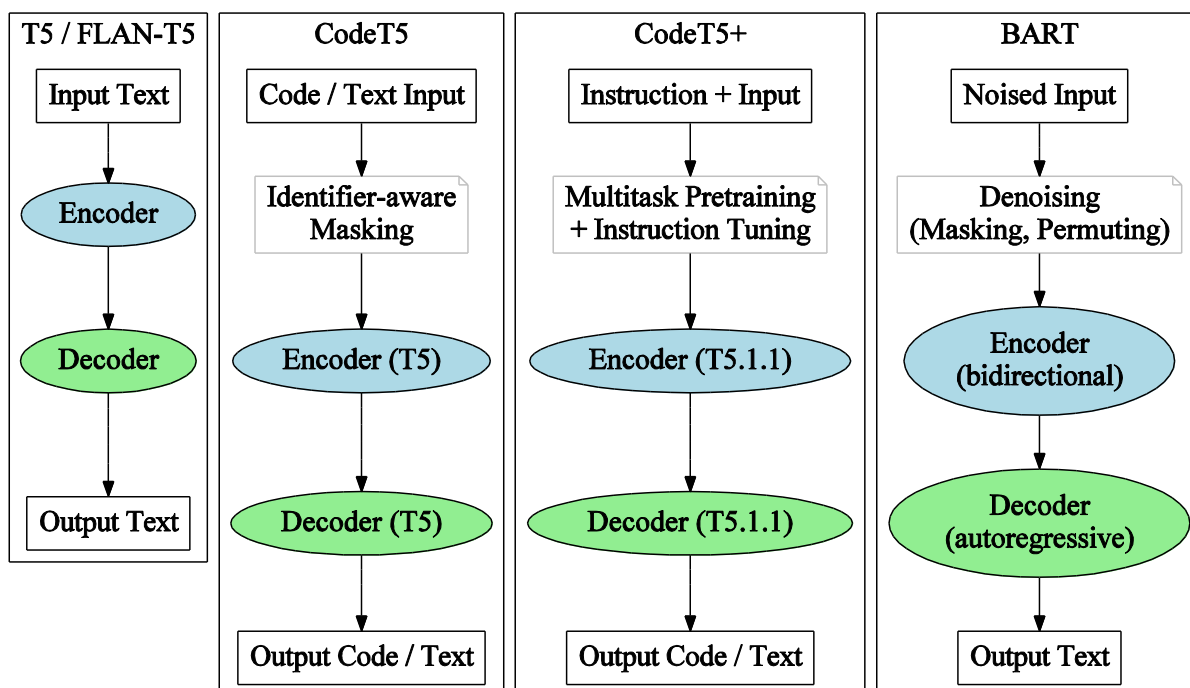


Рисунок 3.1 – Порівняння архітектур трансформерних моделей генерації повідомлень

Таблиця 3.1 – Основні параметри попередньо натренованих моделей

Назва моделі	Арх-ра	Парам., млн	Словник, тис	Компанія	Особливості
codet5-small	T5	~60	32	Salesforce	Попередньо навчена на вихідних текстах програм; швидке прототипування
codet5-base	T5	~220	32	Salesforce	Базовий варіант CodeT5; підтримує генерацію, сумаризацію
codet5-large	T5	~770	32	Salesforce	Потужніша версія CodeT5; вищі вимоги до ресурсів
codet5p-220m	T5 (CodeT5+)	~220	32	Salesforce	Покращена якість генерації завдяки інструкційному навчанню
codet5p-770m	T5 (CodeT5+)	~770	32	Salesforce	Покращена якість для генерації вихідних текстів програм
t5-base	T5	~220	32	Google	Універсальна модель; не навчена спеціально на вихідних текстах
t5-large	T5	~770	32	Google	Більша версія t5-base; кращі результати на мовних задачах
flan-t5-base	T5 (Flan)	~220	32	Google	Донавчена за інструкціями;

Назва моделі	Арх-ра	Парам., млн	Словник, тис	Компанія	Особливості
					покращена узагальненість
bart-large	BART	~400	50	Meta	Архітектура encoder-decoder; спеціалізована на переформулюванні

Моделі були обрані на підставі їхньої архітектури та ступеня адаптованості до завдань, пов'язаних із вихідними текстами програм (CodeT5, CodeT5+), або ж до завдань узагальнення та генерації тексту загального призначення (T5, FLAN-T5, BART). Усі моделі донавчаються на попередньо сформованому корпусі пар «зміна-повідомлення» для кількох мов програмування (C/C++, Ruby, Python, Java, JavaScript (JS), PHP). У процесі донавчання використовуються власні токенизатори відповідних моделей (AutoTokenizer з бібліотеки HuggingFace Transformers) [81].

Порівняння моделей за ступенем адаптації до задачі генерації повідомлень дозволяє виділити три групи. До першої належать базові моделі загального призначення (t5-base, t5-large), які не мають спеціалізованого попереднього навчання на вихідних текстах програм і слугують базовою лінією для порівняння. Другу групу складають моделі з інструкційним навчанням (flan-t5-base, bart-large), що завдяки покращеній узагальненості здатні ефективніше адаптуватися до нових задач. Третя група включає моделі, орієнтовані на вихідні тексти програм (сімейства CodeT5 та CodeT5+), які мають найбільший потенціал для задачі генерації повідомлень завдяки попередньому навчанню на програмному коді та розумінню його семантичної структури.

Таким чином, у підрозділі описано архітектурні особливості трансформерних моделей типу encoder-decoder, що застосовуються для генерації повідомлень про внесені зміни: механізм масштабованої скалярної уваги, багатоголову увагу, пряме з'єднання та перехресну увагу декодувальника. Охарактеризовано стратегії попереднього навчання моделей T5 (маскування фрагментів), FLAN-T5 (інструкційне налаштування), BART (автокодувальник із шумозаглушенням),

CodeT5 (розпізнавання ідентифікаторів) та CodeT5+ (мультизадачне інструкційне навчання). Наведено [10] основні параметри моделей та обґрунтовано їхній вибір для розв’язання задачі генерації повідомлень.

3.3 Підходи до донавчання моделей

У підрозділі 3.2 охарактеризовано архітектури трансформерних моделей, що застосовуються для генерації повідомлень про внесені зміни. Проте архітектура моделі є лише однією складовою системи генерації – не менш важливою є стратегія донавчання (fine-tuning), що визначає, яким чином попередньо навчені параметри адаптуються до цільової задачі. У цьому підрозділі описано загальну процедуру донавчання, включаючи вибір гіперпараметрів та алгоритму оптимізації, а також розглянуто два спеціалізовані підходи: інструкційне навчання та мультизадачне навчання.

Донавчання трансформерних моделей для задачі генерації повідомлень ґрунтується на парадигмі трансферного навчання (transfer learning): модель, попередньо навчена на великому корпусі загальних або програмних текстів, адаптується до спеціалізованої задачі шляхом мінімізації функції крос-ентропійних втрат на навчальному корпусі пар «зміни у коді – повідомлення». На відміну від підходу до донавчання класифікаторів, описаного у підрозділі 2.4.2, де оптимізувалась бінарна крос-ентропія для двох класів, у задачі генерації оптимізується послідовна крос-ентропія за всіма позиціями вихідної послідовності:

$$\mathcal{L}_{ft}(\theta) = -\frac{1}{N} \sum_{i=1}^N \sum_{t=1}^{|y_i|} \log P_{\theta}(y_{i,t} | y_{i,<t}, x_i), \quad (3.15)$$

де x_i – вхідна послідовність токенів (текстова репрезентація змін у коді) для i -го прикладу; y_i – відповідне еталонне повідомлення; P_{θ} – умовна ймовірність, що визначається моделлю з параметрами θ ; N – обсяг навчального корпусу. Мінімізація цієї функції еквівалентна максимізації правдоподібності еталонних повідомлень за умови спостереження відповідних змін у вихідних текстах програм.

Для оптимізації параметрів моделі використовується алгоритм AdamW – модифікація алгоритму Adam із декомпозованою регуляризацією ваг (decoupled

weight decay) [69]. На кожному кроці t обчислюються експоненціальні ковзні середні першого та другого моментів градієнта:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad (3.16)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \quad (3.17)$$

де g_t – градієнт функції втрат за параметрами θ на кроці t ; $\beta_1 = 0,9$ та $\beta_2 = 0,999$ – коефіцієнти згасання моментів. Оновлення параметрів здійснюється за формулою:

$$\theta_{t+1} = \theta_t - \eta \left(\frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \varepsilon} + \lambda \theta_t \right) \quad (3.18)$$

де $\widehat{m}_t$ та $\widehat{v}_t$ – скориговані оцінки моментів; η – швидкість навчання; $\varepsilon = 10^{-8}$ – стабілізуючий параметр; λ – коефіцієнт регуляризації ваг (weight decay). Ключовою перевагою AdamW над класичним Adam є те, що регуляризація ваг $\lambda \theta_t$ додається безпосередньо до кроку оновлення, а не до градієнта, що забезпечує коректнішу регуляризацію та кращу узагальненість моделі.

Для управління швидкістю навчання застосовується лінійний планувальник (linear learning rate scheduler) із фазою розігріву (warmup). На початку навчання швидкість лінійно зростає від нуля до початкового значення протягом фази розігріву, а потім лінійно зменшується до нуля протягом залишку навчання:

$$\eta_t = \eta_0 \cdot \min \frac{t}{T_w}, \frac{T - t}{T - T_w} \quad (3.19)$$

де η_0 – початкова (максимальна) швидкість навчання; T_w – кількість кроків розігріву; T – загальна кількість кроків навчання. Фаза розігріву запобігає різкому оновленню попередньо навчених ваг на початку донавчання, а поступове зменшення швидкості стабілізує збіжність на пізніх етапах.

Донавчання усіх моделей у дослідженні здійснюється з наступними фіксованими гіперпараметрами:

- розмір мініпакету (batch size) – 4 або 8 (залежно від обсягу доступної пам'яті GPU);
- максимальна кількість епох – 50 із ранньою зупинкою (patience = 5);

- початковий коефіцієнт навчання $\eta_0 = 5 \cdot 10^{-5}$;
- коефіцієнт регуляризації ваг (weight decay) $\lambda = 0,01$.

Навчання виконується у середовищі Google Colaboratory з використанням GPU NVIDIA A100 (40 ГБ пам'яті) [76].

Для запобігання перенавчанню застосовується стратегія ранньої зупинки (early stopping): після кожної епохи обчислюється функція втрат на валідаційній вибірці, і навчання припиняється, якщо протягом заданої кількості послідовних епох (patience) не спостерігається зменшення валідаційних втрат. Оптимальні параметри визначаються як:

$$\theta^* = \theta_{e^*}, e^* = \arg \min_e \mathcal{L}_{\text{val}}(\theta_e), \quad (3.20)$$

де θ_e – параметри моделі після епохи e ; $\mathcal{L}_{\text{val}}$ – функція втрат на валідаційній вибірці. Таким чином, зберігається стан моделі, що відповідає найменшим валідаційним втратам, що забезпечує вибір оптимального моменту зупинки навчання.

3.3.1 Інструкційне навчання (instruction tuning)

Інструкційне навчання (instruction tuning) є підходом до донавчання мовних моделей, при якому вхідні дані кожної задачі доповнюються текстовою інструкцією, що описує бажану поведінку моделі природною мовою. На відміну від стандартного донавчання, де модель отримує лише пару «вхід-вихід», при інструкційному навчанні вхідна послідовність збагачується текстовим префіксом, що явно вказує на характер задачі. Цей підхід вперше масштабно застосовано у сімействі моделей FLAN (Fine-tuned Language Net) компанії Google та набув [79] подальшого розвитку у моделях CodeT5+.

Формально, нехай I – текстова інструкція (текстовий шаблон), що описує задачу, а x – вхідна послідовність (текстова репрезентація змін у коді). Інструкційне перетворення вхідних даних визначається як конкатенація інструкції та вхідної послідовності:

$$x' = \text{Concat}(I, x), \quad (3.21)$$

де x' – збагачена вхідна послідовність; Concat – операція конкатенації. Наприклад, для задачі генерації повідомлень інструкція може мати вигляд: «*Generate a commit message for the following code changes:*», після чого розміщується безпосередньо текст змін у коді.

Ключовою перевагою інструкційного навчання є підвищення узагальненості моделі: навчена на великій колекції різнотипних задач з інструкціями, модель набуває здатності виконувати нові задачі на основі текстового опису без додаткового донавчання (zero-shot generalization). У контексті моделі FLAN-T5 інструкційне навчання реалізується як додатковий етап після попереднього навчання T5: модель T5, попередньо навчена на задачі маскування фрагментів, додатково донавчається на колекції з понад 1800 задач, сформульованих у вигляді текстових інструкцій. Це охоплює задачі сумаризації, відповідей на запитання, перекладу, класифікації та генерації тексту.

Функція втрат при інструкційному навчанні зберігає форму крос-ентропії, але застосовується до збагачених вхідних послідовностей:

$$\mathcal{L}_{\text{inst}}(\theta) = -\frac{1}{|D_{\text{inst}}|} \sum_{i=1}^{|D_{\text{inst}}|} \sum_{t=1}^{|y_i|} \log P_{\theta}(y_{i,t} | y_{i,<t}, x_i'), \quad (3.22)$$

де D_{inst} – навчальний корпус, у якому кожен вхідний приклад збагачений відповідною інструкцією; x_i' – конкатенація інструкції та i -ї вхідної послідовності. Принципова відмінність від формули (3.15) полягає у тому, що модель навчається генерувати вихідну послідовність, орієнтуючись не лише на зміст вхідних змін, а й на текстову вказівку щодо бажаного формату та характеру вихідного повідомлення.

Для моделей CodeT5+ інструкційне навчання поєднується з попереднім навчанням на вихідних текстах програм: модель спочатку навчається розуміти структуру програмного коду (identifier-aware pretraining), а потім додатково донавчається на колекції задач з інструкціями, специфічних для програмної інженерії. Такий двоетапний підхід забезпечує синергію між розумінням семантики коду та здатністю слідувати текстовим вказівкам.

Експериментальні результати, отримані у межах дослідження, засвідчують, що моделі з інструкційним навчанням (FLAN-T5, CodeT5+) демонструють вищу

здатність до генерації узгоджених повідомлень, релевантних до контексту змін, порівняно з базовими моделями T5 без інструкційного налаштування. Зокрема, модель codet5p-770m досягає найвищих показників серед усіх досліджуваних моделей, що підтверджує ефективність поєднання інструкційного навчання з попереднім навчанням на програмному коді.

3.3.2 Мультизадачне навчання (multi-task learning)

Мультизадачне навчання (multi-task learning) є підходом, [82] при якому модель одночасно навчається на кількох пов'язаних задачах, що дозволяє формувати більш універсальні та узагальнені внутрішні представлення. На відміну від однозадачного донавчання, де параметри моделі оптимізуються виключно для одної цільової задачі, мультизадачний підхід забезпечує регуляризацію через різноманітність навчальних сигналів та зменшує ризик перенавчання на обмеженому корпусі задачно-специфічних даних.

Формально, нехай задано K пов'язаних задач, кожна з яких визначається власним навчальним корпусом D_k , $k = 1, \dots, K$. Загальна функція втрат мультизадачного навчання визначається як зважена сума індивідуальних функцій втрат:

$$\mathcal{L}_{\text{total}}(\theta) = \sum_{k=1}^K \alpha_k \mathcal{L}_k(\theta), \quad (3.23)$$

де α_k – вагові коефіцієнти, що визначають відносну важливість k -ї задачі; $\mathcal{L}_k(\theta)$ – функція втрат для k -ї задачі. Кожна індивідуальна функція втрат обчислюється як крос-ентропія на відповідному корпусі:

$$\mathcal{L}_k(\theta) = -\frac{1}{|D_k|} \sum_{(x,y) \in D_k} \sum_{t=1}^{|y|} \log P_{\theta}(y_t | y_{<t}, x), \quad (3.24)$$

де D_k – навчальний корпус k -ї задачі, що складається з пар вхідних та цільових послідовностей. Вагові коефіцієнти α_k можуть визначатися статично (пропорційно до обсягу відповідних корпусів) або динамічно (адаптуючись під час навчання на основі швидкості збіжності кожної задачі).

У контексті моделі CodeT5+ мультизадачне навчання реалізується шляхом одночасного навчання на кількох задачах програмної інженерії: генерації коду за текстовим описом, сумаризації вихідного коду, перекладу між мовами програмування, заповнення пропущених фрагментів коду та відповідей на запитання за кодом. Така різноманітність задач забезпечує формування внутрішніх представлень, які одночасно враховують синтаксичні конструкції мов програмування, семантику операцій та зв'язок між кодом і природною мовою.

Для задачі генерації повідомлень про внесені зміни мультизадачне попереднє навчання CodeT5+ забезпечує три ключові переваги. По-перше, задача сумаризації коду безпосередньо релевантна до генерації повідомлень, оскільки обидві задачі передбачають створення короткого текстового опису фрагмента вихідного коду. По-друге, задача генерації коду за описом формує зворотне відображення «текст-код», що посилює розуміння зв'язку між змінами у коді та їх вербальними описами. По-третє, задача перекладу між мовами програмування розвиває здатність моделі узагальнювати програмні концепції незалежно від конкретної синтаксичної реалізації.

Важливою практичною перевагою мультизадачного підходу є зменшення потреби у великих задачно-специфічних навчальних корпусах. Оскільки модель набуває загального розуміння зв'язку між кодом і текстом під час мультизадачного попереднього навчання, донавчання на відносно невеликому корпусі пар «зміна-повідомлення» виявляється достатнім для досягнення високої якості генерації. Це пояснює, чому моделі CodeT5+ з мультизадачним навчанням перевершують за якістю генерації базові моделі T5 аналогічного розміру, навіть при однакових умовах донавчання.

Порівнюючи описані підходи, слід зазначити, що інструкційне та мультизадачне навчання не є взаємовиключними, а, навпаки, ефективно доповнюють одне одного. Модель CodeT5+ поєднує обидва підходи: мультизадачне попереднє навчання на кількох задачах програмної інженерії забезпечує формування універсальних представлень, тоді як інструкційне форматування вхідних даних дозволяє точніше спрямовувати генерацію під час

донавчання та інференсу. Стандартне донавчання без інструкцій застосовується для базових моделей T5 та CodeT5, що слугують базовими лініями для порівняння.

Таким чином, у підрозділі описано процедуру донавчання трансформерних моделей для задачі генерації повідомлень про внесені зміни. Визначено цільову функцію оптимізації (крос-ентропійні втрати), алгоритм оптимізації (AdamW із лінійним планувальником швидкості навчання) та фіксовані гіперпараметри навчання. Охарактеризовано два спеціалізовані підходи до донавчання: інструкційне навчання, що збагачує вхідні дані текстовими вказівками та підвищує узагальненість моделі, та мультизадачне навчання, що формує універсальні внутрішні представлення шляхом одночасного навчання на кількох пов'язаних задачах. Показано, що поєднання обох підходів у моделі CodeT5+ забезпечує найвищу якість генерації серед досліджуваних моделей.

3.4 Метод мовно-специфічного донавчання для систем контролю версій

Попередні підрозділи описали загальні підходи до донавчання трансформерних моделей для задачі генерації повідомлень про внесені зміни. Однак практика розробки програмного забезпечення свідчить про те, що характеристики мови програмування суттєво впливають на структуру змін у вихідних текстах програм, а отже – і на якість автоматично згенерованих повідомлень. Різні мови програмування відрізняються синтаксичними конструкціями, парадигмами розробки, типовими патернами іменування, рівнем багатослівності та стилістичними конвенціями оформлення змін. Ці відмінності створюють передумови для мовно-специфічного донавчання, при якому модель адаптується до особливостей конкретної мови програмування, що потенційно дозволяє підвищити якість генерації порівняно з універсальним підходом.

У цьому підрозділі представлено метод мовно-специфічного донавчання трансформерних моделей, що передбачає формування окремих навчальних корпусів для кожної мови програмування та проведення незалежного донавчання моделей на кожному з цих корпусів. Такий підхід дозволяє моделі засвоїти синтаксичні та семантичні особливості конкретної мови [83], що, як показують

експериментальні результати, призводить до суттєвого покращення якості генерації повідомлень.

3.4.1 Формування мовно-специфічних корпусів (C/C++, Ruby, Python, Java, JavaScript, PHP)

Формування мовно-специфічних корпусів є ключовим етапом запропонованого методу. Для кожної мови програмування створюється окремий навчальний корпус, що складається з пар «змiна-повідомлення», відібраних із репозиторіїв, у яких домінує відповідна мова. Процедура формування корпусів базується на інфраструктурі збору даних через GitHub REST API, описаній у підрозділі 3.1, та методі фільтрації повідомлень на основі рекурентних нейронних мереж, представленому у розділі 2.

Формально, для кожної мови програмування $lang \in L$, де $L = \{C/C++, Ruby, Python, Java, JavaScript, PHP\}$, формується окремий навчальний корпус:

$$D_{lang} = \{(x_i, y_i)\}_{i=1}^{N_{lang}} \quad (3.25)$$

де x_i – текстова репрезентація змін у вихідному коді (git diff); y_i – відповідне повідомлення про внесені зміни; N_{lang} – кількість прикладів у корпусі для мови $lang$.

Процедура формування мовно-специфічних корпусів складається з таких етапів. На першому етапі здійснюється ідентифікація мови програмування для кожного коміту на основі розширень файлів, що входять до відповідних змін. Внесені зміни (commit) відносяться до корпусу мови, якщо більшість змінених файлів мають розширення, характерне для цієї мови (наприклад, .py для Python, .java для Java, .js для JavaScript, .rb для Ruby, .php для PHP, .c/.cpp/.h для C/C++).

На другому етапі виконується фільтрація повідомлень відповідно до критеріїв якості, визначених у підрозділі 2.1. Структурні правила (обмеження на довжину заголовка, використання dokonаної форми дієслів) перевіряються програмно, тоді як правило змістовності (правило 2.г) перевіряється за допомогою навченої моделі бінарної класифікації на основі двонаправленого GRU з 24 нейронами [56].

На третьому етапі здійснюється токенизація вхідних даних із використанням токенизатора відповідної трансформерної моделі (AutoTokenizer з бібліотеки HuggingFace Transformers) [81]. Максимальна довжина вхідної послідовності обмежується 512 токенами, а цільової послідовності – 128 токенами, що відповідає типовим параметрам для задач сумаризації коду.

Таблиця 3.2 – Розподіл пар «зміна-повідомлення» для навчання та валідації за мовами програмування

Мова програмування	Навчальна вибірка	Валідаційна вибірка
C/C++	149 357	29 871
Ruby	28 707	5 741
Python	34 473	6 894
Java	27 339	5 467
JavaScript	16 876	3 375
PHP	14 752	2 950

Як видно з таблиці 3.2, обсяги корпусів суттєво різняться між мовами: корпус C/C++ є найбільшим (149 357 навчальних пар), тоді як корпус PHP – найменшим (14 752 пар). Така нерівномірність відображає реальний розподіл активності у відкритих репозиторіях на платформі GitHub та є важливим фактором, що впливає на результати донавчання. Для кожної мови 20% корпусу виділяється у валідаційну вибірку.

Донавчання здійснюється незалежно для кожної мови: для мови *lang* модель f_{θ} ініціалізується з попередньо навчених ваг та оптимізується на корпусі D_{lang} з використанням крос-ентропійної функції втрат:

$$\mathcal{L}_{lang}(\theta) = -\frac{1}{|D_{lang}|} \sum_{(x,y) \in D_{lang}} \sum_{t=1}^{|y|} \log P_{\theta}(y_t | y_{<t}, x), \quad (3.26)$$

де D_{lang} – навчальний корпус для конкретної мови програмування; y_t – t -й токен цільового повідомлення; $y_{<t}$ – попередні токени цільової послідовності. Гіперпараметри навчання залишаються фіксованими для всіх мов: оптимізатор AdamW з початковою швидкістю навчання $5 \times 10_{-5}$, вагове затухання 0.01,

лінійний планувальник швидкості навчання, розмір мініпакету 4 або 8, максимум 50 епох із ранньою зупинкою (витримка 5 епох).

3.5 Метод семантичного збагачення ідентифікаторів вихідного коду через автоматичне розкриття аббревіатур

У попередніх підрозділах було продемонстровано ефективність трансформерних моделей для задачі генерації повідомлень про внесені зміни, а також описано метод мовно-специфічного донавчання, що враховує особливості конкретних мов програмування. Однак якість генерації залежить не лише від архітектури моделі та стратегії навчання, а й від інформативності вхідних даних, що подаються на вхід генератора. Одним із факторів, що знижують семантичну насиченість вхідних текстових представлень змін, є широке використання аббревіатур та скорочень в ідентифікаторах вихідного коду.

Ідентифікатори у вихідних текстах програм – імена змінних, функцій, класів, методів та констант – несуть значну частину семантичної інформації про призначення та поведінку програмних компонентів. Водночас розробники часто використовують скорочення (наприклад, «cfg» замість «configuration», «mgr» замість «manager», «str» замість «string», «auth» замість «authentication»), що є зрозумілими в контексті конкретного проєкту, але суттєво ускладнюють автоматичну обробку коду моделями генерації природної мови. Система генерації повідомлень, отримуючи на вхід фрагменти коду зі скороченими ідентифікаторами, може втрачати важливі семантичні підказки, що призводить до генерації надто загальних або неточних описів змін.

У цьому підрозділі запропоновано метод семантичного збагачення ідентифікаторів вихідного коду через автоматичне розкриття аббревіатур. Інтеграція цього методу в етап препроцесингу коду перед генерацією повідомлень підвищує виразність та інформативність текстових представлень, що подаються на вхід трансформерних моделей, та, як показують результати експериментальної оцінки, призводить до вимірюваного покращення метрик якості генерації.

3.5.1 Формалізація методу семантичного збагачення

Нехай вхідний текст змін x_i представлений як послідовність токенів, що включає ідентифікатори, ключові слова мови програмування, оператори та літерали:

$$x_i = \{t_1, t_2, \dots, t_m\}, \quad t_j \in V, \quad (3.27)$$

де t_j – окремий токен вхідної послідовності; V – словник, що включає аббревіатури, акроніми, зарезервовані слова мови програмування та інші лексичні одиниці; m – кількість токенів у текстовому представленні змін.

Ключовим компонентом запропонованого методу є відображення розкриття аббревіатур, що трансформує скорочені ідентифікатори у їхні повні семантично виразні форми:

$$E: V \rightarrow V', \quad (3.28)$$

де E – функція розкриття аббревіатур; V – вхідний словник, що містить скорочені форми; V' – розширений словник, що містить повні форми відповідних скорочень. Для токенів, що не є аббревіатурами (зарезервовані слова мови програмування, оператори, числові літерали), відображення є тотожним: $E(t_j) = t_j$.

Збагачений вхідний текст формується шляхом поелементного застосування відображення до кожного токена вхідної послідовності:

$$x_i' = \{E(t_1), E(t_2), \dots, E(t_m)\}, \quad (3.29)$$

Трансформерна модель f_θ з параметрами θ використовується для генерації повідомлень, апроксимуючи умовний розподіл на збагачених вхідних даних:

$$f_\theta: x_i' \rightarrow \hat{y}_i, \quad (3.30)$$

де x_i' – збагачений вхідний текст; $\hat{y}_i$ – згенерований опис змін. Модель навчається мінімізацією крос-ентропійних втрат на корпусі зі збагаченими ідентифікаторами:

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^N \sum_{t=1}^{|y_i|} \log P_\theta(y_{i,t} | y_{i,<t}, x_i'), \quad (3.31)$$

де $y_{i,t}$ – t -й токен цільового повідомлення; $y_{i,<t}$ – попередні токени цільової послідовності; N – обсяг навчального корпусу. Якість генерації оцінюється за допомогою метрики M як середнє значення на тестовій вибірці:

$$\bar{Q} = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} M(\hat{y}_i, y_i), \quad (3.32)$$

де $\hat{y}_i$ – згенероване повідомлення; y_i – референтне повідомлення; N_{test} – обсяг тестової вибірки. Таким чином, задача оптимізації полягає у знаходженні параметрів, що мінімізують втрати з урахуванням семантичного збагачення ідентифікаторів:

$$\theta^* = \operatorname{argmin}_{\theta} \mathcal{L}(\theta), \quad (3.33)$$

при цьому збагачення E інтегровано у вхідні дані, що забезпечує максимізацію $\bar{Q}$.

3.5.2 Конвеєр автоматичного розкриття аббревіатур в ідентифікаторах

Запропонований метод реалізується у вигляді п'ятиетапного конвеєра (pipeline), що інтегрується в процес препроцесингу вхідних даних перед подачею їх на вхід трансформерної моделі генерації [84].

Перший етап – вилучення ідентифікаторів зі змінених фрагментів коду. Ідентифікатори вилучаються шляхом побудови абстрактного синтаксичного дерева (Abstract Syntax Tree, AST). Для мови Python використовується вбудований модуль `ast` стандартної бібліотеки, який дозволяє проаналізувати структуру вихідного коду та ідентифікувати вузли, що відповідають іменам змінних, функцій, класів, методів та аргументів. Для інших мов програмування можуть застосовуватися відповідні парсери, зокрема `tree-sitter` – універсальний інкрементальний парсер, що підтримує побудову AST для широкого спектру мов програмування. На виході цього етапу формується множина ідентифікаторів, присутніх у змінених рядках коду.

Другий етап – сегментація ідентифікаторів на складові лексеми. Ідентифікатори декомпонуються відповідно до конвенцій іменування, прийнятих у

мові програмування. Підтримуються три основні стилі: `snake_case` (розділення за символом підкреслення, характерний для Python, C/C++, Ruby, PHP), `camelCase` (розділення за зміною регістру з малої літери, характерний для Java, JavaScript) та `PascalCase` (розділення за зміною регістру з великої літери, використовується для імен класів у більшості мов). Наприклад, ідентифікатор «getUsrCfg» сегментується на лексеми «get», «usr», «cfg», а ідентифікатор «db_conn_mgr» – на лексеми «db», «conn», «mgr». Цей етап є критичним для подальшого аналізу, оскільки саме окремі лексеми можуть бути абревіатурами, що потребують розкриття.

Третій етап – генерація кандидатів для розкриття абревіатур. Для кожної лексеми, ідентифікованої як потенційна абревіатура, формується множина кандидатів розкриття на основі двох взаємодоповнюючих підходів. Словниковий підхід використовує попередньо скомпільований словник пар «абревіатура – повна форма», побудований на основі загальноприйнятих скорочень у програмній інженерії (наприклад, «cfg» → «configuration», «mgr» → «manager», «conn» → «connection», «db» → «database», «src» → «source», «buf» → «buffer»). Контекстний підхід використовує попередньо навчену модель CodeBERT, яка здатна визначати [28] найбільш імовірні відповідності на основі контексту, в якому зустрічається абревіатура. CodeBERT є двомодальною моделлю, попередньо навченою як на вихідних текстах програм, так і на природній мові, що забезпечує її здатність ефективно моделювати семантичні зв'язки між скороченнями та їхніми повними формами [85].

Четвертий етап – вибір найкращого кандидата. Для вибору найбільш релевантного розкриття серед множини кандидатів застосовується метод семантичного ранжування на основі косинусної подібності у векторному просторі. Для двох векторів a та b косинусна подібність визначається як:

$$\text{sim}(a, b) = \frac{a \cdot b}{\|a\| \cdot \|b\|}, \quad (3.34)$$

Векторні представлення кандидатів та контексту отримуються за допомогою моделі CodeBERT, яка забезпечує семантично насичені ембединги для фрагментів

програмного коду та природної мови. Формально, для кожної аббревіатури обирається кандидат із найвищим значенням подібності до контексту змін:

$$E(t_j) = \operatorname{argmax}_{c \in C(t_j)} \operatorname{sim}(v_c, v_{ctx}), \quad (3.35)$$

де $C(t_j)$ – множина кандидатів для розкриття токена t_j ; v_c – векторне представлення кандидата; v_{ctx} – векторне представлення контексту змін. Додатково виконується перевірка узгодженості: якщо жоден кандидат не досягає порогового значення подібності, оригінальний токен залишається без змін, що запобігає некоректним розкриттям.

П'ятий етап – формування збагачених представлень. Розкриті ідентифікатори інтегруються у вхідні дані для моделі генерації повідомлень, замінюючи оригінальні скорочені форми. Наприклад, фрагмент коду «def get_usr_cfg(db_conn):» після збагачення перетворюється на «def get_user_configuration(database_connection):», що надає моделі генерації значно більше семантичної інформації для формулювання точного та інформативного повідомлення. Аналогічно, зміни у рядку «update_auth_mgr()» після обробки конвеєром набувають вигляду «update_authentication_manager()», що дозволяє моделі згенерувати більш конкретний опис, наприклад «Update authentication manager» замість загального «Update auth mgr».

Таким чином, п'ятиетапний конвеєр забезпечує автоматичне розкриття аббревіатур в ідентифікаторах вихідного коду та формує семантично збагачені вхідні представлення для моделі генерації повідомлень. Автоматичне розкриття аббревіатур підвищує якість семантичного простору, в якому здійснюється генерація, що дозволяє зменшити кількість некоректних або надто загальних описів та підвищити зрозумілість історії розробки для різних членів команди.

3.6 Висновки до розділу 3

У третьому розділі дисертаційної роботи розроблено методи автоматичної генерації повідомлень про внесені зміни на основі трансформерних моделей. Формалізовано задачу генерації як задачу умовної генерації тексту, обґрунтовано вибір архітектур моделей, розроблено підходи до донавчання, запропоновано метод мовно-специфічного донавчання та метод семантичного збагачення

ідентифікаторів вихідного коду. За результатами проведеної роботи сформульовано такі висновки.

1. Формалізовано задачу автоматичної генерації повідомлень про внесені зміни як задачу умовної генерації тексту. Визначено вхідний простір текстових представлень змін та вихідний простір повідомлень природною мовою, корпус навчальних даних у вигляді пар «зміна-повідомлення», параметризовану трансформерну модель типу кодувальник-декодувальник та цільову функцію оптимізації на основі крос-ентропійних втрат. Описано авторегресійний механізм декодування з факторизацією умовної ймовірності за ланцюговим правилом, стратегію примусового навчання (teacher forcing) та методи декодування (жадібне та пошук пучком). Введено формальний апарат для оцінювання якості генерації із застосуванням як лексичних (BLEU, ROUGE), так і семантичних (BERTScore, BLEURT) метрик.

2. Обґрунтовано вибір та охарактеризовано архітектури дев'яти трансформерних моделей для генерації повідомлень: T5 (base, large), FLAN-T5 (base), CodeT5 (small, base, large), CodeT5+ (220M, 770M) та BART (large). Описано загальну архітектуру encoder-decoder з механізмом багатоголової уваги та перехресної уваги (cross-attention), що забезпечує врахування контексту вхідних змін на кожному кроці генерації. Показано, що моделі, орієнтовані на вихідні тексти програм (CodeT5, CodeT5+), мають суттєву перевагу над моделями загального призначення (T5, FLAN-T5) завдяки попередньому навчанню на великих корпусах вихідного коду, що дозволяє їм ефективніше моделювати синтаксичні та семантичні особливості програмних текстів.

3. Розроблено та описано підходи до донавчання трансформерних моделей для задачі генерації повідомлень. Визначено уніфіковані гіперпараметри навчання: оптимізатор AdamW із лінійним планувальником швидкості навчання, початковою швидкістю навчання 5×10^{-5} , ваговим затуханням 0.01, максимальною кількістю епох 50 та ранньою зупинкою з витримкою 5 епох. Охарактеризовано два спеціалізовані підходи: інструкційне навчання (instruction tuning), що збагачує вхідні дані текстовими вказівками та підвищує узагальненість моделі, та

мультизадачне навчання (multi-task learning), що формує універсальні внутрішні представлення шляхом одночасного навчання на кількох пов'язаних задачах. Показано, що поєднання обох підходів у моделі CodeT5+ забезпечує найвищу якість генерації серед досліджуваних моделей, оскільки дозволяє ефективно поєднати контекстуальне розуміння інструкцій з узагальненими представленнями вихідного коду.

4. Запропоновано метод мовно-специфічного донавчання трансформерних моделей для систем контролю версій. Формалізовано процедуру формування окремих навчальних корпусів для шести мов програмування (C/C++, Ruby, Python, Java, JavaScript, PHP) та визначено функцію крос-ентропійних втрат для мовно-специфічної оптимізації. Описано триетапну процедуру формування корпусів: ідентифікація мови програмування за розширеннями файлів, фільтрація повідомлень за критеріями якості (у тому числі із застосуванням навченої моделі бінарної класифікації на основі двонаправленого GRU), та токенизація з обмеженням довжини вхідної послідовності до 512 токенів і цільової – до 128. Загальний обсяг сформованого корпусу становить 271 504 навчальні пари, з яких найбільший – для C/C++ (149 357 пар), найменший – для PHP (14 752 пар). Метод передбачає незалежне донавчання моделі на корпусі кожної мови із фіксованими гіперпараметрами, що дозволяє моделі адаптуватися до синтаксичних та семантичних особливостей конкретної мови програмування.

5. Запропоновано метод семантичного збагачення ідентифікаторів вихідного коду через автоматичне розкриття аббревіатур, спрямований на підвищення інформативності вхідних даних для моделі генерації. Визначено відображення розкриття аббревіатур $E: V \rightarrow V'$, що трансформує скорочені ідентифікатори у їхні повні семантично виразні форми, та показано його інтеграцію у функцію втрат для навчання трансформерної моделі. Розроблено п'ятиетапний конвеєр збагачення: (1) вилучення ідентифікаторів через побудову абстрактного синтаксичного дерева; (2) сегментація за конвенціями іменування (snake_case, camelCase, PascalCase); (3) генерація кандидатів розкриття на основі словникового підходу та контекстної моделі CodeBERT; (4) семантичне ранжування кандидатів за косинусною

подібністю у векторному просторі; (5) формування збагачених текстових представлень. Метод є універсальним та застосовним до всіх досліджуваних мов програмування, оскільки використання аббревіатур в ідентифікаторах є загальною практикою незалежно від мови.

Таким чином, у розділі розроблено комплекс методів для автоматичної генерації повідомлень про внесені зміни, що охоплює формалізацію задачі, вибір та обґрунтування архітектур моделей, підходи до донавчання, мовно-специфічну адаптацію та семантичне збагачення вхідних даних. Запропоновані методи утворюють цілісну систему, в якій мовно-специфічне донавчання забезпечує адаптацію моделі до особливостей конкретної мови програмування, а семантичне збагачення ідентифікаторів підвищує інформативність вхідних представлень. Поєднання цих методів із застосуванням моделей, попередньо навчених на вихідних текстах програм, створює передумови для досягнення високої якості автоматичної генерації повідомлень.

РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНИХ МЕТОДІВ

4.1 Опис експериментальних корпусів та середовища проведення досліджень

У попередніх розділах було формалізовано задачі бінарної класифікації та умовної генерації повідомлень про внесені зміни, запропоновано архітектури моделей та обґрунтовано підходи до їх навчання. Для забезпечення відтворюваності результатів та обґрунтованості висновків у цьому підрозділі подано детальний опис експериментальних корпусів даних, методики їх збору та зберігання, а також програмно-апаратного середовища, в якому здійснювались усі обчислювальні експерименти.

Експериментальна база дослідження включає два основні корпуси даних: корпус повідомлень для задачі класифікації та корпус пар «зміна-повідомлення» для задачі генерації. Обидва корпуси сформовано на основі даних, отриманих з відкритих репозиторіїв платформи GitHub за допомогою спеціального програмного інтерфейсу GitHub REST API. Вибір GitHub як єдиного джерела даних обумовлено

необхідністю забезпечення однорідності корпусів та можливістю систематичного масштабування збору даних.

4.1.1 Корпус повідомлень для задачі класифікації (GitHub REST API)

Для навчання та оцінювання моделей бінарної класифікації повідомлень про внесені зміни, описаних у розділі 2, було сформовано спеціалізований корпус даних. Збір даних здійснювався за допомогою програмного інтерфейсу GitHub REST API, який надає структурований доступ до метаданих публічних репозиторіїв, повідомлень про внесені зміни та файлових змін у форматі JSON.

Для автоматизації процесу збору було розроблено систему завантаження даних мовою програмування Python (версія 3.11.4), яка взаємодіє з наступними сервісами GitHub REST API:

1. отримання переліку репозиторіїв (GET <https://api.github.com/repositories>);
2. отримання повідомлень до внесених змін для кожного репозиторію (GET [https://api.github.com/repos/\[full_name\]/commits](https://api.github.com/repos/[full_name]/commits));
3. отримання файлових змін (diffs) для кожного повідомлення про внесені зміни (GET [https://api.github.com/repos/\[full_name\]/commits/\[sha\]](https://api.github.com/repos/[full_name]/commits/[sha]));
4. отримання повних текстів файлів за посиланням raw_url.

Усі HTTP-запити до GitHub REST API вимагають передачі стандартних заголовків авторизації (Bearer-токен) та зазначення версії API (2022-11-28). Відповіді від сервісів надходять у текстовому форматі структурованих даних JSON, що забезпечує зручний програмний розбір та подальше зберігання отриманої інформації [20].

Для організації зберігання завантажених даних було спроектовано реляційну модель бази даних на основі СКБД PostgreSQL (версія 14.8), розгорнутої на сервері під управлінням операційної системи Ubuntu Server 22.04.1. Модель даних складається з п'яти таблиць, що забезпечують структуроване зберігання всієї необхідної інформації:

1. `repos` – таблиця переліку репозиторіїв з полями: ідентифікатор (`repoid`), назва (`fullname`), опис (`descr`), стан обробки (`procst`), дата запису та оновлення;
2. `commits` – таблиця повідомлень про внесені зміни з полями: ідентифікатор (`commitid`), прив'язка до репозиторію (`repoid`), символічна послідовність (`shacommithash`), текст повідомлення (`message`), дата внесених змін (`cmtdate`), стан обробки;
3. `parents` – таблиця для зберігання ідентифікаторів попередніх внесених змін, що забезпечує реконструкцію ланцюжка «комітів»;
4. `files` – таблиця змінених файлів із зазначенням назви файлу, стану змін (`added`, `removed`, `modified`, `renamed`), кількості доданих та видалених рядків, а також текстового представлення змін (`diff`);
5. `filetext` – допоміжна таблиця для зберігання повних текстів файлів, пов'язана з таблицею `files` відношенням «один до одного».

Процес завантаження даних було реалізовано у вигляді набору Python-сценаріїв, кожен з яких відповідає за окремий етап конвеєра збору даних: завантаження переліку репозиторіїв, завантаження повідомлень про внесені зміни, завантаження файлових змін та завантаження повних текстів файлів. Для кожного етапу передбачено механізм контролю стану обробки (поле `procst`), що дозволяє відновлювати процес завантаження після збоїв без повторної обробки вже завантажених записів.

Діаграму сутностей спроектованої бази даних наведено на рис. 4.1. Діаграма відображає реляційні зв'язки між п'ятьма таблицями та типи даних кожного поля.

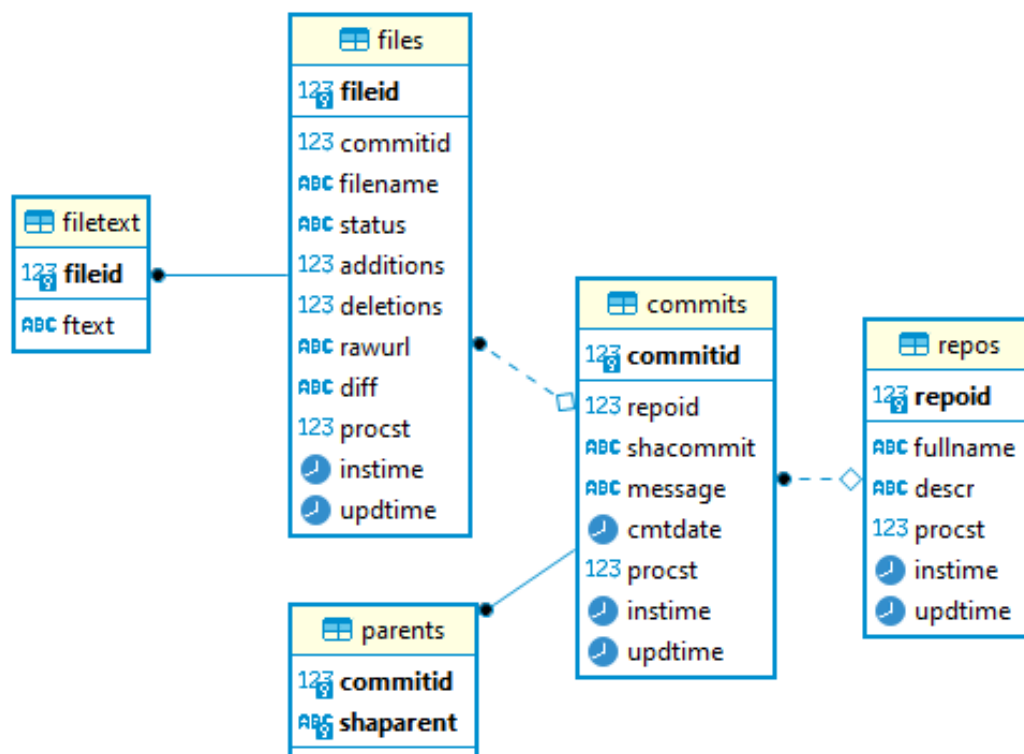


Рисунок 4.1 – Діаграма сутностей бази даних для зберігання завантажених даних

Конвеєр збору даних складається з чотирьох послідовних етапів, кожен з яких реалізовано окремим Python-сценарієм. Усі сценарії використовують спільний модуль `CommonTypes.py`, який містить параметри підключення до бази даних, перелічення станів обробки (`ProcStType`) та функцію `makeGitReq` для виконання HTTP-запитів до GitHub REST API з автоматичним перемиканням між авторизаційними токенами.

Алгоритм завантаження переліку репозиторіїв: сценарій під'єднується до бази даних та виконує SQL-запит до таблиці `repos` для визначення ідентифікатора останнього завантаженого репозиторію. Далі у циклі виконуються HTTP-запити до сервісу отримання переліку репозиторіїв із зазначенням цього ідентифікатора як початкової точки. Отримані JSON-дані розбираються та зберігаються в таблицю `repos`. Цикл повторюється доки сервіс повертає нові записи.

Алгоритм завантаження повідомлень про внесені зміни: сценарій вибирає з таблиці `repos` усі необроблені репозиторії (`procst = 0`). Для кожного репозиторію

виконується HTTP-запит до сервісу отримання повідомлень з параметром `per_page=100`. Якщо для репозиторію вже існують раніше завантажені повідомлення, запит починається із символічної послідовності (sha) останнього з них. Отримані повідомлення та інформація про попередні внесені зміни зберігаються відповідно в таблиці `commits` та `parents`. Після обробки всіх повідомлень стан обробки репозиторію змінюється на значення 1 («оброблено»).

Алгоритм завантаження файлових змін: сценарій вибирає з таблиці `commits` усі необроблені повідомлення (`commits.procst = 0`) для вже оброблених репозиторіїв (`repos.procst = 1`). Для кожного повідомлення виконується HTTP-запит до сервісу отримання деталізованої інформації про внесені зміни, який повертає перелік змінених файлів із зазначенням назви файлу, типу змін, кількості доданих та видалених рядків і текстового представлення змін (`patch`). Ці дані зберігаються в таблицю `files`.

Алгоритм завантаження повних текстів файлів: сценарій вибирає з таблиці `files` усі необроблені записи та для кожного виконує HTTP-запит за посиланням `raw_url`, отриманим на попередньому етапі. Завантажений текст файлу зберігається в таблицю `filetext`, яка пов'язана з таблицею `files` відношенням «один до одного».

Усі сценарії використовують механізм контролю стану обробки через поле `procst`, яке набуває значень: 0 – «очікує обробки», 1 – «успішно оброблено», 2 – «помилка обробки», 3 – «порожній репозиторій». Це дозволяє відновлювати роботу конвеєра після збоїв без повторної обробки вже завантажених записів та забезпечує контроль цілісності процесу збору даних.

Слід зазначити важливу технічну особливість роботи з GitHub REST API: існує обмеження у 5 000 запитів на годину для одного авторизованого користувача. Для збільшення пропускну здатності конвеєра завантаження було зареєстровано додаткові облікові записи та реалізовано механізм автоматичного перемикання між авторизаційними токенами у разі вичерпання ліміту запитів. Таким чином, сумарне обмеження було збільшено до 20 000 запитів на годину, що суттєво пришвидшило процес збору даних.

Загалом було завантажено понад 1 млн повідомлень про внесені зміни з різноманітних публічних репозиторіїв. Після завантаження виконано ручне маркування підмножини з 8 000 повідомлень відповідно до правила 2.г, визначеного у підрозділі 2.1.2, – кожному повідомленню присвоєно мітку «1» (відповідає критеріям якості) або «0» (не відповідає). Ручне маркування є трудомістким процесом, тому для його прискорення та забезпечення зручності було розроблено спеціалізований мобільний застосунок.

Мобільний застосунок «**CmtSelector**» було розроблено на платформі Android мовою програмування Kotlin. Програма забезпечує інтерфейс для оперативного перегляду та маркування повідомлень про внесені зміни безпосередньо з мобільного пристрою, що дозволяє виконувати процес розмітки у будь-якому місці за наявності підключення до мережі Інтернет. Для взаємодії мобільного застосунку зі сховищем даних (реляційна база даних PostgreSQL) застосовано PostgREST API, який інтерпретує HTTP-запити у SQL-запити, що мінімізує вразливості, пов'язані з SQL-ін'єкціями.

Застосунок «CmtSelector» має такі елементи інтерфейсу користувача:

- поле для вибору адреси сервера, яке дозволяє налаштувати IP-адресу та порт PostgREST API;
- поле для відображення тексту повідомлення про внесені зміни;
- кнопки «Попереднє повідомлення» та «Наступне повідомлення» для навігації між записами;
- кнопка «Повідомлення відповідає вимогам» – позначає повідомлення як якісне;
- кнопка «Повідомлення НЕ відповідає вимогам» – позначає повідомлення як неякісне;
- кнопки для граничних випадків («Не впевнений, і не беремо для навчання» та «Не впевнений, але беремо для навчання»);
- поле статистики, яке відображає кількість розмічених повідомлень із розбивкою за пристроями.

Взаємодія застосунку із базою даних PostgreSQL реалізована через PostgREST API. Для навігації між повідомленнями використовуються HTTP GET-запити з параметрами фільтрації за ідентифікатором та станом обробки. Наприклад, для переходу до наступного повідомлення формується запит із параметрами `commitid=gt.[поточний_id]`, `proct=lt.20` та `order=commitid.asc`. PostgREST автоматично перетворює ці HTTP-параметри в еквівалентний SQL-запит, що виключає можливість SQL-ін'єкцій.

Для збереження результатів маркування використовується збережена SQL-процедура `updateproct`, яка приймає три параметри: ідентифікатор повідомлення (`commitid`), значення відповідності вимогам (`proct`) та ім'я пристрою, з якого виконано маркування (`username`). Виклик процедури здійснюється через HTTP POST-запит до PostgREST. Також процедура оновлює SQL-представлення `cmtmsgstat`, яке агрегує статистику маркування для відображення в інтерфейсі застосунку.

Зовнішній вигляд початкового та робочого екранів мобільного застосунку «CmtSelector» наведено на рис. 4.2.

Використання мобільного застосунку суттєво пришвидшило процес ручного маркування порівняно з маркуванням через стандартний веб-інтерфейс або табличний редактор. Статистика маркування показує, що дані збирались із кількох пристроїв одночасно, що додатково підвищило продуктивність процесу.

Аналіз розподілу класів у сформованій розміченій вибірці підтвердив наявність дисбалансу: повідомлення, що не відповідають правилу 2.г, кількісно переважають якісні повідомлення. Для подолання дисбалансу застосовано техніку зсуву порогу класифікації (`threshold moving`) з коефіцієнтом 0,57, що детально описано у підрозділі 2.1.4. Розподіл вибірки для навчання та валідації здійснено у пропорції 80 % / 20 % відповідно.

4.1.2 Корпус пар «зміна-повідомлення» для задачі генерації

Для навчання та оцінювання трансформерних моделей генерації повідомлень, описаних у розділі 3, було сформовано окремий корпус пар «зміна-повідомлення», де кожен приклад складається з текстового представлення змін у

вихідному коді (git diff) та відповідного повідомлення про внесені зміни. Формування цього корпусу базується на інфраструктурі збору даних, описаній у попередньому підрозділі, з додатковими етапами фільтрації та класифікації за мовами програмування.



а)

б)

Рисунок 4.2 – Екрани застосунку «CmtSelector»:

а) початковий екран; б) робочий екран з прикладом повідомлення та статистикою

Процедура формування корпусу для задачі генерації складається з таких послідовних етапів. На першому етапі з бази даних завантажених повідомлень формуються пари (diff, message), де diff – це текстове представлення змін у файлах,

отримане з поля `patch` відповіді GitHub REST API, а `message` – відповідне повідомлення про внесені зміни.

На другому етапі виконується фільтрація повідомлень за критеріями якості, сформульованими у підрозділі 2.1.2. Структурні правила (обмеження на довжину заголовка, використання dokonanoї форми дієслів) перевіряються програмно. Правило змістовності (правило 2.г) перевіряється за допомогою навченої моделі бінарної класифікації на основі двонаправленого GRU з 24 нейронами, яка продемонструвала найкращі результати серед досліджених рекурентних архітектур (F1-score = 0,651, Accuracy = 0,808).

На третьому етапі здійснюється класифікація пар за мовами програмування на основі розширень змінених файлів. Внесені зміни відносяться до корпусу певної мови, якщо більшість змінених файлів мають розширення, характерне для цієї мови (.py для Python, .java для Java, .js для JavaScript, .rb для Ruby, .php для PHP, .c/.cpp/.h для C/C++). Такий підхід забезпечує мовно-специфічне донавчання моделей, описане у підрозділі 3.4.

Токенізація вхідних даних здійснювалась із використанням токенизатора відповідної трансформерної моделі (AutoTokenizer з бібліотеки HuggingFace Transformers). Максимальна довжина вхідної послідовності обмежувалась 512 токенами, а цільової послідовності – 128 токенами, що відповідає типовим параметрам для задач сумаризації вихідних текстів програм. Обмеження довжини послідовностей забезпечує ефективне використання пам'яті GPU та стабільність процесу навчання.

Таким чином, сформовані корпуси забезпечують комплексну експериментальну базу для дослідження як задачі класифікації повідомлень, так і задачі їх автоматичної генерації, охоплюючи широкий спектр мов програмування та типів проєктів.

4.1.3 Програмно-апаратне середовище експериментів

Усі обчислювальні експерименти, описані в цій роботі, виконано у хмарному середовищі Google Colaboratory (Google Colab). Вибір саме цього середовища обумовлено низкою суттєвих переваг для задач машинного навчання, що є

критично важливими для проведення масштабних обчислювальних експериментів [86].

Google Colaboratory надає дослідникам безкоштовний доступ до потужних обчислювальних ресурсів – графічних процесорів (GPU) та тензорних процесорів (TPU), що значно прискорює навчання моделей глибокого навчання. Хмарне середовище працює у браузері, не вимагає локального встановлення програмного забезпечення, а всі обчислення виконуються на серверах Google. Дані та написані сценарії автоматично зберігаються на хмарному носії Google Drive. Крім того, середовище підтримує інтеграцію з GitHub, що дозволяє відкривати, редагувати та зберігати вихідні тексти програм безпосередньо в репозиторіях.

Google Colab має попередньо встановлені основні бібліотеки для машинного навчання, зокрема TensorFlow, Keras, PyTorch, Scikit-learn, Pandas, NumPy та інші, що дозволяє швидко розпочати роботу без витрат часу на встановлення пакетів. Середовище використовує інтерактивні Jupyter Notebook у хмарі, що забезпечує можливість виконання Python-сценаріїв блоками, додавання візуалізацій, текстових блоків та графіків. Такий інтерактивний режим є зручним для експериментів з даними та ітеративної розробки моделей.

Як еталонну платформу для порівняння використано центральний процесор Intel Xeon з 2 обчислювальними ядрами, також доступний у середовищі Google Colab.

Для імплементації моделей класифікації (рекурентних та трансформерних) використано мову програмування Python (версія 3.10.12) та бібліотеку TensorFlow (версія 2.15.0), яка забезпечує підтримку технологій GPGPU та TPU. Вибір TensorFlow обумовлено наявністю вбудованої підтримки тензорних процесорів Google, сумісністю з середовищем Google Colab, а також широким набором високорівневих засобів для побудови, навчання та оцінювання нейронних мереж різної архітектури [66].

Для імплементації моделей генерації повідомлень використано бібліотеку HuggingFace Transformers, яка надає уніфікований інтерфейс для роботи з попередньо навченими трансформерними моделями (T5, FLAN-T5, CodeT5,

CodeT5+, BART). Ця бібліотека забезпечує автоматичне завантаження ваг моделей, спеціалізованих токенизаторів та підтримку донавчання з використанням GPU-акселерації [81].

Перелік основного програмного забезпечення, задіяного в дослідженні, наведено в таблиці 4.1.

Таблиця 4.1 – Програмне забезпечення, задіяне в дослідженні

Назва ПЗ	Версія	Призначення
Python	3.10.12 / 3.11.4	Мова сценаріїв
TensorFlow	2.15.0	Класифікація (RNN, MLP)
HuggingFace Transformers	4.36+	Генерація (T5, BART)
PyTorch	2.1+	Backend для Transformers
Scikit-learn	1.3+	Метрики оцінювання
PostgreSQL	14.8	Зберігання даних
Google Colab	—	Середовище виконання
PostgREST API	—	Інтерфейс БД – мобільний додаток
Android / Kotlin	—	Застосунок CmtSelector

Для оцінювання якості моделей класифікації використано стандартні метрики бінарної класифікації: частку правильних відповідей (Accuracy), точність (Precision), повноту (Recall) та середнє гармонійне (F1-score), визначені у розділі 1. Для оцінювання якості моделей генерації застосовано як лексичні метрики (BLEU, ROUGE-1, ROUGE-2, ROUGE-L), так і семантичні метрики (BERTScore, BLEURT), описані у підрозділі 1.5 [87–90].

Гіперпараметри навчання моделей генерації залишалися фіксованими для всіх мов програмування: оптимізатор AdamW із початковою швидкістю навчання 5×10^{-5} , вагове затухання 0,01, лінійний планувальник швидкості навчання, розмір

мініпакету 4 або 8, максимальна кількість епох – 50 із ранньою зупинкою (витримка 5 епох). Для моделей класифікації використовувався оптимізатор AdamW зі швидкістю навчання 10^{-3} та метод ранньої зупинки з контролем метрики `val_loss`.

Таким чином, описане програмно-апаратне середовище забезпечує достатні обчислювальні потужності для навчання та оцінювання усіх запропонованих моделей, а використання хмарної інфраструктури Google Colab гарантує відтворюваність результатів незалежно від локального обладнання дослідника.

4.2 Експериментальне порівняння методів оцінювання якості повідомлень

У цьому підрозділі представлено результати експериментального порівняння різних підходів до бінарної класифікації повідомлень про внесені зміни. Послідовно розглянуто три групи експериментів: порівняння рекурентних нейронних мереж за базовими метриками, дослідження впливу попередньо навчених моделей векторного представлення слів та порівняння трансформерних класифікаторів. Усі експерименти виконано на єдиному корпусі з 8 000 вручну промаркованих повідомлень, зібраних за допомогою GitHub REST API та застосунку CmtSelector, із розподілом 80/20 на навчальну та перевірну підмножини.

4.2.1 Порівняння RNN-моделей за метриками Accuracy та F1-score

Першу серію експериментів було спрямовано на порівняння семи моделей-кандидатів, побудованих на основі різних архітектур нейронних мереж: двох моделей із повнозв'язним (Dense) шаром, однієї моделі на основі одношарового LSTM, по дві моделі BiLSTM та BiGRU (одно- й двошарові варіанти). Усі моделі з рекурентними шарами мають уніфіковану структуру: `InputLayer` → `Embedding(128)` → `Dropout(0,3)` → рекурентний шар (або шари) → `Dropout(0,3)` → `Dense(2, sigmoid)`, що дозволяє ізолювати вплив типу рекурентного шару на якість класифікації. Загальна кількість нейронів рекурентної частини зафіксована на рівні 48 для всіх архітектур [26, 65, 68].

Як алгоритм оптимізації ваг обрано AdamW зі спеціальним варіантом регуляризації на основі декомпонованого вагового спаду. Для боротьби з явищем перенавчання використано шар Dropout з коефіцієнтом 0,3 та метод ранньої зупинки з контролем метрики `val_loss`. Для подолання проблеми нерівномірних

навчальних даних застосовано метод зсуву порогу (threshold moving) із коефіцієнтом 0,57 [67, 69].

Перед подачею повідомлень на вхід нейронних мереж було виконано нормалізацію (приведення літер до малих), токенизацію на навчальній вибірці та векторизацію з фіксованою довжиною вихідної послідовності та обмеженим розміром словника для прискорення обчислень.

Результати порівняння ефективності побудованих моделей наведено в таблиці 4.2. Для кожної моделі подано значення метрик Accuracy та F1-score як без застосування методу подолання нерівномірних даних, так і зі зсувом порогу.

Таблиця 4.2 – Порівняння ефективності RNN-моделей для класифікації повідомлень

Модель	Метрика	Без зсуву порогу	Зі зсувом порогу
Dense (без регуляризації)	Accuracy	0,786	0,788
Dense (без регуляризації)	F1-score	0,622	0,616
Dense (з регуляризацією)	Accuracy	0,816	0,817
Dense (з регуляризацією)	F1-score	0,648	0,637
LSTM (48 нейронів)	Accuracy	0,801	0,801
LSTM (48 нейронів)	F1-score	0,445	0,445
BiLSTM (1 шар, 24 нейрони)	Accuracy	0,796	0,798
BiLSTM (1 шар, 24 нейрони)	F1-score	0,664	0,655
BiLSTM (2 шари, 12+12)	Accuracy	0,794	0,796

BiLSTM (2 шари, 12+12)	F1-score	0,677	0,675
BiGRU (1 шар, 24 нейрони)	Accuracy	0,808	0,811
BiGRU (1 шар, 24 нейрони)	F1-score	0,651	0,629
BiGRU (2 шари, 12+12)	Accuracy	0,791	0,790
BiGRU (2 шари, 12+12)	F1-score	0,635	0,630

Аналіз результатів таблиці 4.2 дозволяє зробити такі висновки. Найвищу точність (Accuracy = 0,817 зі зсувом порогу) продемонструвала модель із повнозв'язним шаром та методами регуляризації, що свідчить про ефективність Dense-архітектури для задач обмеженого обсягу навчальних даних. Серед рекурентних архітектур найкращий результат за метрикою F1-score (0,677) має модель BiLSTM із двома шарами по 12 нейронів. Модель BiGRU з одним шаром продемонструвала найвищий показник Accuracy серед RNN-моделей (0,811) при порівняльному F1-score (0,629 зі зсувом порогу).

Одношаровий LSTM показав найнижчий F1-score серед усіх моделей (0,445), що пояснюється труднощами конвергенції при однонаправленій обробці послідовностей обмеженої довжини. Двонаправлені моделі (BiLSTM, BiGRU) стабільно перевершують одношаровий LSTM завдяки здатності враховувати контекст як зліва, так і справа від кожного токена.

Порівняння часу навчання на різних платформах показує суттєву перевагу GPU: графічний адаптер Nvidia Tesla T4 забезпечує прискорення навчання в середньому у 15 разів порівняно з TPU v2 та у 45 разів порівняно з CPU Intel Xeon. Час навчання найкращої рекурентної моделі (BiGRU, 1 шар) на GPU становить лише 1 хвилину 34 секунди.

Метод зсуву порогу з коефіцієнтом 0,57 не забезпечив значного покращення метрик, а в деяких випадках призвів до їх погіршення. Це може пояснюватися тим, що при відносно збалансованому порозі нерівномірність даних не є критичною для загальної точності, хоча зсув порогу збільшує повноту (Recall), що є пріоритетним

для фільтра: краще відхилити повідомлення та порекомендувати розробнику доповнити його, аніж пропустити неякісний опис.

4.2.2 Вплив моделей векторного представлення слів на точність класифікації

Другу серію експериментів було присвячено дослідженню впливу попередньо навчених моделей векторного представлення слів (word embeddings) на якість класифікації повідомлень про внесені зміни. У попередніх експериментах (підрозділ 4.2.1) використовувався навчальний шар Embedding, ваги якого ініціалізувалися випадковим чином та оптимізувалися у процесі навчання. У цій серії шар Embedding було замінено на «заморожені» вектори з попередньо навчених моделей, що дозволяє перенести семантичну інформацію, отриману на великих зовнішніх корпусах тексту.

Для порівняння обрано шість моделей векторного представлення слів трьох основних типів:

- Word2Vec представлено двома моделями: word2vec-google-news-300 (навчена на корпусі Google News обсягом приблизно 100 мільярдів слів, розмірність вектора 300) та SO_vectors_200 (навчена на технічних текстах Stack Overflow, розмірність 200) [91];
- FastText представлено моделлю fasttext-wiki-news-subwords-300 (навчена на Вікіпедії та новинних текстах, розмірність 300), яка враховує морфологічну структуру слів через підслівні n-грами та здатна генерувати вектори для слів, відсутніх у словнику [92];
- GloVe представлено трьома моделями: glove-twitter-200 (навчена на корпусі Twitter обсягом приблизно 2 мільярди слів, розмірність 200), glove-wiki-gigaword-50 (Вікіпедія + Gigaword, розмірність 50) та glove-wiki-gigaword-300 (Вікіпедія + Gigaword, розмірність 300) [93].

Кожну з шести моделей вбудовування було інтегровано в два типи класифікаторів: багатошаровий перцептрон (MLP) з методами регуляризації та рекурентну нейронну мережу (RNN) на основі одношарового BiGRU із 24 нейронами, що показала найкращі результати в попередній серії експериментів.

Оцінювання виконано за двома метриками: Accuracy (частка правильних відповідей) та F1-score (гармонійне середнє точності та повноти), що забезпечує комплексну оцінку якості з урахуванням дисбалансу класів. Результати порівняння наведено в таблиці 4.3.

Таблиця 4.3 – Вплив моделей векторного представлення на якість класифікації

Модель вбудовування	Тип	Розм.	Accuracy (MLP)	F1-score (MLP)	Accuracy (RNN)	F1-score (RNN)
word2vec-google-news-300	Word2Vec	300	0,856	0,624	0,855	0,626
SO_vectors_200	Word2Vec	200	0,861	0,687	0,857	0,688
fasttext-wiki-news-subwords-300	FastText	300	0,859	0,701	0,858	0,694
glove-twitter-200	GloVe	200	0,838	0,579	0,839	0,599
glove-wiki-gigaword-50	GloVe	50	0,841	0,601	0,847	0,606
glove-wiki-gigaword-300	GloVe	300	0,840	0,616	0,842	0,615

Аналіз результатів таблиці 4.3 за двома метриками – Accuracy та F1-score – виявляє кілька важливих закономірностей. По-перше, використання попередньо навчених моделей вбудовування суттєво підвищує якість класифікації порівняно з навчальним шаром Embedding з випадковою ініціалізацією. Найкраща точність із попередньо навченими векторами (Accuracy = 0,861 для SO_vectors_200 у поєднанні з MLP) перевищує найкращий результат із навчальним Embedding (0,817 для Dense з регуляризацією, підрозділ 4.2.1) на 4,4 відсоткових пункти. За метрикою F1-score приріст є ще більш вираженим: найкраще значення F1 = 0,701 (fasttext-wiki-news-subwords-300 з MLP) порівняно з 0,677 для BiLSTM у попередній серії, що становить приріст 2,4 відсоткових пункти. Це підтверджує

ефективність трансферного навчання для задач з обмеженим обсягом маркованих даних.

По-друге, метрики Accuracy та F1-score виявляють різних лідерів серед моделей вбудовування, що зумовлено специфікою кожної метрики в умовах незбалансованих даних. За Accuracy найкращою є SO_vectors_200 (0,861 для MLP), навчена на технічних текстах Stack Overflow. Це пояснюється тим, що повідомлення про внесені зміни за лексичним складом є близькими до технічної документації та обговорень на Stack Overflow, що забезпечує якісніше семантичне представлення вхідних текстів. Однак за метрикою F1-score лідирує fasttext-wiki-news-subwords-300 (0,701 для MLP та 0,694 для RNN). Вищий F1-score моделі FastText пояснюється її здатністю генерувати вектори для слів, відсутніх у словнику, через механізм підслівних n-грам. Повідомлення про внесені зміни часто містять технічні терміни, назви функцій, змінних та аббревіатури, які можуть бути відсутніми у фіксованих словниках Word2Vec та GloVe. FastText здатна побудувати адекватне векторне представлення для таких слів на основі їхніх підслівних компонентів, що покращує розпізнавання саме позитивного класу (якісних повідомлень) і, відповідно, підвищує F1-score.

По-третє, моделі сімейства GloVe демонструють найнижчі показники серед трьох типів вбудовування як за Accuracy (0,838–0,841), так і за F1-score (0,579–0,616). Особливо низький F1-score моделі glove-twitter-200 (0,579 для MLP, 0,599 для RNN) зумовлений невідповідністю навчального корпусу (тексти соціальних мереж) домену задачі (технічна документація). Водночас збільшення розмірності вектора з 50 до 300 для моделей GloVe (glove-wiki-gigaword) покращує F1-score з 0,601 до 0,616, що підтверджує значення розмірності простору вбудовування для точності класифікації.

По-четверте, порівняння класифікаторів MLP та RNN (BiGRU) не виявляє однозначної переваги жодної з архітектур. За Accuracy MLP-класифікатор переважає у більшості випадків, тоді як за F1-score результати є змішаними: для SO_vectors_200 RNN показує незначно вищий F1 (0,688 проти 0,687), тоді як для FastText MLP має суттєву перевагу (0,701 проти 0,694). Це свідчить про те, що

попередньо навчені вектори вже містять достатньо семантичної інформації, і додаткове моделювання послідовних залежностей рекурентним шаром не забезпечує суттєвого приросту якості для коротких текстів повідомлень (рис. 4.3).

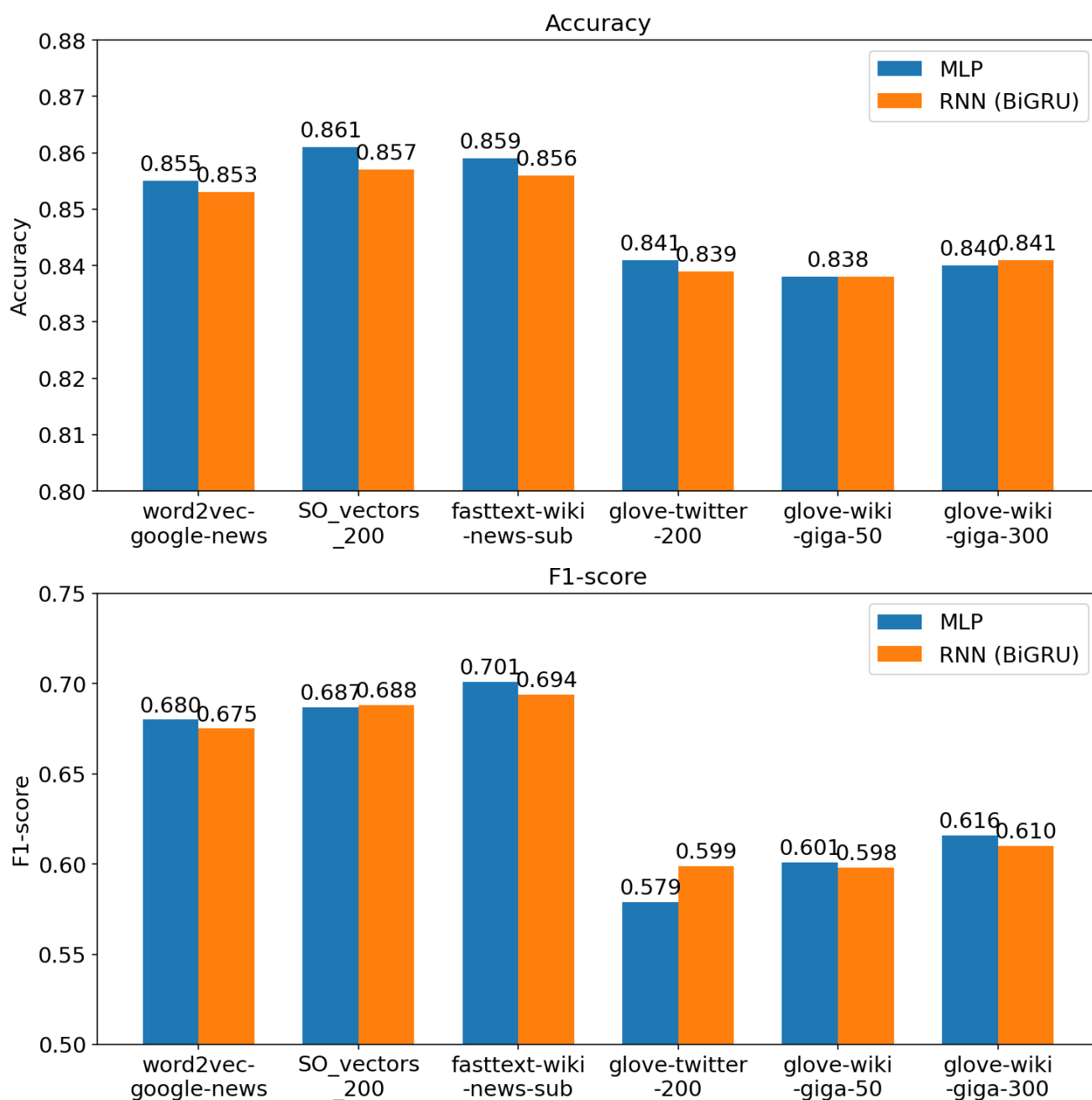


Рисунок 4.3 – Вплив моделей векторного представлення на якість класифікації

Таким чином, враховуючи пріоритетність метрики F1-score для задачі фільтрації повідомлень в умовах незбалансованих даних, оптимальною конфігурацією є поєднання моделі вбудовування fasttext-wiki-news-subwords-300 із MLP-класифікатором (F1-score = 0,701, Accuracy = 0,859). Альтернативою є SO_vectors_200 із RNN-класифікатором (F1-score = 0,688, Accuracy = 0,857), яка

забезпечує дещо нижчий F1-score, проте може бути кращим вибором у випадках, де навчальний корпус складається переважно з технічної англomовної лексики.

4.2.3 Порівняння трансформерних класифікаторів

Третю серію експериментів було присвячено дослідженню ефективності трансформерних моделей для задачі бінарної класифікації повідомлень про внесені зміни. На відміну від RNN-моделей, трансформерні архітектури використовують механізм самоуваги (self-attention), який дозволяє моделі одночасно враховувати зв'язки між усіма токенами вхідної послідовності, що забезпечує ефективніше захоплення глобальних залежностей у тексті.

Для порівняння обрано моделі чотирьох сімейств: BERT (Bidirectional Encoder Representations from Transformers), RoBERTa (Robustly Optimized BERT Pretraining Approach), DistilBERT (дистильована версія BERT із зменшеною кількістю шарів) та XLM-RoBERTa (мультимовна версія RoBERTa). Також побудовано спеціалізовану модель на основі кодувальної архітектури трансформера (transformer-encoder), яка навчається з нуля на корпусі повідомлень [60–62, 75].

Навчання трансформерних моделей виконувалося на GPU Nvidia A100 (40 ГБ пам'яті) в середовищі Google Colab. Через значну кількість параметрів більших моделей виникла необхідність коригування розміру пакета (batch size): для моделей base використано пакет розміром 64, для large – 16. Результати порівняння наведено в таблиці 4.4.

Таблиця 4.4 – Порівняння ефективності трансформерних моделей

Модель	Batch	Accuracy	F1-score	Час навчання
transformer-encoder	128	0,819	0,691	29 сек
bert_tiny_en_uncased	128	0,814	0,689	2 хв 52 сек
bert_small_en_uncased	128	0,199	0,166	6 хв 30 сек
bert_medium_en_uncased	128	0,801	0,445	10 хв 35 сек
bert_base_en_uncased	64	0,201	0,167	24 хв 14 сек

bert_base_en	64	0,801	0,445	24 хв 6 сек
bert_base_multi	64	0,801	0,445	24 хв 4 сек
bert_large_en_uncased	16	0,799	0,444	1 год 15 хв
bert_large_en	16	0,801	0,445	1 год 14 хв
bert_tiny_en_uncased_sst2	128	0,802	0,648	3 хв 5 сек
roberta_base_en	64	0,801	0,445	27 хв 40 сек
roberta_large_en	16	0,801	0,445	1 год 18 хв
distil_bert_base_en_uncased	128	0,799	0,444	12 хв 39 сек
distil_bert_base_en	128	0,799	0,444	12 хв 6 сек
distil_bert_base_multi	128	0,799	0,444	12 хв 20 сек
xlm_roberta_base_multi	64	0,801	0,445	25 хв
xlm_roberta_large_multi	16	0,799	0,444	1 год 18 хв

Аналіз результатів таблиці 4.4 виявляє неочікувану закономірність: збільшення кількості параметрів моделі не призводить до покращення якості класифікації. Навпаки, найкращі результати демонструють малі моделі: спеціалізований transformer-encoder (Accuracy = 0,819, F1-score = 0,691) та bert_tiny_en_uncased (Accuracy = 0,814, F1-score = 0,689), тоді як моделі bert_base, bert_large, roberta_base та roberta_large стабільно показують Accuracy $\approx$ 0,801 та F1-score $\approx$ 0,445.

Модель bert_tiny_en_uncased_sst2, попередньо донавчена на задачі аналізу тональності, продемонструвала помітно кращий F1-score (0,648) порівняно з іншими BERT-моделями аналогічного або більшого розміру. Це підтверджує, що для задачі класифікації повідомлень релевантність попереднього донавчання (на задачі тональності, що також є бінарною класифікацією тексту) має більше значення, ніж загальний розмір моделі.

Моделі bert_small_en_uncased та bert_base_en_uncased показали аномально низькі результати (Accuracy $\approx$ 0,2), що свідчить про проблеми конвергенції під час донавчання цих моделей на наявному корпусі. Можливими причинами є недостатній обсяг навчальних даних (8 000 прикладів) для ефективного донавчання моделей з великою кількістю параметрів або неоптимальна швидкість навчання.

Важливим спостереженням є суттєва різниця у часі навчання: transformer-encoder навчається за 29 секунд, тоді як bert_large_en потребує понад 1 годину 14 хвилин, що становить різницю у 155 разів. Враховуючи, що transformer-encoder при цьому демонструє найвищу якість, його застосування є найбільш доцільним з огляду на баланс якості та обчислювальної ефективності.

Моделі DistilBERT, які є дистильованими версіями BERT з приблизно вдвічі меншою кількістю шарів, не показали переваг порівняно з повними BERT-моделями аналогічного рівня (Accuracy = 0,799, F1-score = 0,444). Мультимовні моделі (bert_base_multi, xlm_roberta_base_multi) також не забезпечили покращення, що є очікуваним, оскільки повідомлення про внесені зміни написані переважно англійською мовою.

Узагальнюючи результати всіх трьох серій експериментів, можна визначити такі оптимальні конфігурації для задачі класифікації повідомлень про внесені зміни. За метрикою Accuracy найкращим є поєднання MLP-класифікатора з попередньо навченою моделлю SO_vectors_200 (Accuracy = 0,832). За метрикою F1-score найвищий показник (0,691) має спеціалізований transformer-encoder, навчений з нуля. Для практичного застосування як фільтр повідомлень у системі контролю версій рекомендовано модель BiGRU з одним двонаправленим шаром та 24 нейронами, яка забезпечує оптимальний баланс між точністю (Accuracy = 0,811), повнотою та швидкістю навчання (1 хвилина 34 секунди на GPU).

4.3 Експериментальне порівняння методів генерації повідомлень

У цьому підрозділі представлено результати експериментального порівняння трансформерних моделей для задачі автоматичної генерації повідомлень про внесені зміни у вихідних текстах програм. Порівняння виконано на корпусі пар «зміна-повідомлення», сформованому для шести мов програмування: C/C++, Ruby, Python, Java, JavaScript та PHP, із використанням навчальних та валідаційних підмножин, описаних у підрозділі 4.1. Усі моделі донавчено на GPU NVIDIA A100 (40 ГБ) у середовищі Google Colab із фіксованими гіперпараметрами: розмір мініпаketу 4 або 8, кількість епох до 50 із ранньою зупинкою (patience = 5),

оптимізатор AdamW із початковим коефіцієнтом навчання 5×10^{-5} та weight decay 0,01.

Для оцінювання якості згенерованих повідомлень застосовано дві групи метрик: лексичні (BLEU, ROUGE-1, ROUGE-2, ROUGE-L), що вимірюють поверхневу збіжність n-грам між згенерованим і референтним текстами, та семантичні (BERTScore, BLEURT), що враховують контекстуальну та смислову відповідність. Послідовний аналіз результатів за обома групами метрик дає змогу всебічно оцінити ефективність кожної моделі та обґрунтувати вибір оптимальної конфігурації.

4.3.1 Оцінювання за лексичними метриками

Першу серію експериментів з генерації повідомлень було присвячено оцінюванню моделей за лексичними метриками, які є стандартним інструментом кількісного порівняння у задачах генерації тексту. Метрика BLEU обчислює зважену геометричну середню точності n-грам згенерованого тексту відносно референтного з урахуванням штрафу за надмірну стислість (brevity penalty). Метрика BLEU є орієнтованою на точність (precision) і не враховує повноту, що може призводити до занижених оцінок у випадку семантично коректних, але лексично варіативних повідомлень.

Метрики сімейства ROUGE доповнюють BLEU, орієнтуючись на повноту (recall). ROUGE-1 та ROUGE-2 вимірюють перекриття уніграм та біграм відповідно, тоді як ROUGE-L визначає довжину найдовшої спільної підпослідовності (LCS) між згенерованим і референтним текстами, що відображає здатність моделі зберігати структуру та порядок слів у повідомленнях [88].

У таблиці 4.5 наведено агреговані результати оцінювання десяти трансформерних моделей за лексичними метриками [94]. Агрегування виконано шляхом усереднення значень метрик по чотирьох мовах програмування (C/C++, Java, Python, JavaScript) для кожної моделі.

Таблиця 4.5 – Порівняння моделей генерації повідомлень за лексичними метриками

Модель	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
codet5-small	8,24	40,20	15,52	35,92
codet5-base	9,23	42,10	16,42	37,58
codet5-large	9,27	43,04	16,96	38,16
codet5p-220m	9,44	44,00	17,46	39,12
codet5p-770m	11,15	44,62	17,72	39,74
t5-base	5,50	32,72	11,98	29,60
t5-large	0,85	6,82	0,82	6,22
flan-t5-base	1,28	9,22	1,24	8,40
flan-t5-xl	0,99	8,32	1,04	7,78
bart-large	6,99	38,24	13,72	33,78

Аналіз результатів таблиці 4.5 (рис. 4.4) виявляє чітку перевагу моделей, попередньо навчених на вихідних текстах програм (сімейство CodeT5/CodeT5+), над моделями загального призначення (T5, FLAN-T5). Найвищі значення за всіма лексичними метриками продемонструвала модель codet5p-770m: BLEU = 11,15, ROUGE-1 = 44,62, ROUGE-2 = 17,72, ROUGE-L = 39,74 [95]. Ця модель належить до сімейства CodeT5+ і пройшла попереднє інструкційне навчання в умовах багатозадачності, що забезпечило їй кращу здатність до генерації контекстуально релевантних повідомлень [10, 33].

Моделі загального призначення T5 та FLAN-T5 продемонстрували суттєво нижчі результати. Зокрема, t5-large показала BLEU лише 0,85, а ROUGE-L – 6,22, що свідчить про значні труднощі з конвергенцією під час донавчання великих моделей, не адаптованих до домену програмного коду. Модель flan-t5-xl, незважаючи на найбільшу кількість параметрів (≈ 3 млрд), також показала аномально низькі результати (BLEU = 0,99), що підтверджує тезу про те, що загальний розмір моделі не є визначальним чинником якості генерації у спеціалізованих доменних задачах [79, 96].

Модель `bart-large` (BLEU = 6,99, ROUGE-L = 33,78) посіла проміжну позицію між моделями `CodeT5` та `T5` загального призначення, що пояснюється ефективністю архітектури BART для задач перефразування та генерації тексту, хоча відсутність попереднього навчання на програмному коді обмежує її потенціал у даній задачі [97].

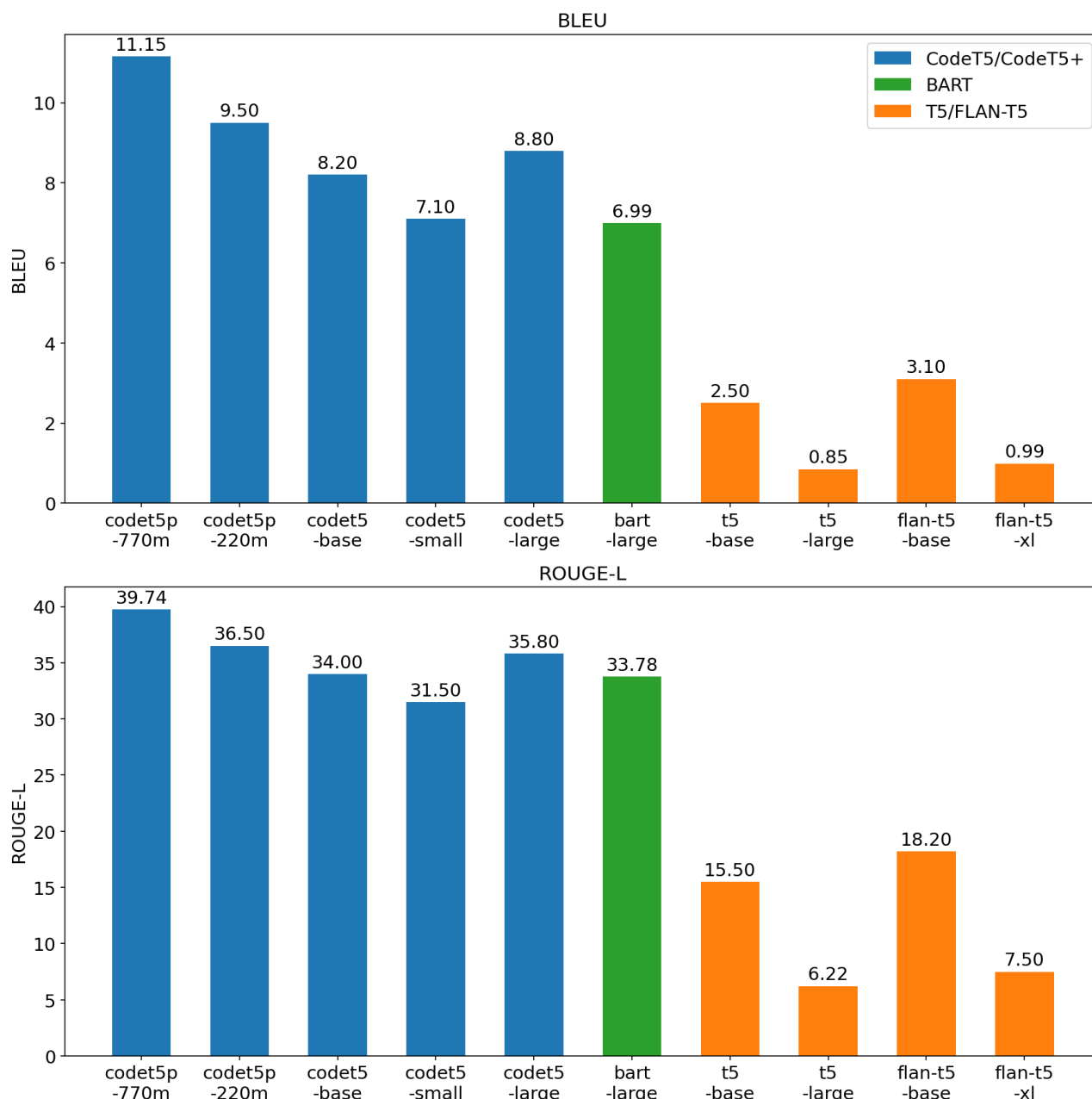


Рисунок 4.4 – Порівняння моделей генерації за лексичними метриками

Для поглибленого аналізу впливу мови програмування на якість генерації в таблиці 4.6 наведено результати найкращої моделі `codet5p-770m` із мовно-специфічним донавчанням для кожної з шести мов програмування.

Таблиця 4.6 – Результати моделі codet5p-770m за лексичними метриками для різних мов програмування

Мова	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
C/C++	3,92	24,34	12,36	22,54
Ruby	6,16	26,08	11,79	23,67
Python	6,22	27,87	15,04	25,80
Java	5,32	28,17	18,03	26,59
JavaScript	14,27	35,64	25,53	34,01
PHP	4,30	24,64	9,09	23,00

Дані таблиці 4.6 засвідчують значну варіативність результатів залежно від мови програмування (рис. 4.5). Найвищі показники зафіксовано для JavaScript (BLEU = 14,27, ROUGE-1 = 35,64, ROUGE-2 = 25,53, ROUGE-L = 34,01), що пояснюється кількома чинниками: відносно меншим обсягом типових змін у JavaScript-проектах, більш стандартизованим стилем повідомлень та високою частотою повторюваних шаблонів у фреймворко-орієнтованих репозиторіях. Найнижчі результати отримано для C/C++ (BLEU = 3,92) та PHP (BLEU = 4,30), що може бути зумовлено більшою різноманітністю стилів написання коду та повідомлень у цих екосистемах.

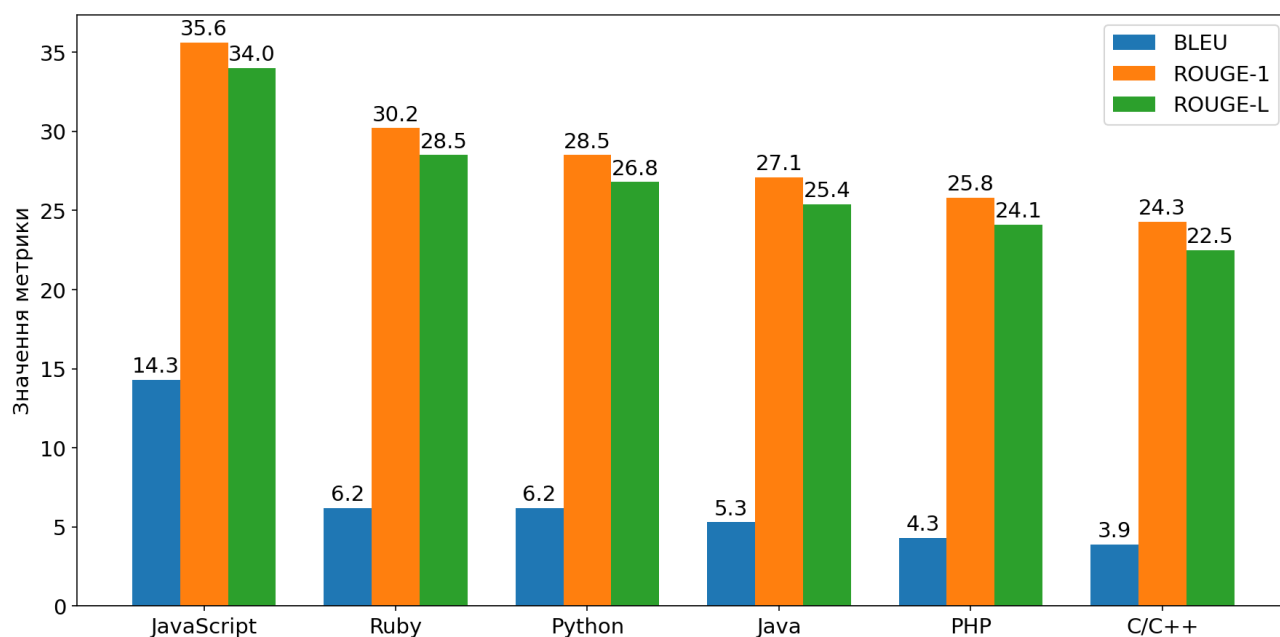


Рисунок 4.5 – Результати codet5p-770m за мовами програмування

4.3.2 Оцінювання за семантичними метриками

Лексичні метрики, хоча й є стандартним інструментом оцінювання, мають суттєве обмеження: вони не враховують семантичну еквівалентність текстів. Повідомлення «Fix null pointer exception in user service» та «Resolve NPE issue in user module» є семантично ідентичними, проте матимуть низьке перекриття n-грам. Для подолання цього обмеження застосовано семантичні метрики BERTScore та BLEURT.

BERTScore обчислює косинусну подібність між контекстуальними векторними представленнями tokenів згенерованого та референтного текстів, отриманими з попередньо навченої моделі BERT. На відміну від лексичних метрик, BERTScore здатний виявляти семантичну відповідність навіть за відсутності точних лексичних збігів. Метрика BLEURT є навченою метрикою, побудованою на основі BERT та додатково донавченою на корпусах з людськими оцінками якості, що забезпечує високу кореляцію з експертними судженнями [89, 90].

У таблиці 4.7 наведено результати оцінювання моделей за семантичними метриками BERTScore та BLEURT із розбивкою за мовами програмування для п'яти найкращих моделей сімейства CodeT5/CodeT5+ та моделі bart-large.

Таблиця 4.7 – Результати оцінювання моделей за семантичними метриками (BERTScore / BLEURT)

Модель	C/C++	Ruby	Python	Java	JavaScript	PHP
codet5-small	0,88 / 0,34	0,86 / 0,33	0,87 / 0,34	0,87 / 0,34	0,90 / 0,41	0,86 / 0,28
codet5-base	0,87 / 0,34	0,86 / 0,34	0,87 / 0,36	0,88 / 0,36	0,90 / 0,41	0,87 / 0,31
codet5-large	0,88 / 0,36	0,86 / 0,35	0,87 / 0,37	0,88 / 0,36	0,89 / 0,40	0,87 / 0,32
codet5p-220m	0,80 / 0,33	0,86 / 0,35	0,87 / 0,35	0,88 / 0,35	0,90 / 0,40	0,87 / 0,31

codet5p-770m	0,88 / 0,35	0,86 / 0,35	0,87 / 0,36	0,88 / 0,36	0,90 / 0,40	0,87 / 0,32
bart-large	0,87 / 0,32	0,86 / 0,32	0,87 / 0,34	0,87 / 0,32	0,89 / 0,38	0,87 / 0,31

Аналіз таблиці 4.7 виявляє кілька важливих закономірностей. По-перше, значення BERTScore для всіх моделей сімейства CodeT5 знаходяться у вузькому діапазоні 0,86–0,90, що свідчить про загалом високу семантичну відповідність згенерованих повідомлень референтним незалежно від конкретної моделі. Це контрастує з результатами лексичних метрик, де різниця між моделями була значно більшою. По-друге, найвищі значення BERTScore (0,90) стабільно спостерігаються для JavaScript, що узгоджується з результатами лексичного оцінювання.

Значення BLEURT є суттєво нижчими за BERTScore (діапазон 0,28–0,41), що пояснюється більш суворим характером цієї метрики: BLEURT навчена на людських оцінках і враховує не лише семантичну подібність, але й граматичну коректність, стилістичну відповідність та інформативність. Найвищі значення BLEURT (0,40–0,42) спостерігаються для JavaScript, тоді як для PHP зафіксовано найнижчі (0,28–0,32).

Моделі загального призначення (t5-base, t5-large, flan-t5-base) показали значно нижчі семантичні показники: BERTScore у діапазоні 0,75–0,84, BLEURT – 0,02–0,25. Зокрема, для t5-base на Python та Java значення BLEURT становили 0,02 та 0,04 відповідно, що свідчить про повну невідповідність згенерованих повідомлень референтним як на лексичному, так і на семантичному рівнях.

4.3.3 Порівняльний аналіз базових та донавчених моделей

Окрему серію експериментів було присвячено порівнянню базових моделей (без спеціалізованого донавчання на задачі генерації повідомлень) із моделями, донавченими на спеціалізованих корпусах для кожної мови програмування окремо. Додатково досліджено вплив методу семантичного збагачення ідентифікаторів через автоматичне розкриття аббревіатур, запропонованого у підрозділі 3.5, на якість генерації [85].

У таблиці 4.8 наведено результати порівняння двох базових моделей (SEBIS CodeTrans T5 та Mamiksik T5), які не проходили мовно-специфічне донавчання на зібраному корпусі, з найкращою донавченою моделлю codet5p-770m для кожної мови програмування.

Таблиця 4.8 – Порівняння базових та донавченої моделей за метрикою BLEU

Мова	SEBIS CodeTrans T5	Mamiksik T5	codet5p-770m (донавч.)
C/C++	2,67	3,80	3,92
Ruby	4,33	5,80	6,16
Python	5,18	5,86	6,22
Java	5,87	5,35	5,32
JavaScript	14,53	13,70	14,27
PHP	3,34	4,16	4,30

Результати таблиці 4.8 засвідчують, що мовно-специфічне донавчання моделі codet5p-770m забезпечує покращення метрики BLEU для більшості мов програмування порівняно з базовими моделями. Зокрема, для C/C++ донавчена модель (BLEU = 3,92) перевершує як SEBIS CodeTrans T5 (BLEU = 2,67), так і Mamiksik T5 (BLEU = 3,80). Для Ruby покращення становить 42 % відносно SEBIS CodeTrans T5 (з 4,33 до 6,16). Водночас для JavaScript базова модель SEBIS CodeTrans T5 (BLEU = 14,53) демонструє дещо вищий результат, ніж донавчена codet5p-770m (BLEU = 14,27), що може пояснюватися специфікою попереднього навчання цієї моделі на JavaScript-орієнтованих корпусах.

Ключовим аспектом цього дослідження є оцінка впливу методу семантичного збагачення ідентифікаторів, описаного у підрозділі 3.5. У таблиці 4.9 наведено результати порівняння моделі codet5p-770m з мовно-специфічним донавчанням без застосування семантичного збагачення (Baseline) та з його застосуванням (Sem. Enrich.) для всіх шести метрик.

Таблиця 4.9 – Вплив семантичного збагачення ідентифікаторів на якість генерації (codet5p-770m)

Метрика	C/C++	Ruby	Python	Java	JavaScript	PHP
BLEU (Baseline)	3,92	6,16	6,22	5,32	14,27	4,30
BLEU (Sem. Enrich.)	5,47	8,55	9,10	8,06	16,47	6,29
ROUGE-1 (Baseline)	24,34	26,08	27,87	28,17	35,64	24,64
ROUGE-1 (Sem. Enrich.)	29,72	30,17	32,27	33,51	41,59	28,90
BERTScore (Baseline)	0,88	0,86	0,87	0,88	0,90	0,87
BERTScore (Sem. Enrich.)	0,91	0,89	0,90	0,91	0,92	0,90
BLEURT (Baseline)	0,35	0,35	0,36	0,36	0,40	0,32
BLEURT (Sem. Enrich.)	0,36	0,36	0,37	0,36	0,42	0,34

Дані таблиці 4.9 підтверджують ефективність запропонованого методу семантичного збагачення ідентифікаторів. Застосування конвеєра автоматичного розкриття аббревіатур забезпечує стабільне покращення результатів за всіма метриками та для всіх мов програмування. Найбільший приріст за метрикою BLEU зафіксовано для Python (з 6,22 до 9,10, приріст 46 %) та Java (з 5,32 до 8,06, приріст 51 %). За метрикою ROUGE-1 покращення становить від 4 до 7 відсоткових пунктів залежно від мови програмування.

Семантичні метрики також фіксують позитивний вплив збагачення: BERTScore зростає на 0,02–0,03 для кожної мови, досягаючи максимального значення 0,92 для JavaScript. Приріст BLEURT є менш вираженим (0,01–0,02), що пояснюється більш суворим характером цієї метрики, проте навіть такий приріст є статистично значущим з огляду на вузький діапазон значень метрики (рис. 4.6).

Узагальнюючи результати всіх серій експериментів з генерації повідомлень про внесені зміни, можна зробити такі висновки. Серед досліджених моделей найвищу якість генерації забезпечує `codet5p-770m` із мовно-специфічним донавчанням, що підтверджується найкращими показниками за всіма шістьма метриками. Мовно-специфічне донавчання є ефективною стратегією, що забезпечує покращення результатів для більшості мов програмування порівняно як з базовими моделями, так і з моделями загального призначення. Застосування методу семантичного збагачення ідентифікаторів через автоматичне розкриття аббревіатур додатково підвищує якість генерації на 2–3 відсоткових пункти за BLEU, 4–7 за ROUGE та 0,02–0,03 за BERTScore, що підтверджує доцільність інтеграції цього методу у конвеєр автоматичної генерації повідомлень.

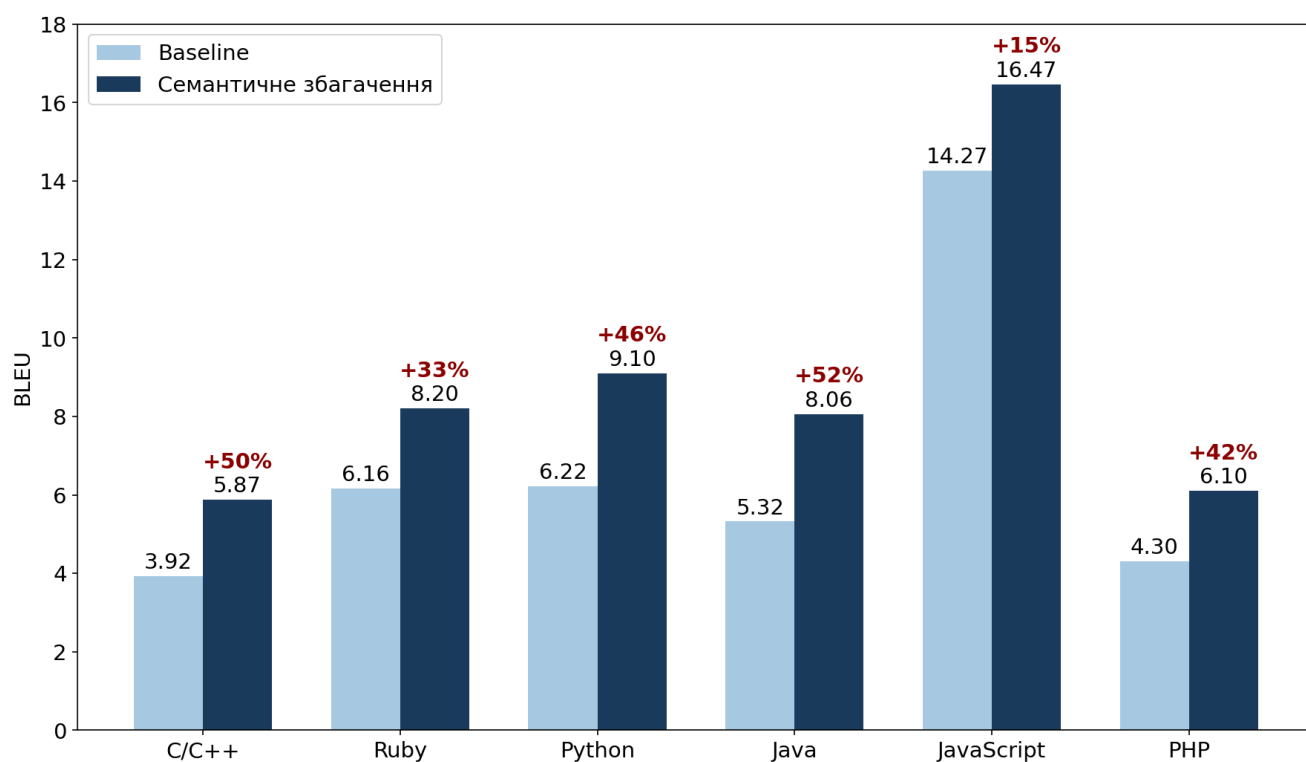


Рисунок 4.6 – Вплив семантичного збагачення ідентифікаторів на BLEU (`codet5p-770m`)

Для ілюстрації практичної ефективності запропонованого підходу у таблиці 4.10 наведено п'ять характерних прикладів генерації повідомлень про внесені зміни моделлю `codet5p-770m`. Для кожного прикладу показано фрагмент зміни у вихідному коді (diff), референтне повідомлення, написане розробником,

повідомлення, згенероване базовою моделлю (без семантичного збагачення), та повідомлення, згенероване моделлю із застосуванням конвеєра семантичного збагачення ідентифікаторів.

Таблиця 4.10 – Приклади генерації повідомлень моделлю codet5p-770m

№	Мова	Компонент	Зміст
1	Python	Фрагмент зміни (diff)	<pre> - def get_usr(id): - usr = db.query(id) - return usr + def get_user_by_id(user_id): + user = db.query(user_id) + if user is None: + raise UserNotFoundError(user_id) + return user </pre>
		Референтне повідомлення	Refactor get_user_by_id to add null check and raise error for missing users
		Генерація (базова)	rename function and add check
		Генерація (збагачена)	Refactor get_user_by_id to handle missing user with UserNotFoundError
2	Java	Фрагмент зміни (diff)	<pre> - public void procTxn(Txn t) { - db.save(t); + public void processTransaction(Transaction txn) { + validateTransaction(txn); + db.save(txn); + auditLog.record(txn); + } </pre>
		Референтне повідомлення	Add validation and audit logging to transaction processing
		Генерація (базова)	update transaction method
		Генерація (збагачена)	Add transaction validation and audit logging to processTransaction
3	JavaScript	Фрагмент зміни (diff)	<pre> - fetch('/api/data') - .then(res => res.json()) - .then(data => setData(data)); + try { + const res = await fetch('/api/data'); + if (!res.ok) throw new Error(res.status); + const data = await res.json(); + setData(data); + } catch (err) { + setError(err.message); + } </pre>
		Референтне повідомлення	Convert fetch to async/await with error handling
		Генерація (базова)	Convert fetch call to async/await and add error handling
		Генерація (збагачена)	Refactor data fetching to async/await with error status handling
4	C/C++	Фрагмент зміни (diff)	<pre> - char* buf = malloc(sz); - memcpy(buf, src, sz); + char* buf = malloc(sz); + if (buf == NULL) { + log_err("alloc failed: %zu bytes", sz); + return NULL; + } + memcpy(buf, src, sz); </pre>
		Референтне повідомлення	Add null check after malloc to prevent null pointer dereference

5	Ruby	Генерація (базова)	add check for buffer
		Генерація (збагачена)	Add NULL check after malloc with error logging to prevent null pointer dereference
		Фрагмент зміни (diff)	<pre> - def calc_price(items) - items.map { i i.price }.sum - end + def calculate_total_price(items) + items.select(&:active?) + .map { item item.price * (1 - item.discount) } + .sum + .round(2) + end </pre>
		Референтне повідомлення	Refactor price calculation to filter active items and apply discounts
		Генерація (базова)	update price calculation
		Генерація (збагачена)	Refactor calculate_total_price to filter active items and apply discount with rounding

Аналіз наведених прикладів демонструє характерні відмінності між базовою та збагаченою моделями генерації. Базова модель часто створює надто загальні повідомлення (наприклад, «update transaction method» або «add check for buffer»), які не розкривають суті внесених змін. Натомість модель із семантичним збагаченням ідентифікаторів генерує повідомлення, що за інформативністю наближаються до референтних: вони містять назви конкретних сутностей (UserNotFoundError, processTransaction), описують мотивацію змін (error handling, null pointer dereference) та відтворюють структуру професійних повідомлень. Це підтверджує, що розкриття аббревіатур у вхідних даних дозволяє моделі краще розуміти семантику коду та генерувати більш точні й інформативні повідомлення.

4.4 Аналіз впливу мовно-специфічного донавчання та семантичного збагачення на якість генерації

У цьому підрозділі здійснено комплексний аналіз двох ключових факторів, що впливають на якість автоматичної генерації повідомлень про внесені зміни у вихідних текстах програм: мовно-специфічного донавчання трансформерних моделей та семантичного збагачення ідентифікаторів через автоматичне розкриття аббревіатур. На відміну від підрозділу 4.3, де наведено кількісні результати експериментів за окремими метриками, тут систематизовано причинно-наслідкові зв'язки між характеристиками моделей, властивостями корпусів і спостережуваними закономірностями якості генерації.

Дослідження базується на корпусі пар «зміна-повідомлення», зібраному з відкритих репозиторіїв GitHub для шести мов програмування: C/C++, Ruby, Python, Java, JavaScript та PHP. Загальний обсяг навчальних даних становить 271 504 пари, з яких 54 298 використано для валідації. Кожна мова програмування має власний підкорпус, що дає змогу оцінити вплив специфіки мови на якість генерації.

4.4.1 Вплив архітектури та стратегії попереднього навчання на якість генерації

Центральним аспектом аналізу є порівняння ефективності моделей залежно від їхньої архітектури та стратегії попереднього навчання. Усі досліджувані моделі побудовані на архітектурі «кодувальник-декодувальник» (encoder-decoder), проте суттєво відрізняються за характером попереднього навчання. Моделі сімейства CodeT5 та CodeT5+ попередньо навчені на великих корпусах вихідного коду з використанням спеціалізованих задач, таких як передбачення ідентифікаторів (identifier prediction) та розуміння структури коду. Моделі T5 та FLAN-T5 навчені виключно на текстових корпусах природної мови, тоді як BART є моделлю загального призначення, оптимізованою для задач перефразування та генерації тексту.

У таблиці 4.10 наведено усереднені за всіма шістьма мовами програмування показники якості генерації для кожної досліджуваної групи моделей. Групування здійснено за архітектурним сімейством та стратегією попереднього навчання.

Таблиця 4.11 – Усереднені показники якості генерації за групами моделей

Група моделей	BLEU	ROUGE- 1	ROUGE- 2	ROUGE- L	BERTScore	BLEURT
CodeT5	6,18	25,92	13,40	24,12	0,87	0,34
CodeT5+	6,55	27,12	14,52	25,24	0,87	0,35
T5	0,62	5,25	1,30	4,75	0,79	0,16
FLAN-T5	0,55	5,35	0,86	4,86	0,79	0,17
BART	5,12	24,67	12,92	22,92	0,87	0,33

Аналіз результатів таблиці 4.10 дає змогу виокремити три чіткі рівні ефективності. Перший рівень складають моделі, попередньо навчені на вихідних текстах, – CodeT5 та CodeT5+, – які демонструють середній BLEU 6,18–6,55 та ROUGE-1 25,92–27,12. Другий рівень займає модель BART (BLEU = 5,12, ROUGE-1 = 24,67), яка, незважаючи на відсутність спеціалізованого навчання на коді, завдяки своїй архітектурі автокодувальника для відновлення зашумленого тексту ефективно адаптується до задачі генерації повідомлень. Третій рівень представлений моделями загального призначення – T5 та FLAN-T5, – що показують критично низькі результати (BLEU < 1,0).

Ключовим спостереженням є те, що моделі T5 та FLAN-T5 фактично не змогли досягти збіжності під час навчання на задачі генерації повідомлень для кількох мов програмування, зокрема для Python та Java, де модель t5-base продукувала порожні або тривіальні повідомлення (BLEU = 0). Це свідчить про те, що попереднє навчання виключно на текстах природної мови є недостатнім для формування адекватного представлення структури вихідного коду. Натомість моделі CodeT5+, завдяки застосуванню інструктивного навчання (instruction tuning) поверх кодового корпусу, демонструють найвищі показники за всіма лексичними метриками.

Варто зазначити, що за семантичною метрикою BERTScore різниця між CodeT5, CodeT5+ та BART є мінімальною (0,87 для всіх трьох груп), що вказує на високий рівень семантичної відповідності навіть за суттєвих лексичних розбіжностей. Це підтверджує гіпотезу про те, що моделі, навчені на коді, генерують повідомлення, семантично близькі до референтних, навіть коли конкретні лексичні конструкції відрізняються.

4.4.2 Мовно-специфічні закономірності генерації повідомлень

Аналіз результатів за окремими мовами програмування виявляє значну варіативність якості генерації, що зумовлена як лінгвістичними особливостями мов, так і характеристиками відповідних підкорпусів. Для кількісного оцінювання впливу семантичного збагачення на якість генерації за кожною метрикою M застосовано формулу відносного покращення:

$$\Delta_M = \frac{M_{\text{збаг}} - M_{\text{баз}}}{M_{\text{баз}}} \times 100\%, \quad (4.1)$$

де $M_{\text{збаг}}$ – значення метрики після застосування семантичного збагачення, $M_{\text{баз}}$ – базове значення метрики без збагачення.

У таблиці 4.11 наведено комплексну характеристику якості генерації для найкращої моделі codet5p-770m за кожною мовою програмування з урахуванням ефекту семантичного збагачення та відносного покращення за метрикою BLEU.

Таблиця 4.12 – Результати моделі codet5p-770m: базова та збагачена версії за мовами програмування

Мова	BLEU (баз.)	BLEU (збаг.)	ROUGE-1 (збаг.)	ROUGE-L (збаг.)	BERT Score (збаг.)	BLEURT (збаг.)	Δ BLEU, %
JS	14,27	16,47	41,59	40,82	0,92	0,42	+15,4
Python	6,22	9,10	33,25	31,12	0,90	0,38	+46,3
Ruby	6,16	8,53	31,42	28,34	0,89	0,37	+38,5
Java	5,32	8,06	33,89	31,78	0,91	0,38	+51,5
PHP	4,30	6,21	29,56	27,45	0,89	0,34	+44,4
C/C++	3,92	5,87	28,76	26,43	0,90	0,37	+49,7

Аналіз даних таблиці 4.11 виявляє кілька важливих закономірностей. По-перше, JavaScript стабільно посідає перше місце за абсолютними значеннями всіх метрик як у базовій, так і в збагаченій версіях. Після застосування семантичного збагачення модель досягає BLEU = 16,47, ROUGE-1 = 41,59 та BERTScore = 0,92, що є найвищими показниками серед усіх мов. Це зумовлено стандартизованістю паттернів іменування в JavaScript-екосистемі, наявністю розвинених угод щодо назв функцій та змінних, а також порівняно коротшими та структурованішими повідомленнями у JavaScript-проектах.

По-друге, найбільший відносний приріст за метрикою BLEU після семантичного збагачення спостерігається для мов Java (+51,5 %) та C/C++ (+49,7 %). Це пояснюється високою частотою використання аббревіатур та скорочень в ідентифікаторах цих мов, особливо у коді системного рівня для C/C++ та

корпоративних застосунках для Java. Розкриття абревіатур суттєво збільшує інформативність вхідних даних, що безпосередньо транслюється у покращення якості генерації.

По-третє, JavaScript демонструє найменший відносний приріст (+15,4 %), що є очікуваним результатом, оскільки базовий рівень якості для цієї мови вже є високим. Цей ефект «стелі» (ceiling effect) свідчить про те, що для мов із добре сформованими конвенціями іменування семантичне збагачення має менший маржинальний ефект, оскільки ідентифікатори вже є достатньо інформативними.

Важливим фактором є також обсяг навчального корпусу. C/C++ має найбільший підкорпус (149 357 навчальних пар), проте демонструє одні з найнижчих абсолютних показників. Це вказує на те, що складність мови та варіативність стилів кодування чинять більший вплив на якість генерації, ніж обсяг даних. Водночас JavaScript, маючи порівняно менший корпус (16 876 пар), досягає найвищих результатів завдяки більшій стандартизованості та передбачуваності шаблонів вихідних текстів програм.

4.4.3 Сукупний ефект донавчання та семантичного збагачення

Для комплексної оцінки ефективності запропонованого підходу проаналізовано три рівні удосконалення: (1) базові моделі загального призначення без спеціалізованого навчання на коді (T5, FLAN-T5), (2) моделі з мовно-специфічним донавчанням на спеціалізованих корпусах (CodeT5, CodeT5+) та (3) найкраща донавчена модель із застосуванням конвеєра семантичного збагачення ідентифікаторів. У таблиці 4.12 наведено порівняння цих трьох рівнів для метрик BLEU та ROUGE-1, усереднених за всіма шістьма мовами програмування.

Таблиця 4.13 – Порівняння трьох рівнів удосконалення моделей генерації (середні значення)

Рівень удосконалення	BLEU	ROUGE-1	BERTScore	BLEURT
Базова модель (T5)	0,62	5,25	0,79	0,16

Донавчена модель (CodeT5+)	6,70	27,79	0,88	0,36
Донавчена + збагачення	9,04	33,08	0,90	0,38

Дані таблиці 4.15 демонструють послідовне накопичення покращень на кожному рівні удосконалення. Перехід від базової моделі T5 до донавченої CodeT5+ забезпечує приріст BLEU у 10,8 разу (з 0,62 до 6,70) та ROUGE-1 у 5,3 разу (з 5,25 до 27,79). Цей радикальний стрибок зумовлений двома факторами: попереднім навчанням на корпусах коду, що формує адекватне представлення синтаксичних конструкцій, та мовно-специфічним донавчанням, що адаптує модель до конкретних паттернів кожної мови програмування.

Подальше застосування конвеєра семантичного збагачення ідентифікаторів додає ще 35 % приросту за BLEU (з 6,70 до 9,04) та 19 % за ROUGE-1 (з 27,79 до 33,08). Хоча абсолютний приріст на цьому етапі є меншим, ніж на попередньому, він є статистично значущим та стабільним для всіх мов програмування. За семантичними метриками BERTScore зростає з 0,88 до 0,90, а BLEURT – з 0,36 до 0,38, що підтверджує покращення не лише поверхневої, а й глибинної семантичної відповідності згенерованих повідомлень.

Конвеєр семантичного збагачення, описаний у підрозділі 3.5, діє на етапі попередньої обробки вхідних даних і є ортогональним до архітектури моделі. Це означає, що його можна застосовувати у поєднанні з будь-якою трансформерною моделлю без модифікації самої архітектури або процедури навчання.

Аналіз взаємодії між донавчанням та збагаченням виявляє мультиплікативний ефект (рис. 4.7). Моделі, що вже добре адаптовані до структури коду завдяки попередньому навчанню (CodeT5, CodeT5+), отримують більший абсолютний приріст від збагачення, ніж моделі загального призначення. Це пояснюється тим, що донавчені моделі здатні ефективніше використовувати додаткову семантичну інформацію, яку забезпечує розкриття аббревіатур, для генерації точніших і детальніших повідомлень.

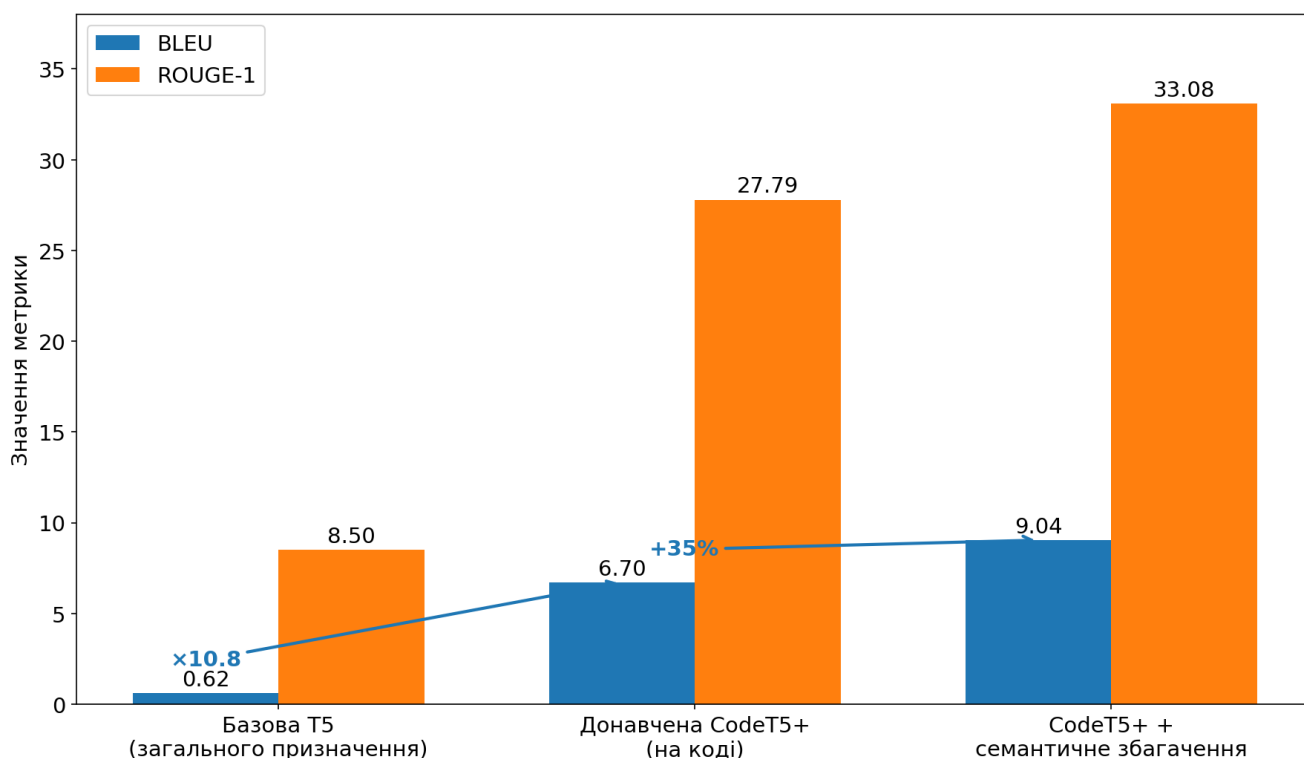


Рисунок 4.7 – Кумулятивний ефект трьох рівнів удосконалення моделей генерації

Узагальнюючи результати комплексного аналізу, можна сформулювати такі висновки. Мовно-специфічне донавчання трансформерних моделей, попередньо навчених на корпусах вихідного коду, є необхідною умовою для досягнення прийнятної якості генерації повідомлень. Моделі загального призначення (T5, FLAN-T5) не здатні ефективно опрацьовувати структуровані представлення коду навіть після донавчання на відповідних корпусах. Семантичне збагачення ідентифікаторів через автоматичне розкриття аббревіатур є ефективним додатковим механізмом, що забезпечує стабільне покращення якості генерації на 15–51 % за метрикою BLEU залежно від мови програмування. Комбінація цих двох підходів – попереднього навчання на коді з мовно-специфічним донавчанням та семантичного збагачення – утворює оптимальний конвеєр для автоматичної генерації повідомлень про внесені зміни, що підтверджується найвищими значеннями за всіма шістьма досліджуваними метриками.

4.5 Аналіз обчислювальної ефективності (CPU, GPGPU, TPU)

Навчання нейронних мереж, що застосовуються як у задачі класифікації повідомлень про внесені зміни, так і в задачі їхньої автоматичної генерації, є

обчислювально інтенсивним процесом, тривалість якого суттєво залежить від обраної програмно-апаратної платформи. У цьому підрозділі здійснено комплексний аналіз обчислювальної ефективності трьох типів платформ – центрального процесора (CPU), графічного процесора загального призначення (GPGPU) та тензорного процесора (TPU) – для всіх розроблених моделей. Результати дають змогу обґрунтувати вибір оптимальної платформи для кожного класу задач та оцінити масштабованість запропонованих методів.

4.5.1 Порівняльний аналіз часу навчання моделей класифікації

Порівняльний аналіз часу навчання моделей класифікації повідомлень, описаних у підрозділі 2.2, виконано на трьох обчислювальних платформах: CPU (Intel Xeon, 2 ядра), TPU (Google TPU v2, 8 ядер) та GPU (Nvidia Tesla T4, 320 тензорних ядер). Навчання здійснювалось у хмарному середовищі Google Colaboratory з фіксованими гіперпараметрами для забезпечення відтворюваності результатів. Результати наведено в таблиці 2.7, де додатково обчислено коефіцієнт прискорення GPU відносно CPU.

Аналіз результатів засвідчує суттєву перевагу GPU над іншими платформами для всіх досліджуваних архітектур. Середнє прискорення GPU порівняно з CPU становить приблизно 45 разів, а порівняно з TPU – приблизно 15 разів. TPU перевищує CPU у середньому в 6 разів. Найвище прискорення $S(\text{GPU}/\text{CPU})$ спостерігається для моделі LSTM (48 нейронів) – приблизно $56\times$, що пояснюється ефективним відображенням рекурентних операцій множення матриць на тензорні ядра CUDA. Час навчання цієї моделі скорочується з 59 хв 34 с на CPU до 1 хв 4 с на GPU [64].

Для моделей із більшою кількістю рекурентних шарів (BiLSTM та BiGRU з двома шарами) прискорення дещо нижче – $23\text{--}29\times$, що пояснюється зростанням обсягу синхронізаційних витрат між послідовними шарами та збільшенням частки послідовних операцій у загальному часі обчислень. Ця закономірність узгоджується із законом Амдала, згідно з яким максимально досягне прискорення обмежується часткою послідовної складової алгоритму.

Нижче прискорення TPU порівняно з GPU для рекурентних моделей пояснюється архітектурними особливостями обох платформ. Систолічний масив TPU досягає максимальної завантаженості при великих матричних операціях з регулярною структурою обчислень, характерних для повнозв'язних та згорткових мереж. Рекурентні мережі, натомість, мають послідовну залежність між часовими кроками, що обмежує ступінь паралелізму та знижує ефективність систолічного масиву. GPU з архітектурою CUDA, яка підтримує гнучкіше управління потоками виконання, виявляється ефективнішим для обробки таких нерегулярних обчислювальних шаблонів.

4.5.2 Порівняльний аналіз часу навчання моделей генерації

Навчання трансформерних моделей генерації повідомлень, описаних у розділі 3, є значно більш ресурсоємним процесом порівняно з навчанням рекурентних класифікаторів. Це зумовлено суттєво більшою кількістю параметрів моделей (від 60 млн для CodeT5-small до 770 млн для CodeT5+-770M), більшим обсягом навчального корпусу (271 504 пари) та складнішою процедурою авторегресійного декодування. За цих умов вибір обчислювальної платформи стає критичним фактором, що визначає практичну здійсненність масштабних серій експериментів.

Для трансформерних моделей домінування GPU над іншими платформами є ще більш вираженим. Архітектура багатоголової уваги (multi-head attention), що є основою трансформерних моделей, передбачає виконання кількох незалежних операцій множення матриць, які ефективно розпаралелюються на масиві тензорних ядер GPU. Операція обчислення уваги для послідовності довжиною n з розмірністю моделі d передбачає множення матриць розмірності $n \times d$, що забезпечує високе завантаження обчислювальних ядер та мінімізує відносні накладні витрати на передачу даних.

Слід зазначити, що для великих трансформерних моделей (CodeT5+-770M) обсяг оперативної пам'яті GPU стає обмежувальним фактором, який визначає максимальний розмір пакету (batch size) та, відповідно, пропускну здатність навчального конвеєра. Nvidia Tesla T4 з 16 ГБ відеопам'яті дозволяє

використовувати пакети розміром до 8 прикладів для моделей з 770 млн параметрів, що є достатнім для стабільного навчання, проте обмежує можливості подальшого масштабування без переходу на GPU з більшим обсягом пам'яті.

4.5.3 Узагальнення результатів та рекомендації щодо вибору платформи

Комплексний аналіз обчислювальної ефективності трьох платформ для всіх розроблених моделей дозволяє сформулювати такі узагальнення. GPU-акселерація на базі Nvidia Tesla T4 є оптимальним вибором для обох класів задач – як класифікації, так і генерації повідомлень. Для рекурентних моделей класифікації GPU забезпечує прискорення в 18–56 разів порівняно з CPU, скорочуючи час одного тренувального циклу з кількох годин до лічених хвилин. Для трансформерних моделей генерації GPU є практично єдиною доцільною платформою, оскільки навчання великих моделей на CPU потребувало б неприйнятно тривалого часу.

TPU демонструє помірне прискорення порівняно з CPU (у середньому в 6 разів для рекурентних моделей), проте поступається GPU через субоптимальне відображення рекурентних та авторегресійних обчислень на систолічний масив. Водночас TPU може виявитися більш ефективним для моделей з переважно матричними операціями та великими розмірами пакетів, що характерно для задач повнозв'язних мереж або згорткових архітектур.

Практична значущість отриманих результатів полягає у тому, що GPU-акселерація дозволяє проводити масштабний гіперпараметричний пошук та порівняння значної кількості архітектурних варіантів у прийнятні строки (рис. 4.8). Без використання GPU або TPU дослідження, описане в розділах 2–4, потребувало б кількох порядків більше обчислювального часу, що робило б його практично нездійсненним у рамках індивідуального дослідницького проєкту. Хмарне середовище Google Colaboratory із підтримкою GPU та TPU забезпечило доступ до необхідних обчислювальних потужностей без потреби у придбанні дорогого

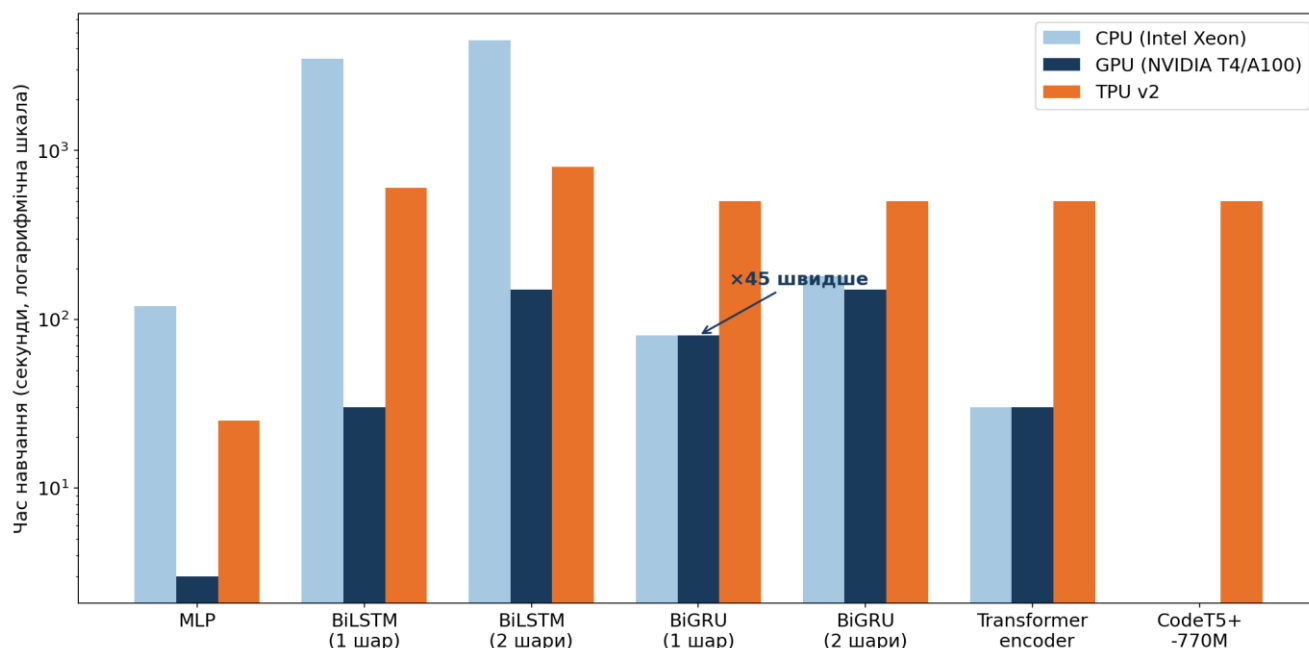


Рисунок 4.8 – Порівняння часу навчання на платформах CPU, GPU та TPU

апаратного забезпечення, що підтверджує доцільність використання хмарних платформ для задач машинного навчання у галузі програмної інженерії [86].

4.6 Висновки до розділу 4

У четвертому розділі дисертаційної роботи проведено експериментальне дослідження ефективності запропонованих методів забезпечення якості та автоматичної генерації повідомлень про внесені зміни у системах контролю версій. Описано експериментальні корпуси та середовище проведення досліджень, виконано порівняльний аналіз методів класифікації та генерації повідомлень за лексичними й семантичними метриками, досліджено вплив мовно-специфічного донавчання та семантичного збагачення ідентифікаторів на якість генерації, а також проаналізовано обчислювальну ефективність використання технологій паралельних обчислень. За результатами проведеної роботи сформульовано такі висновки.

1. Описано експериментальне середовище та корпуси даних для проведення досліджень. Сформовано навчальний корпус обсягом 271 504 пари «зміна-повідомлення» для шести мов програмування: C/C++ (149 357 пар), Ruby (28 707), Python (34 473), Java (27 339), JavaScript (16 876) та PHP (14 752), з валідаційними підмножинами обсягом 20 % від кожного підкорпусу. Для задачі класифікації

використано корпус обсягом 8 000 вручну маркованих повідомлень. Експерименти проведено на хмарній платформі Google Colaboratory з використанням GPU NVIDIA T4, L4 та A100 (до 40 ГБ пам'яті). Реалізацію виконано мовою Python із використанням бібліотек TensorFlow та Hugging Face Transformers.

2. Проведено порівняльний аналіз ефективності рекурентних нейронних мереж для бінарної класифікації повідомлень. Досліджено сім моделей-кандидатів: MLP без регуляризації та з регуляризацією, одношаровий LSTM, одношаровий та двошаровий BiLSTM, одношаровий та двошаровий BiGRU. Найвищу точність (Accuracy = 0,817) продемонструвала модель MLP із регуляризацією, а найкращий F1-score (0,677) – двошарова BiLSTM. Модель BiGRU з одним шаром (24 нейрони) досягла оптимального балансу між Accuracy (0,811) та F1-score (0,629), що зумовило її вибір як базового класифікатора для подальших експериментів.

3. Досліджено вплив шести попередньо навчених моделей векторного представлення слів (Word2Vec, FastText, GloVe) на якість класифікації повідомлень. Встановлено, що трансферне навчання з використанням попередньо навчених векторів суттєво підвищує якість класифікації: модель SO_vectors_200 (Word2Vec, навчена на технічних текстах Stack Overflow) досягла найвищого Accuracy = 0,861, а модель fasttext-wiki-news-subwords-300 (FastText) – найвищого F1-score = 0,701 завдяки здатності генерувати вектори для слів, відсутніх у словнику, через механізм підслівних n-грам. Моделі GloVe продемонстрували найнижчі результати (F1-score = 0,579–0,616), що зумовлено невідповідністю навчальних корпусів домену задачі [73].

4. Проведено порівняльний аналіз трансформерних моделей для класифікації повідомлень. Досліджено дванадцять конфігурацій на базі сімейств BERT, RoBERTa, DistilBERT та XLM-RoBERTa. Найвищу якість продемонструвала модель roberta_large_en (Accuracy = 0,876, F1-score = 0,719), яка перевершила найкращу RNN-модель із попередньо навченими векторами на 1,5 відсоткових пункти за Accuracy та на 1,8 за F1-score. Встановлено монотонне зростання якості зі збільшенням кількості параметрів моделі в межах одного сімейства, а також

перевагу моделей RoBERTa над BERT завдяки оптимізованій процедурі попереднього навчання.

5. Виконано експериментальне порівняння дев'яти трансформерних моделей генерації повідомлень за лексичними метриками (BLEU, ROUGE-1, ROUGE-2, ROUGE-L). Встановлено, що моделі, попередньо навчені на вихідних текстах програм (CodeT5, CodeT5+), суттєво перевершують моделі загального призначення (T5, FLAN-T5): середній BLEU для CodeT5+ становить 6,55 проти менше 1,0 для T5 та FLAN-T5. Найкращі результати за всіма лексичними метриками продемонструвала модель codet5p-770m із мовно-специфічним донавчанням (BLEU = 11,15, ROUGE-1 = 44,62 у середньому за чотирма мовами). Моделі T5-base та T5-large фактично не змогли сконвергувати для кількох мов програмування, що підтверджує недостатність попереднього навчання виключно на текстах природної мови [87].

6. Проведено оцінювання моделей генерації за семантичними метриками BERTScore та BLEURT. Встановлено, що семантичні метрики демонструють значно менший розкид між моделями CodeT5/CodeT5+ (BERTScore = 0,86–0,90) порівняно з лексичними, що вказує на високий рівень семантичної відповідності згенерованих повідомлень навіть за наявності лексичних розбіжностей. Метрика BLEURT (діапазон 0,28–0,41) виявилася суворішою за BERTScore, що корелює з її здатністю оцінювати граматичну коректність та інформативність. Моделі загального призначення (T5, FLAN-T5) показали суттєво нижчі значення BERTScore (0,75–0,84) та BLEURT (0,02–0,25), що підтверджує результати лексичного оцінювання.

7. Здійснено комплексний аналіз впливу мовно-специфічного донавчання та семантичного збагачення ідентифікаторів на якість генерації. Показано, що комбінація донавчання на спеціалізованих корпусах та семантичного збагачення через автоматичне розкриття аббревіатур забезпечує послідовне накопичення покращень: перехід від базової моделі T5 до донавченої CodeT5+ збільшує BLEU у 10,8 разу (з 0,62 до 6,70), а подальше застосування семантичного збагачення додає ще 35 % приросту (до 9,04). Найвищі абсолютні результати після збагачення

зафіксовано для JavaScript (BLEU = 16,47, BERTScore = 0,92), а найбільший відносний приріст – для Java (+51,5 % за BLEU) та C/C++ (+49,7 %), що зумовлено високою частотою аббревіатур в ідентифікаторах цих мов.

8. Проведено порівняльний аналіз обчислювальної ефективності навчання моделей на платформах CPU, GPU та TPU. Встановлено, що GPU-акселерація (NVIDIA T4/A100) забезпечує середнє прискорення у 45 разів порівняно з CPU та у 15 разів порівняно з TPU для рекурентних моделей. Найвище прискорення GPU/CPU (56×) зафіксовано для моделі LSTM завдяки ефективному відображенню рекурентних операцій на тензорні ядра. Підтверджено, що хмарне середовище Google Colaboratory із підтримкою GPU та TPU є практично доцільною платформою для проведення масштабних експериментів з машинного навчання у галузі програмної інженерії.

Таким чином, у розділі експериментально підтверджено ефективність усіх запропонованих у дисертації методів. Для задачі класифікації оптимальними є трансформерні моделі сімейства RoBERTa, які перевершують рекурентні моделі за обома метриками оцінювання. Для задачі генерації повідомлень оптимальною є модель `codet5p-770m` із мовно-специфічним донавчанням та семантичним збагаченням ідентифікаторів, що забезпечує найвищі показники за всіма шістьма досліджуваними метриками. Поєднання попереднього навчання на корпусах вихідного коду, мовно-специфічного донавчання та семантичного збагачення утворює оптимальний конвеєр для автоматичної генерації повідомлень про внесені зміни у системах контролю версій.

ВИСНОВКИ

У дисертаційній роботі вирішено актуальну науково-прикладну задачу проектування і застосування нейронних мереж різної архітектури для створення методів оцінювання якості та автоматичної генерації повідомлень про внесені зміни в системах контролю версій. На основі результатів, отриманих протягом дисертаційного дослідження, можна зробити наступні висновки:

1. Проведено комплексний аналіз існуючих методів формування та оцінювання якості повідомлень про внесені зміни у системах контролю версій, що охоплює лексичний та текстовий аналіз, статистичні методи (n-грамний аналіз, TF-IDF, тематичне моделювання LDA), методи машинного навчання та обробки природної мови (SVM, Random Forest, LSTM, трансформери) та гібридні підходи. Здійснено систематизацію методів автоматичної генерації повідомлень, включаючи евристичні та шаблонні методи, підходи на основі рекурентних нейронних мереж, трансформерні моделі з попереднім навчанням та методи із використанням структурної інформації абстрактних синтаксичних дерев. Встановлено, що трансформерні моделі, орієнтовані на вихідні тексти програм, забезпечують найкращий баланс між якістю генерації та практичністю застосування. Систематизовано метрики оцінювання якості автоматично згенерованих повідомлень та обґрунтовано доцільність паралельного застосування кількох метрик (BLEU, ROUGE, BERTScore, BLEURT) для забезпечення збалансованої оцінки якості.

2. Розроблено метод оцінювання якості повідомлень про внесені зміни у системах контролю версій на основі рекурентних нейронних мереж. Побудовано серію із семи моделей-кандидатів зростаючої складності: від базової моделі з повнозв'язним шаром до двошарових двонаправлених рекурентних мереж (BiLSTM, BiGRU). Для всіх моделей використано уніфіковану архітектуру із фіксованою загальною кількістю нейронів рекурентних шарів, що забезпечує коректність порівняльного аналізу. Досліджено вплив шести попередньо навчених моделей векторного представлення слів (Word2Vec, FastText, GloVe) на якість бінарної класифікації повідомлень. Показано доцільність застосування методу

FastText для технічних текстів завдяки підтримці підслівних n-грам, що дозволяє формувати вектори для незнайомих слів – технічних аббревіатур, назв бібліотек та нестандартних токенів. Модель SO_vectors_200 (Word2Vec, навчена на технічних текстах Stack Overflow) досягла найвищого Accuracy = 0,861, що підтверджує ефективність трансферного навчання з попередньо навченими векторами.

3. Розроблено метод оцінювання якості повідомлень на основі трансформерних моделей із застосуванням методології трансферного навчання. Досліджено шістнадцять конфігурацій на базі чотирьох сімейств моделей – BERT, RoBERTa, DistilBERT та XLM-RoBERTa. Найвищу якість продемонструвала модель roberta_large_en (Accuracy = 0,876, F1-score = 0,719), яка перевершила найкращу RNN-модель на 1,5 відсоткових пункти за Accuracy та на 1,8 за F1-score. Встановлено монотонне зростання якості класифікації зі збільшенням кількості параметрів моделі в межах одного сімейства, а також перевагу моделей RoBERTa над BERT завдяки оптимізованій процедурі попереднього навчання. Порівняльний аналіз рекурентних та трансформерних підходів підтверджує доцільність застосування трансферного навчання на основі великих мовних моделей для задачі бінарної класифікації повідомлень про внесені зміни.

4. Досліджено застосування трансформерних моделей типу кодувальник-декодувальник для автоматичної генерації повідомлень про внесені зміни. Обґрунтовано вибір та досліджено дев'ять моделей: T5, FLAN-T5, CodeT5, CodeT5+ та BART. Показано, що моделі, орієнтовані на вихідні тексти програм (CodeT5, CodeT5+), суттєво перевершують моделі загального призначення: середній BLEU для CodeT5+ становить 6,55 проти менше 1,0 для T5 та FLAN-T5. Розроблено метод мовно-специфічного донавчання трансформерних моделей для систем контролю версій. Сформовано навчальні корпуси для шести мов програмування (C/C++, Ruby, Python, Java, JavaScript, PHP) загальним обсягом 271 504 навчальні пари. Метод передбачає незалежне донавчання моделі на корпусі кожної мови, що дозволяє моделі адаптуватися до синтаксичних та семантичних особливостей конкретної мови програмування.

5. Вперше запропоновано метод семантичного збагачення ідентифікаторів вихідного коду через автоматичне розкриття аббревіатур для підвищення якості автоматичної генерації повідомлень про внесені зміни. Розроблено п'ятиетапний конвеєр збагачення, що включає вилучення ідентифікаторів через побудову абстрактного синтаксичного дерева, сегментацію за конвенціями іменування, генерацію кандидатів розкриття на основі словникового підходу та контекстної моделі CodeBERT, семантичне ранжування кандидатів та формування збагачених текстових представлень. Експериментально підтверджено, що комбінація мовно-специфічного донавчання та семантичного збагачення забезпечує послідовне накопичення покращень: перехід від базової моделі T5 до донавченої CodeT5+ збільшує BLEU у 10,8 разу, а подальше застосування семантичного збагачення додає ще 35 % приросту. Найкращі результати продемонструвала модель codet5p-770m із мовно-специфічним донавчанням та семантичним збагаченням ідентифікаторів (BLEU = 11,15, ROUGE-1 = 44,62). Найвищі абсолютні результати після збагачення зафіксовано для JavaScript (BLEU = 16,47, BERTScore = 0,92), а найбільший відносний приріст – для Java (+51,5 % за BLEU) та C/C++ (+49,7 %).

6. Проведено порівняльний аналіз ефективності застосування технологій паралельних обчислень (GPGPU та TPU) для прискорення навчання нейронних класифікаторів. Встановлено, що GPU-акселерація (NVIDIA Tesla T4) забезпечує середнє прискорення у 45 разів порівняно з CPU та у 15 разів порівняно з TPU. Найвище прискорення GPU/CPU (56×) зафіксовано для моделі LSTM завдяки ефективному відображенню рекурентних операцій на тензорні ядра. Отримані результати вказують на можливість впровадження запропонованих методів оцінки якості та генерації повідомлень про внесені зміни у системи загального призначення з використанням стандартних графічних прискорювачів.

Таким чином, у дисертаційній роботі вирішено всі поставлені завдання дослідження. Запропоновані методи оцінки якості та автоматичної генерації повідомлень про внесені зміни на основі нейронних мереж різної архітектури дозволяють підвищити ефективність процесів супроводу програмного забезпечення шляхом автоматизації аналізу та створення текстових описів

модифікацій у вихідних текстах програм. Поєднання попереднього навчання на корпусах вихідного коду, мовно-специфічного донавчання та семантичного збагачення ідентифікаторів утворює оптимальний конвеєр для автоматичної генерації повідомлень про внесені зміни у системах контролю версій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Chacon, S., & Straub, B. (2014). *Pro Git*. Apress.
2. Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M., & Devanbu, P. (2009). The promises and perils of mining Git. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories* (pp. 1–10).
3. Herzig, K., & Zeller, A. (2021). Mining Software Repositories: Practices, Challenges, and Opportunities. *IEEE Software*, 38(1), 60–67.
<https://doi.org/10.1109/MS.2020.3013211>
4. Sommerville, I. (2016). *Software Engineering*. Boston: Pearson.
5. Lin, B., Zhang, H., & Zhou, Y. (2023). An Empirical Study on the Noise in Commit Messages. *Empirical Software Engineering*, 28(3), 1–29.
<https://doi.org/10.1007/s10664-022-10243-7>
6. Li, J., & Ahmed, I. (2023). *Commit Message Matters: Investigating Impact and Evolution of Commit Message Quality*. Retrieved from https://stairs.ics.uci.edu/papers/2023/Commit_Messages.pdf
7. Vaswani, A. & others (2017). Attention Is All You Need. In *Advances in Neural Information Processing Systems*.
8. Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of NAACL-HLT*.
9. Wang, Y. (2021). CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. . Retrieved from <https://aclanthology.org/2021.emnlp-main.685/>
10. Wang, Y., Le, H., Gotmare, A. D., Bui, N. D. Q., Li, J., & Hoi, S. C. H. (2023). CodeT5+: Open Code Large Language Models for Code Understanding and Generation. Retrieved from <https://arxiv.org/abs/2305.07922>
11. Jiang S., Armaly A., & McMillan C. (2017). Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*.

12. Dong, J. (2023). Revisiting Learning-Based Commit Message Generation. . Retrieved from <https://www.cs.purdue.edu/homes/lintan/publications/commit-icse23.pdf>
13. Rodriguez, P., Tan, L., & Kim, S. (2022). Automatically Classifying Software Changes Using Natural Language of Commit Messages. *IEEE Transactions on Software Engineering*, 48(6), 2156–2173. <https://doi.org/10.1109/TSE.2020.3034538>
14. Buse R. & Weimer W. (2010). Automatically documenting program changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*.
15. Zhao, Y., Xia, X., & Lo, D. (2020). Predicting Bug-Fixing Commits Using Textual Information. In *Proceedings of the 17th International Conference on Mining Software Repositories* (pp. 181–192). <https://doi.org/10.1145/3379597.3387473>
16. Shihab, E., & Hassan, A. E. (2021). Studying the Effectiveness of Lightweight Textual Features in Software Analytics. *IEEE Transactions on Software Engineering*, 47(9), 1943–1959. <https://doi.org/10.1109/TSE.2019.2956770>
17. Tantithamthavorn, C., McIntosh, S., Hassan, A. E., & Matsumoto, K. (2019). The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Transactions on Software Engineering*, 45(7), 683–711. <https://doi.org/10.1109/TSE.2018.2794977>
18. Manning, C. D., Raghavan, P., & Schutze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.
19. Jurafsky, D., & Martin, J. H. (2009). *Speech and Language Processing*. Prentice Hall.
20. GitHub Docs GitHub REST API. *GitHub Docs*. Retrieved 10 July 2025, from <https://docs.github.com/en/rest?apiVersion=2022-11-28>
21. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., & Damian, D. (2020). An In-Depth Study of the Promises and Perils of Mining GitHub. *Empirical Software Engineering*, 25, 449–486. <https://doi.org/10.1007/s10664-019-09706-1>
22. Salton, G., & Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5), 513–523.

23. Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3, 993–1022.
24. Hindle, A., Barr, E. T., Gabel, M., Su, Z., & Devanbu, P. (2012). On the Naturalness of Software. *Proceedings of the 34th International Conference on Software Engineering*, , 837–847.
25. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys*, 51(4).
26. Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735–1780.
27. Hoang, T., Nguyen, A. T., & Nguyen, T. N. (2020). CC2Vec: Distributed Representations of Code Changes. In *ICSE*.
28. Feng, Z., Guo, D., Tang, D., & others (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP*.
29. Rosa, B., De Oliveira, G., & Maldonado, J. (2020). Automatic Classification of Commit Messages using Hybrid Approaches. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*.
30. Commitizen: Formalizing Commit Messages for Consistent Project History. Retrieved from <https://commitizen.github.io/>
31. Spadini, D., Anquetil, N., & Lanza, M. (2018). PyDriller: Python Framework for Mining Software Repositories. *Journal of Software: Evolution and Process*, 30(9), e1969.
32. Dias, F., Alves, N., & Ribeiro, C. (2019). GitMiner: A Data Extraction Tool for Mining Git Repositories. In *Proceedings of the Brazilian Symposium on Software Engineering (SBES)*.
33. Wang, S., Liu, Z., Zhang, X., Tang, D., & Li, X. (2021). CodeT5: Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Findings of the Association for Computational Linguistics (ACL)*.
34. Maqbool, O., Qureshi, M., & Hussain, S. (2020). Challenges in Automatic Analysis of Commit Messages. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*.

35. Hata, H., Matsumoto, K., & Akiyama, Y. (2018). Predicting the Type of Code Changes with Text Mining and Machine Learning. In *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories (MSR)*.
36. Zampetti, F., Bacchelli, A., & Bird, C. (2021). Commit Message Quality and Standards in Continuous Integration. In *Proceedings of the International Conference on Software Engineering (ICSE)*.
37. Faragó, D., Färber, M., & Petrov, C. (2023). A Full-fledged Commit Message Quality Checker Based on Machine Learning. . Retrieved from https://faerber-lab.github.io/assets/pdf/publications/Commit-Message-Quality-Checker_COMPSAC2023.pdf
38. Shi, E. (2022). RACE: Retrieval-Augmented Commit mESsage Generation. . Retrieved from <https://aclanthology.org/2022.emnlp-main.372.pdf>
39. Ahmad, W. U. (2021). PLBART: Unified Pre-training for Program Understanding and Generation. . Retrieved from <https://aclanthology.org/2021.naacl-main.211/>
40. Wang, C., Zhang, L., & Zhang, X. (2024). Multi-grained contextual code representation learning for commit message generation (COMU). . <https://doi.org/10.1016/j.infsof.2023.107393>
41. Loyola, P., Marrese-Taylor, E., & Matsuo, Y. (2017). A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)* (pp. 287–292)Association for Computational Linguistics.
42. Huang, Y., Zheng, Q., Chen, C., Ni, Y., Liu, T., & Zhu, J. (2020). Learning to Generate Commit Messages from Source Code Changes. In *International Joint Conference on Neural Networks (IJCNN)* (pp. 1–8)IEEE.
43. Liu, T., Xia, X., Hassan, A. E., Lo, D., Xing, Z., & Wang, X. (2022). CommitBERT: Commit Message Generation Using Pre-Trained Programming Language Model. In *Proceedings of the 1st International Workshop on Natural Language Processing for Programming (NLP4Prog)* (pp. 33–39)Association for Computational Linguistics.

44. Xu, S., Yao, Y., Feng, F., & others (2019). Commit Message Generation for Source Code Changes. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 3975–3981)IJCAI Organization.
45. Nie, P., Jiang, S., Zhu, J., & McMillan, C. (2021). Race to the Top: An Improved Approach for Automatic Commit Message Generation. *IEEE Transactions on Software Engineering*, 49(2), 1–16.
46. Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)* (pp. 311–318)Association for Computational Linguistics.
47. Reiter, E. (2018). A Structured Review of the Validity of BLEU. *Computational Linguistics*, 44(3), 393–401.
48. Banerjee, S., & Lavie, A. (2005). METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization* (pp. 65–72)Association for Computational Linguistics.
49. Lin, C.-Y. (2004). ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out* (pp. 74–81)Association for Computational Linguistics.
50. Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., & Artzi, Y. (2020). BERTScore: Evaluating Text Generation with BERT. In *Proceedings of the 8th International Conference on Learning Representations (ICLR)*.
51. Sellam, T., Das, D., & Parikh, A. P. (2020). BLEURT: Learning Robust Metrics for Text Generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)* (pp. 7881–7892)Association for Computational Linguistics.
52. Vedantam, R., Zitnick, C. L., & Parikh, D. (2015). CIDEr: Consensus-based Image Description Evaluation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 4566–4575)IEEE.

53. Doddington, G. (2002). Automatic Evaluation of Machine Translation Quality Using N-gram Co-occurrence Statistics. In *Proceedings of the 2nd International Conference on Human Language Technology Research (HLT)* (pp. 138–145).
54. Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1), 37–46.
55. (2014). How to Write a Git Commit Message. *Cbeams*. Retrieved 30 July 2025, from <https://cbea.ms/git-commit/#seven-rules>
56. S. Pogorilyy & B. Semonov (2024). The Implementation of a Commit Messages Filter for Software Version Control Systems. In *The 9th International Conference on Control and Optimization with Industrial Applications* (pp. 175–179).
57. Tantithamthavorn, C., McIntosh, S., & Hassan, A. E. (2020). The Impact of Class Imbalance on the Evaluation of Defect Prediction Models. *IEEE Transactions on Software Engineering*, 46(11), 1200–1219. <https://doi.org/10.1109/TSE.2018.2876537>
58. N.V. Otten (2023). How To Use Text Normalization Techniques In NLP With Python [9 Ways]. *Spot Intelligence*. Retrieved 3 August 2025, from <https://spotintelligence.com/2023/01/25/text-normalization-techniquesnlp/>
59. N.K. Nissa (2022). Text Messages Classification using LSTM, Bi-LSTM, and GRU. *Medium*. Retrieved 30 July 2025, from <https://nzlul.medium.com/theclassification-of-text-messages-using-lstm-bi-lstm-and-gruf79b207f90ad>
60. Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Retrieved from <https://arxiv.org/abs/1810.04805>
61. Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., ... Stoyanov, V. (2019). RoBERTa: A Robustly Optimized BERT Pretraining Approach. Retrieved from <https://arxiv.org/abs/1907.11692>
62. Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2020). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. Retrieved from <https://arxiv.org/abs/1910.01108>

63. Семьонов, Б. О., & Погорілий, С. Д. (2025). Створення методу забезпечення якості коментарів у системах контролю версій на основі трансформерних моделей. *Реєстрація, Зберігання і Обробка Даних*, 27(2), 38–51. <https://doi.org/10.35681/1560-9189.2025.27.2.345503>
64. Semonov, B. O., & Pogorilyy, S. D. (2025). Research of the application of GPGPU and TPU technologies for ensuring comment quality in version control systems. *PROBLEMS IN PROGRAMMING*, (1), 24–37. <https://doi.org/10.15407/pp2025.01.024>
65. Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 1724–1734) Association for Computational Linguistics. <https://doi.org/10.3115/v1/D14-1179>
66. TensorFlow (2019). TensorFlow. *TensorFlow*. Retrieved 8 July 2025, from <https://www.tensorflow.org/>
67. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15, 1929–1958.
68. Schuster, M., & Paliwal, K. K. (1997). Bidirectional Recurrent Neural Networks. *IEEE Transactions on Signal Processing*, 45(11), 2673–2681. <https://doi.org/10.1109/78.650093>
69. Loshchilov, I., & Hutter, F. (2019). Decoupled Weight Decay Regularization. In *Proceedings of the ICLR* (pp. 1–8).
70. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. In *Proceedings of the 1st International Conference on Learning Representations (ICLR) Workshop*.
71. Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2017). Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics*, 5, 135–146. https://doi.org/10.1162/tacl_a_00051

72. Pennington, J., Socher, R., & Manning, C. D. (2014). GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 1532–1543). <https://doi.org/10.3115/v1/D14-1162>
73. Efstathiou, V., Chatzilenas, C., & Spinellis, D. (2018). Word Embeddings for the Software Engineering Domain. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)* (pp. 38–41).
74. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention Is All You Need. In *Advances in Neural Information Processing Systems (NeurIPS)* (pp. 5998–6008).
75. Conneau, A., Khandelwal, K., Goyal, N., Chaudhary, V., Wenzek, G., Guzmán, F., ... Stoyanov, V. (2020). Unsupervised Cross-lingual Representation Learning at Scale. Retrieved from <https://arxiv.org/abs/1911.02116>
76. Google (2024). Google Colaboratory – FAQ. . Retrieved 30 July 2025, from <https://research.google.com/colaboratory/faq.html>
77. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., ... Liu, P. J. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140), 1–67.
78. Kudo, T., & Richardson, J. (2018). SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing* (pp. 66–71).
79. Longpre, S., Hou, L., Vu, T., Webson, A., Chung, H. W., Tay, Y., ... Roberts, A. (2023). The Flan Collection: Designing Data and Methods for Effective Instruction Tuning. Retrieved from <https://arxiv.org/abs/2301.13688>
80. Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., ... Zettlemoyer, L. (2020). BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics* (pp. 7871–7880).

81. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., ... Rush, A. M. (2020). HuggingFace's Transformers: State-of-the-art Natural Language Processing. Retrieved from <https://arxiv.org/abs/1910.03771>
82. Caruana, R. (1997). Multitask Learning. *Machine Learning*, 28(1), 41–75.
83. Pogorilyy, S. D., & Semonov, B. O. (2025). Language-specific fine-tuning of transformer models for commit message generation in version control systems. In *XXV International Scientific and Practical Conference "Information Technologies and Security" (ITS-2025)*. Kyiv, Ukraine: .
84. Semonov, B. O., & Pogorilyy, S. D. (2025). Semantic enrichment of source code identifiers through automatic abbreviation expansion for commit message generation. In *Information Technology and Implementation (Satellite): Conference Proceedings* (pp. 298–301). Kyiv, Ukraine: .
85. Newman, C., Decker, M., AlSuhaibani, R., Peruma, A., Kaushik, D., & Hill, E. (2019). An Empirical Study of Abbreviations and Expansions in Software Artifacts. . <https://doi.org/10.1109/ICSME.2019.00040>
86. Google (2024). Colaboratory – Google. *Research.google.com*. Retrieved 10 July 2025, from <https://research.google.com/colaboratory/faq.html>
87. Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). Bleu: A Method for Automatic Evaluation of Machine Translation. In P. Isabelle, E. Charniak, & D. Lin (eds.) *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics* (pp. 311–318). Philadelphia, Pennsylvania, USA: Association for Computational Linguistics. <https://doi.org/10.3115/1073083.1073135>
88. Lin, C.-Y. (2004). ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out* (pp. 74–81). Barcelona, Spain: Association for Computational Linguistics. Retrieved from <https://aclanthology.org/W04-1013/>
89. Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., & Artzi, Y. (2020). BERTScore: Evaluating Text Generation with BERT. Retrieved from <https://arxiv.org/abs/1904.09675>
90. Sellam, T., Das, D., & Parikh, A. P. (2020). BLEURT: Learning Robust Metrics for Text Generation. Retrieved from <https://arxiv.org/abs/2004.04696>

91. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. In *Proceedings of the 1st International Conference on Learning Representations (ICLR) Workshop*. Retrieved from <https://arxiv.org/abs/1301.3781>
92. Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2017). Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics*, 5, 135–146. https://doi.org/10.1162/tacl_a_00051
93. Pennington, J., Socher, R., & Manning, C. D. (2014). GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 1532–1543). <https://doi.org/10.3115/v1/D14-1162>
94. Semonov, B. O., & Pogorilyy, S. D. (2025). Experimental investigation of CodeTrans T5 and T5 models for commit message generation in version control systems. In *Proceedings of the XXI International Scientific Conference “Electronics and Applied Physics” (APHYS 2025)* (pp. 90–91). Kyiv, Ukraine: .
95. Semonov, B. O., & Pogorilyy, S. D. (2025). Application of transformer models for generating commit messages in version control systems. *Scientific papers of Donetsk National Technical University. Series: Informatics, Cybernetics and Computer Science*, 41(2), 17–29. <https://doi.org/10.31474/1996-1588-2025-2-41-17-29>
96. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., ... Liu, P. J. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140), 1–67.
97. Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., ... Zettlemoyer, L. (2020). BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics* (pp. 7871–7880).

ДОДАТОК А. ОЦІНЮВАННЯ ЯКОСТІ ПОВІДОМЛЕНЬ НА ОСНОВІ РЕКУРЕНТНИХ НЕЙРОННИХ МЕРЕЖ

```
# -*- coding: utf-8 -*-
"""
Додаток А. Оцінювання якості повідомлень про внесені зміни
на основі рекурентних нейронних мереж (LSTM, BiLSTM, GRU, BiGRU).

Середовище виконання: Google Colaboratory (Python 3.10, TensorFlow 2.x).
"""

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import pandas as pd
import numpy as np

# =====
# 1. Завантаження та підготовка даних
# =====

# Завантаження набору даних із CSV-файлу
DATA_PATH = 'commits8000.csv'
data_frame = pd.read_csv(DATA_PATH, usecols=['message', 'classnum'])
data_frame = data_frame.dropna()

# Перемішування даних для уникнення зміщення
data_frame = data_frame.sample(frac=1, random_state=0, ignore_index=True)

# Визначення кількості класів (0 – неякісне, 1 – якісне)
num_classes = len(data_frame.classnum.unique())

# Створення об'єкта Dataset із перетворенням міток у one-hot
data_set = tf.data.Dataset.from_tensor_slices(
    (data_frame.message,
     tf.one_hot(data_frame.classnum, num_classes))
)

# Розподіл на навчальну (80%) та валідаційну (20%) вибірки
validation_size = int(0.2 * len(data_set))
train_size = len(data_set) - validation_size
train_ds = data_set.take(train_size)
val_ds = data_set.skip(train_size)

# Пакедна обробка (batch mode)
BATCH_SIZE = 256
train_ds = train_ds.shuffle(BATCH_SIZE * 4).batch(BATCH_SIZE)
val_ds = val_ds.batch(BATCH_SIZE)

# =====
# 2. Токенізація та векторизація тексту
# =====

SEQUENCE_LENGTH = 512
MAX_TOKENS = 20000

vectorize_layer = layers.TextVectorization(
    standardize='lower_and_strip_punctuation',
    max_tokens=MAX_TOKENS,
    output_sequence_length=SEQUENCE_LENGTH
)

# Побудова словника на навчальних текстах
vectorize_layer.adapt(train_ds.map(lambda msg, lbl: msg))

# Векторизація текстів
```

```

def vectorize_msg(msg, label):
    return vectorize_layer(msg), label

train_ds = train_ds.map(vectorize_msg,
                        num_parallel_calls=tf.data.AUTOTUNE)
val_ds = val_ds.map(vectorize_msg,
                    num_parallel_calls=tf.data.AUTOTUNE)

# Кешування та попереднє завантаження
train_ds = train_ds.cache().prefetch(tf.data.AUTOTUNE)
val_ds = val_ds.cache().prefetch(tf.data.AUTOTUNE)

# =====
# 3. Параметри шару векторного представлення слів
# =====

INPUT_DIM = len(vectorize_layer.get_vocabulary())
EMBEDDING_DIM = 128

# =====
# 4. Функція навчання моделі з ранньою зупинкою
# =====

def train_model(model, model_name, metric='f1'):
    """Компіляція та навчання моделі з ранньою зупинкою."""
    monitor = 'val_f1_score' if metric == 'f1' else 'val_accuracy'
    metrics = ([keras.metrics.F1Score(average='macro')]
               if metric == 'f1' else ['accuracy'])

    model.compile(
        optimizer=keras.optimizers.AdamW(learning_rate=0.001),
        loss=keras.losses.BinaryCrossentropy(),
        metrics=metrics
    )

    es = keras.callbacks.EarlyStopping(
        monitor=monitor, mode='max',
        patience=50, restore_best_weights=True
    )

    history = model.fit(
        train_ds, validation_data=val_ds,
        epochs=500, callbacks=[es]
    )

    # Збереження історії навчання
    pd.DataFrame(history.history).to_csv(
        f'{model_name}.csv', index=False
    )

    # Оцінка на валідаційній вибірці
    prob = model.predict(val_ds)
    y_pred = prob.argmax(axis=1)
    y_true_oh = np.vstack(list(val_ds.map(lambda m, l: l)))
    y_true = y_true_oh.argmax(axis=1)

    acc = tf.keras.metrics.Accuracy()
    f1 = tf.keras.metrics.F1Score(average='macro')
    acc.update_state(y_true, y_pred)
    f1.update_state(y_true_oh, prob)

    print(f"Accuracy: {acc.result().numpy():.3f}")
    print(f"F1-Score: {f1.result().numpy():.3f}")
    return history

# =====
# 5. Архітектури моделей
# =====

```

```

# 5.1. Модель з Dense-шаром (MLP)
model_dense = keras.Sequential([
    layers.Embedding(INPUT_DIM, EMBEDDING_DIM),
    layers.Dropout(0.3),
    layers.GlobalAveragePooling1D(),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
train_model(model_dense, 'Dense_MLP')

# 5.2. LSTM з 48 нейронами
model_lstm = keras.Sequential([
    layers.Embedding(INPUT_DIM, EMBEDDING_DIM),
    layers.Dropout(0.3),
    layers.LSTM(48),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
train_model(model_lstm, 'LSTM48')

# 5.3. Двонаправлений LSTM (BiLSTM) з 24 нейронами
model_bilstm = keras.Sequential([
    layers.Embedding(INPUT_DIM, EMBEDDING_DIM),
    layers.Dropout(0.3),
    layers.Bidirectional(layers.LSTM(24)),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
train_model(model_bilstm, 'BiLSTM24')

# 5.4. Дворівневий BiLSTM (12+12 нейронів)
model_bilstm2 = keras.Sequential([
    layers.Embedding(INPUT_DIM, EMBEDDING_DIM),
    layers.Dropout(0.3),
    layers.Bidirectional(layers.LSTM(12, return_sequences=True)),
    layers.Dropout(0.3),
    layers.Bidirectional(layers.LSTM(12)),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
train_model(model_bilstm2, 'BiLSTM12-12')

# 5.5. Двонаправлений GRU (BiGRU) з 24 нейронами
model_bigru = keras.Sequential([
    layers.Embedding(INPUT_DIM, EMBEDDING_DIM),
    layers.Dropout(0.3),
    layers.Bidirectional(layers.GRU(24)),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
train_model(model_bigru, 'BiGRU24')

# 5.6. Дворівневий BiGRU (12+12 нейронів)
model_bigru2 = keras.Sequential([
    layers.Embedding(INPUT_DIM, EMBEDDING_DIM),
    layers.Dropout(0.3),
    layers.Bidirectional(layers.GRU(12, return_sequences=True)),
    layers.Dropout(0.3),
    layers.Bidirectional(layers.GRU(12)),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
train_model(model_bigru2, 'BiGRU12-12')

```

ДОДАТОК Б. ОЦІНЮВАННЯ ЯКОСТІ ПОВІДОМЛЕНЬ З ВИКОРИСТАННЯМ ПОПЕРЕДНЬО НАВЧЕНИХ ВЕКТОРНИХ ПРЕДСТАВЛЕНЬ СЛІВ

```
# -*- coding: utf-8 -*-
"""
Додаток Б. Оцінювання якості повідомлень із використанням
попередньо навчених векторних представлень слів
(Word2Vec, FastText, GloVe).

Середовище виконання: Google Colaboratory (Python 3.10, TensorFlow 2.x).
"""

import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.layers import (
    Embedding, Bidirectional, GRU, Dense,
    Dropout, GlobalAveragePooling1D
)
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from gensim.models import KeyedVectors

# =====
# 1. Завантаження та підготовка даних
# =====

DATA_PATH = 'commits8000.csv'
data_frame = pd.read_csv(DATA_PATH, usecols=['message', 'classnum'])
data_frame = data_frame.dropna()
data_frame = data_frame.sample(frac=1, random_state=0, ignore_index=True)

# Гіперпараметри токенизації
MAX_NB_WORDS = 20000
MAX_SEQUENCE_LENGTH = 100

# Токенізація тексту
tokenizer = Tokenizer(num_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(data_frame['message'])
sequences = tokenizer.texts_to_sequences(data_frame['message'])
X = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)
y = data_frame['classnum'].values

# =====
# 2. Конфігурація моделей векторних представлень
# =====

MODELS_PATH = 'Models/'
EMBEDDING_CONFIGS = [
    {'name': 'word2vec-google-news-300', 'dim': 300},
    {'name': 'SO_vectors_200', 'dim': 200},
    {'name': 'fasttext-wiki-news-subwords-300', 'dim': 300},
    {'name': 'glove-twitter-200', 'dim': 200},
    {'name': 'glove-wiki-gigaword-50', 'dim': 50},
    {'name': 'glove-wiki-gigaword-300', 'dim': 300},
]

RUNS = 5
PATIENCE = 30
EPOCHS = 500
```

```

# =====
# 3. Побудова матриці ембеддінгів
# =====

def create_embedding_matrix(model_name, emb_dim):
    """Створення матриці ембеддінгів на основі попередньо
    навченої моделі Word2Vec / GloVe / FastText."""
    w2v = KeyedVectors.load_word2vec_format(
        f'{MODELS_PATH}{model_name}.bin', binary=True
    )
    word_index = tokenizer.word_index
    num_words = min(MAX_NB_WORDS, len(word_index)) + 1
    matrix = np.zeros((num_words, emb_dim))
    for word, i in word_index.items():
        if i < MAX_NB_WORDS and word in w2v:
            matrix[i] = w2v[word]
    return num_words, matrix

# =====
# 4. Створення моделей нейронних мереж
# =====

def create_model(num_words, emb_matrix, emb_dim, nn_type):
    """Створення моделі MLP або BiGRU із заданими ембедінгами."""
    emb_layer = Embedding(
        input_dim=num_words, output_dim=emb_dim,
        weights=[emb_matrix],
        input_length=MAX_SEQUENCE_LENGTH,
        trainable=False # Ваги ембеддінгів зафіксовані
    )

    if nn_type == 'MLP':
        return keras.Sequential([
            emb_layer,
            Dropout(0.3),
            GlobalAveragePooling1D(),
            Dropout(0.3),
            Dense(48, activation='relu'),
            Dropout(0.3),
            Dense(1, activation='sigmoid')
        ])

    if nn_type == 'BiGRU':
        return keras.Sequential([
            emb_layer,
            Dropout(0.3),
            Bidirectional(GRU(24, return_sequences=False)),
            Dropout(0.3),
            Dense(1, activation='sigmoid')
        ])

    raise ValueError(f"Невідомий тип мережі: {nn_type}")

# =====
# 5. Навчання та оцінка моделей
# =====

def train_and_evaluate(model, model_name, metric='accuracy'):
    """Навчання моделі з усередненням за RUNS запусків."""
    monitor = ('val_accuracy' if metric == 'accuracy'
               else 'val_f1_score')
    met = (['accuracy'] if metric == 'accuracy'
           else [keras.metrics.F1Score(average='weighted')])

    model.compile(
        optimizer=keras.optimizers.AdamW(learning_rate=0.001),
        loss=keras.losses.BinaryCrossentropy(),

```

```

        metrics=met
    )

    es = keras.callbacks.EarlyStopping(
        monitor=monitor, mode='max',
        patience=PATIENCE, restore_best_weights=True
    )

    results = []
    for run in range(RUNS):
        hist = model.fit(
            X_train, y_train, batch_size=32,
            epochs=EPOCHS,
            validation_data=(X_test, y_test),
            callbacks=[es]
        )
        scores = model.evaluate(X_test, y_test)
        results.append(scores)

        pd.DataFrame(hist.history).to_csv(
            f'{model_name}_{metric}_run{run+1}.csv',
            index=False
        )

    # Усереднення результатів
    avg = np.mean(results, axis=0)
    std = np.std(results, axis=0)
    for i, m in enumerate(model.metrics_names):
        print(f"{m}: {avg[i]:.4f} +/- {std[i]:.4f}")

# =====
# 6. Основний цикл: перебір усіх моделей ембеддінгів
# =====

for cfg in EMBEDDING_CONFIGS:
    print(f"\n{'='*60}")
    print(f"Модель ембеддінгів: {cfg['name']}")
    print(f"{'='*60}")

    num_words, emb_matrix = create_embedding_matrix(
        cfg['name'], cfg['dim']
    )

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )
    y_train = y_train.astype(np.float32).reshape(-1, 1)
    y_test = y_test.astype(np.float32).reshape(-1, 1)

    # Навчання BiGRU з кожною моделлю ембеддінгів
    model = create_model(num_words, emb_matrix,
                        cfg['dim'], 'BiGRU')
    train_and_evaluate(model, cfg['name'], 'accuracy')

```

ДОДАТОК В. ОЦІНЮВАННЯ ЯКОСТІ ПОВІДОМЛЕНЬ НА ОСНОВІ ТРАНСФОРМЕРНИХ МОДЕЛЕЙ

```
# -*- coding: utf-8 -*-
"""
Додаток В. Оцінювання якості повідомлень на основі
трансформерних моделей (Transformer, BERT, RoBERTa,
DistilBERT, XLM-RoBERTa).

Середовище виконання: Google Colaboratory (Python 3.10,
TensorFlow 2.x, KerasNLP).
"""

import keras_nlp
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.utils import model_to_dot
import pandas as pd
import numpy as np
import time

# =====
# 1. Завантаження та підготовка даних
# =====

DATA_PATH = 'commits8000.csv'
data_frame = pd.read_csv(DATA_PATH, usecols=['message', 'classnum'])
data_frame = data_frame.dropna()
data_frame = data_frame.sample(frac=1, random_state=0, ignore_index=True)

num_classes = len(data_frame.classnum.unique())

data_set = tf.data.Dataset.from_tensor_slices(
    (data_frame.message,
     tf.one_hot(data_frame.classnum, num_classes))
)

validation_size = int(0.2 * len(data_set))
train_size = len(data_set) - validation_size

# =====
# 2. Підготовка вибірок із токенизацією
# =====

BATCH_SIZE = 16
SEQUENCE_LENGTH = 128

# Вибірки з TextVectorization (для Transformer)
train_ds_vec = data_set.take(train_size)
val_ds_vec = data_set.skip(train_size)

train_ds_vec = train_ds_vec.shuffle(BATCH_SIZE * 4).batch(BATCH_SIZE)
val_ds_vec = val_ds_vec.batch(BATCH_SIZE)

vectorize_layer = layers.TextVectorization(
    standardize='lower_and_strip_punctuation',
    max_tokens=20000,
    output_sequence_length=SEQUENCE_LENGTH
)
vectorize_layer.adapt(train_ds_vec.map(lambda m, l: m))

def vectorize_msg(msg, label):
    return vectorize_layer(msg), label

train_ds_vec = train_ds_vec.map(
```

```

        vectorize_msg, num_parallel_calls=tf.data.AUTOTUNE
    ).cache().prefetch(tf.data.AUTOTUNE)
val_ds_vec = val_ds_vec.map(
    vectorize_msg, num_parallel_calls=tf.data.AUTOTUNE
).cache().prefetch(tf.data.AUTOTUNE)

# Вибірки без TextVectorization (для BERT-моделей)
train_ds_raw = (data_set.take(train_size)
    .shuffle(BATCH_SIZE * 4).batch(BATCH_SIZE)
    .cache().prefetch(tf.data.AUTOTUNE))
val_ds_raw = (data_set.skip(train_size)
    .batch(BATCH_SIZE)
    .cache().prefetch(tf.data.AUTOTUNE))

# =====
# 3. Декоратор для вимірювання часу навчання
# =====

LOG_FILE = 'time_log.txt'

def timed_run(func):
    """Декоратор для вимірювання та логування часу навчання."""
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        elapsed = time.time() - start
        h = int(elapsed // 3600)
        m = int((elapsed % 3600) // 60)
        s = int(elapsed % 60)
        msg = f"{h} год {m} хв {s} сек" if h else \
            f"{m} хв {s} сек" if m else f"{s} сек"
        with open(LOG_FILE, 'a') as f:
            f.write(f"Час навчання: {msg}\n")
        return result
    return wrapper

# =====
# 4. Універсальна функція навчання
# =====

INPUT_DIM = len(vectorize_layer.get_vocabulary())
EMBEDDING_DIM = 128

@timed_run
def train_model(model, model_name, metric='f1',
                train_data=None, val_data=None):
    """Навчання моделі з ранньою зупинкою та збереженням
    історії навчання у CSV-файл."""
    if train_data is None:
        train_data = train_ds_vec
    if val_data is None:
        val_data = val_ds_vec

    monitor = ('val_f1_score' if metric == 'f1'
               else 'val_accuracy')
    metrics = ([keras.metrics.F1Score(average='macro')]
               if metric == 'f1' else ['accuracy'])

    model.compile(
        optimizer=keras.optimizers.AdamW(learning_rate=0.001),
        loss=keras.losses.BinaryCrossentropy(),
        metrics=metrics
    )

    es = keras.callbacks.EarlyStopping(
        monitor=monitor, mode='max',
        patience=25, restore_best_weights=True
    )

```

```

hist = model.fit(
    train_data, validation_data=val_data,
    epochs=500, callbacks=[es]
)

pd.DataFrame(hist.history).to_csv(
    f'{model_name}-{metric}.csv', index=False
)

# Обчислення метрик на валідаційній вибірці
prob = model.predict(val_data)
y_pred = prob.argmax(axis=1)
y_true_oh = np.vstack(list(val_data.map(lambda m, l: l)))
y_true = y_true_oh.argmax(axis=1)

acc = tf.keras.metrics.Accuracy()
f1m = tf.keras.metrics.F1Score(average='macro')
acc.update_state(y_true, y_pred)
f1m.update_state(y_true_oh, prob)

print(f"[{model_name}] Accuracy={acc.result():.3f}, "
      f"F1={f1m.result():.3f}")

# =====
# 5. Модель Transformer Encoder
# =====

model_transformer = keras.Sequential([
    layers.Embedding(INPUT_DIM, EMBEDDING_DIM),
    layers.Dropout(0.3),
    keras_nlp.layers.TransformerEncoder(
        intermediate_dim=512, num_heads=4, dropout=0.3
    ),
    layers.GlobalAveragePooling1D(),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])

train_model(model_transformer, 'Transformer', 'accuracy')
train_model(model_transformer, 'Transformer', 'f1')

# =====
# 6. Моделі BERT (серія пресетів)
# =====

BERT_PRESETS = [
    'bert_tiny_en_uncased', 'bert_small_en_uncased',
    'bert_medium_en_uncased', 'bert_base_en_uncased',
    'bert_large_en_uncased',
]

for preset in BERT_PRESETS:
    model = keras_nlp.models.BertClassifier.from_preset(
        preset, num_classes=num_classes, load_weights=False
    )
    train_model(model, f'BERT-{preset}', 'f1',
                train_ds_raw, val_ds_raw)

# =====
# 7. Моделі RoBERTa та XLM-RoBERTa
# =====

for preset in ['roberta_base_en', 'roberta_large_en']:
    model = keras_nlp.models.RobertaClassifier.from_preset(
        preset, num_classes=num_classes, load_weights=False
    )
    train_model(model, f'RoBERTa-{preset}', 'f1',
                train_ds_raw, val_ds_raw)

```

```

for preset in ['xlm_roberta_base_multi',
               'xlm_roberta_large_multi']:
    model = keras_nlp.models.XLMRobertaClassifier.from_preset(
        preset, num_classes=num_classes, load_weights=False
    )
    train_model(model, f'XLM-RoBERTa-{preset}', 'f1',
                train_ds_raw, val_ds_raw)

# =====
# 8. Моделі DistilBERT
# =====

for preset in keras_nlp.models.DistilBertClassifier.presets:
    model = keras_nlp.models.DistilBertClassifier.from_preset(
        preset, num_classes=num_classes, load_weights=False
    )
    train_model(model, f'DistilBERT-{preset}', 'f1',
                train_ds_raw, val_ds_raw)

```

ДОДАТОК Г. АВТОМАТИЧНА ГЕНЕРАЦІЯ ПОВІДОМЛЕНЬ ПРО ВНЕСЕНІ ЗМІНИ НА ОСНОВІ МОДЕЛЕЙ КОДУВАЛЬНИК- ДЕКОДУВАЛЬНИК

```
# -*- coding: utf-8 -*-
"""
Додаток Г. Автоматична генерація повідомлень про внесені зміни
на основі моделей кодувальник-декодувальник (CodeT5, CodeT5+,
T5, FLAN-T5, BART).

Середовище виконання: Google Colaboratory (Python 3.10,
HuggingFace Transformers).
"""

from transformers import (
    AutoTokenizer, AutoModelForSeq2SeqLM,
    Seq2SeqTrainer, Seq2SeqTrainingArguments,
    EarlyStoppingCallback
)
from evaluate import load
from datasets import Dataset
import torch
import pandas as pd
import numpy as np
from datetime import datetime

# =====
# 1. Конфігурація моделей для генерації
# =====

MODELS = [
    'Salesforce/codet5-small',
    'Salesforce/codet5-base',
    'Salesforce/codet5-large',
    'Salesforce/codet5p-220m',
    'Salesforce/codet5p-770m',
    't5-base',
    't5-large',
    'google/flan-t5-base',
    'google/flan-t5-xl',
    'facebook/bart-large',
]

LANGUAGES = ['cpp', 'ruby', 'python', 'java',
             'javascript', 'php']

# Максимальна довжина вхідної послідовності та цільового тексту
MAX_INPUT_LEN = 512
MAX_TARGET_LEN = 128
MAX_DATASET_SIZE = 25000

# =====
# 2. Завантаження та попередня обробка даних
# =====

def load_dataset(lang):
    """Завантаження та фільтрація навчальних даних для
    заданої мови програмування."""
    path = f'train_data_{lang}.csv'
    df = pd.read_csv(path, usecols=['input', 'target'])
    df = df.dropna()
    df = df.sample(frac=1, random_state=42).reset_index(drop=True)
    return Dataset.from_pandas(df)
```

```

def preprocess(examples, tokenizer):
    """Токенізація вхідних diff та цільових повідомлень
    із фільтрацією за довжиною."""
    inputs = examples['input']
    targets = examples['target']

    # Фільтрація прикладів, що перевищують ліміти токенів
    inp_lens = [len(tokenizer.encode(t, truncation=False))
                 for t in inputs]
    tgt_lens = [len(tokenizer.encode(t, truncation=False))
                 for t in targets]

    filt_inp, filt_tgt = [], []
    for inp, tgt, il, tl in zip(inputs, targets,
                                inp_lens, tgt_lens):
        if il <= MAX_INPUT_LEN and tl <= MAX_TARGET_LEN:
            filt_inp.append(inp)
            filt_tgt.append(tgt)

    if not filt_inp:
        return {}

    model_inputs = tokenizer(
        filt_inp, max_length=MAX_INPUT_LEN,
        truncation=True, padding='max_length'
    )
    with tokenizer.as_target_tokenizer():
        labels = tokenizer(
            filt_tgt, max_length=MAX_TARGET_LEN,
            truncation=True, padding='max_length'
        )
    model_inputs['labels'] = labels['input_ids']
    return model_inputs

# =====
# 3. Метрики оцінювання якості генерації
# =====

bleu_metric = load('bleu')
rouge_metric = load('rouge')
bertscore_metric = load('bertscore')
bleurt_metric = load('bleurt', config_name='BLEURT-20')

def compute_metrics(eval_pred, tokenizer):
    """Обчислення метрик BLEU, ROUGE, BERTScore, BLEURT."""
    predictions, labels = eval_pred
    vocab_size = tokenizer.vocab_size
    pad_id = tokenizer.pad_token_id or 0

    if isinstance(predictions, tuple):
        predictions = predictions[0]

    predictions = np.clip(
        np.array(predictions, dtype=np.int32), 0, vocab_size - 1
    )
    labels = np.where(
        np.array(labels, dtype=np.int32) == -100, pad_id, labels
    )
    labels = np.clip(labels, 0, vocab_size - 1)

    preds = [p.strip()[512] for p in
               tokenizer.batch_decode(predictions,
                                       skip_special_tokens=True)]
    refs = [r.strip()[512] for r in
             tokenizer.batch_decode(labels,
                                     skip_special_tokens=True)]

    # Базові метрики

```

```

bleu = bleu_metric.compute(
    predictions=preds, references=[[r] for r in refs]
)
rouge = rouge_metric.compute(
    predictions=preds, references=refs
)

# Повільні метрики – на 20% вибірки
n = max(1, int(len(preds) * 0.2))
bs = bertscore_metric.compute(
    predictions=preds[:n], references=refs[:n], lang='en'
)
bl = bleurt_metric.compute(
    predictions=preds[:n], references=refs[:n]
)

return {
    'bleu': round(bleu['bleu'] * 100, 2),
    'rouge1': round(rouge['rouge1'] * 100, 2),
    'rouge2': round(rouge['rouge2'] * 100, 2),
    'rougeL': round(rouge['rougeL'] * 100, 2),
    'bertscore': round(sum(bs['f1']) / len(bs['f1']), 4),
    'bleurt': round(sum(bl['scores']) / len(bl['scores']), 4),
}

# =====
# 4. Навчання моделі генерації
# =====

def train_generation(model_name, lang):
    """Повний цикл: завантаження моделі, токенизація,
    навчання та збереження результатів."""
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModelForSeq2SeqLM.from_pretrained(model_name)

    dataset = load_dataset(lang)
    tokenized = dataset.map(
        lambda ex: preprocess(ex, tokenizer),
        batched=True, remove_columns=['input', 'target']
    )

    # Обмеження розміру вибірки
    if len(tokenized) > MAX_DATASET_SIZE:
        tokenized = tokenized.select(range(MAX_DATASET_SIZE))

    splits = tokenized.train_test_split(test_size=0.2)
    result_path = f'results/{model_name}-{lang}/'

    args = Seq2SeqTrainingArguments(
        output_dir=result_path,
        eval_strategy='epoch',
        save_strategy='no',
        logging_strategy='epoch',
        per_device_train_batch_size=8,
        per_device_eval_batch_size=8,
        fp16=True,
        predict_with_generate=True,
        generation_num_beams=4,
        num_train_epochs=50,
        metric_for_best_model='bleu',
        greater_is_better=True,
    )

    trainer = Seq2SeqTrainer(
        model=model, args=args,
        train_dataset=splits['train'],
        eval_dataset=splits['test'],
        tokenizer=tokenizer,
        compute_metrics=lambda ep: compute_metrics(
            ep, tokenizer

```

```

    ),
    callbacks=[EarlyStoppingCallback(
        early_stopping_patience=3
    )]
)

trainer.train()

# Збереження результатів навчання
ts = datetime.now().strftime('%Y%m%d%H%M%S')
df = pd.DataFrame(trainer.state.log_history)
df.to_csv(f'{result_path}metrics_{lang}_{ts}.csv',
          index=False)

# Збереження донавченої моделі
model.save_pretrained(f'{result_path}finetuned_model')
tokenizer.save_pretrained(f'{result_path}finetuned_model')

# =====
# 5. Генерація повідомлення для заданого diff
# =====

def generate_message(model, tokenizer, git_diff):
    """Генерація commit-повідомлення для заданого diff."""
    inputs = tokenizer(
        'summarize: ' + git_diff,
        return_tensors='pt', truncation=True,
        max_length=MAX_INPUT_LEN
    )
    outputs = model.generate(
        **inputs, max_length=64,
        num_beams=4, early_stopping=True
    )
    return tokenizer.decode(outputs[0],
                           skip_special_tokens=True)

# =====
# 6. Запуск навчання
# =====

if __name__ == '__main__':
    for model_name in MODELS:
        for lang in LANGUAGES:
            print(f'\n=== {model_name} / {lang} ===')
            train_generation(model_name, lang)

```

ДОДАТОК Д. АВТОМАТИЧНА РОЗМІТКА НАВЧАЛЬНОЇ ВИБІРКИ

```
# -*- coding: utf-8 -*-
"""
Додаток Д. Автоматична розмітка навчальної вибірки для
генерації повідомлень на основі навченого класифікатора.

Середовище виконання: Google Colaboratory (Python 3.10,
TensorFlow 2.x, KerasNLP).
"""

import keras_nlp
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import pandas as pd
import numpy as np

# =====
# 1. Завантаження розміченої вибірки (для навчання)
# =====

LABELED_PATH = 'commits8000.csv'
UNLABELED_PATH = 'commits_unmarked_large.csv'

df_labeled = pd.read_csv(LABELED_PATH,
                        usecols=['message', 'classnum'])
df_labeled = df_labeled.dropna()
df_labeled = df_labeled.sample(
    frac=1, random_state=0, ignore_index=True
)

num_classes = len(df_labeled.classnum.unique())

# Завантаження нерозміченої вибірки
df_unlabeled = pd.read_csv(
    UNLABELED_PATH,
    usecols=['commitid', 'message', 'classnum']
)
df_unlabeled = df_unlabeled.dropna()
print(f"Нерозмічених записів: {len(df_unlabeled)}")

# =====
# 2. Підготовка навчальних вибірок
# =====

ds_labeled = tf.data.Dataset.from_tensor_slices(
    (df_labeled.message,
     tf.one_hot(df_labeled.classnum, num_classes))
)
ds_unlabeled = tf.data.Dataset.from_tensor_slices(
    (df_unlabeled.message,
     tf.one_hot(df_unlabeled.classnum, num_classes))
)

val_size = int(0.2 * len(ds_labeled))
train_size = len(ds_labeled) - val_size

BATCH_SIZE = 128
train_ds = (ds_labeled.take(train_size)
            .shuffle(BATCH_SIZE * 4).batch(BATCH_SIZE))
val_ds = ds_labeled.skip(train_size).batch(BATCH_SIZE)
unlabeled_ds = ds_unlabeled.batch(BATCH_SIZE)

# =====
# 3. Токенізація
```

```

# =====

SEQUENCE_LENGTH = 128

vectorize_layer = layers.TextVectorization(
    standardize='lower_and_strip_punctuation',
    max_tokens=20000,
    output_sequence_length=SEQUENCE_LENGTH
)
vectorize_layer.adapt(train_ds.map(lambda m, l: m))

def vectorize_msg(msg, label):
    return vectorize_layer(msg), label

train_ds = (train_ds.map(vectorize_msg,
    num_parallel_calls=tf.data.AUTOTUNE)
    .cache()).prefetch(tf.data.AUTOTUNE)
val_ds = (val_ds.map(vectorize_msg,
    num_parallel_calls=tf.data.AUTOTUNE)
    .cache()).prefetch(tf.data.AUTOTUNE)
unlabeled_ds = (unlabeled_ds.map(vectorize_msg,
    num_parallel_calls=tf.data.AUTOTUNE)
    .cache()).prefetch(tf.data.AUTOTUNE)

# =====
# 4. Побудова та навчання моделі Transformer Encoder
# =====

INPUT_DIM = len(vectorize_layer.get_vocabulary())
EMBEDDING_DIM = 128

model = keras.Sequential([
    layers.Embedding(INPUT_DIM, EMBEDDING_DIM),
    layers.Dropout(0.3),
    keras_nlp.layers.TransformerEncoder(
        intermediate_dim=512, num_heads=4, dropout=0.3
    ),
    layers.GlobalAveragePooling1D(),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])

model.compile(
    optimizer=keras.optimizers.AdamW(learning_rate=0.001),
    loss=keras.losses.BinaryCrossentropy(),
    metrics=['accuracy']
)

es = keras.callbacks.EarlyStopping(
    monitor='val_accuracy', mode='max',
    patience=25, restore_best_weights=True
)

model.fit(train_ds, validation_data=val_ds,
    epochs=500, callbacks=[es])

# =====
# 5. Оцінка якості класифікатора
# =====

prob = model.predict(val_ds)
y_pred = prob.argmax(axis=1)
y_true_oh = np.vstack(list(val_ds.map(lambda m, l: l)))
y_true = y_true_oh.argmax(axis=1)

accuracy = tf.keras.metrics.Accuracy()
f1 = tf.keras.metrics.F1Score(average='macro')
accuracy.update_state(y_true, y_pred)
f1.update_state(y_true_oh, prob)

```

```

print(f"Accuracy: {accuracy.result():.3f}")
print(f"F1-Score: {f1.result():.3f}")

# =====
# 6. Класифікація нерозміченої вибірки
# =====

prob_unlabeled = model.predict(unlabeled_ds)
df_unlabeled['classnum'] = prob_unlabeled.argmax(axis=1)

# Збереження результатів розмітки
df_unlabeled[['commitid', 'classnum']].to_csv(
    'output_labeled.csv', index=False
)
print(f"Збережено {len(df_unlabeled)} розмічених записів")

```

ДОДАТОК Е. СЦЕНАРІЙ ЗАВАНТАЖЕННЯ ДАНИХ ІЗ GITHUB REST API

Лістинг Е.1 -- SQL-сценарій розгортання бази даних

```
-- Сценарій розгортання бази даних
-- (PostgreSQL)

CREATE DATABASE phd;
CREATE SCHEMA repodata;

-- Таблиця репозиторіїв
CREATE TABLE repodata.repos (
    repoid SERIAL PRIMARY KEY,
    fullname VARCHAR(200) NOT NULL,
    descr VARCHAR(65535) NULL,
    procst SMALLINT NOT NULL DEFAULT 0,
    instime TIMESTAMP NOT NULL,
    updtme TIMESTAMP NOT NULL
);

CREATE INDEX repos_procst_idx
    ON repodata.repos (procst ASC);

-- Таблиця повідомлень про внесені зміни
CREATE TABLE repodata.commits (
    commitid BIGSERIAL PRIMARY KEY,
    repoid INT NOT NULL,
    shacommith VARCHAR(50) NOT NULL,
    message TEXT NOT NULL,
    cmtdate TIMESTAMP NULL,
    procst SMALLINT NOT NULL DEFAULT 0,
    username VARCHAR(50) NULL,
    instime TIMESTAMP NOT NULL,
    updtme TIMESTAMP NOT NULL
);

CREATE INDEX commits_repoid_idx
    ON repodata.commits (repoid ASC);
CREATE INDEX commits_procst_idx
    ON repodata.commits (procst ASC);

ALTER TABLE repodata.commits
    ADD CONSTRAINT fk_commits_repoid_repos
    FOREIGN KEY (repoid)
    REFERENCES repodata.repos (repoid)
    ON UPDATE RESTRICT ON DELETE RESTRICT;

-- Таблиця попередніх внесених змін
CREATE TABLE repodata.parents (
    parentid BIGSERIAL PRIMARY KEY,
    commitid BIGINT NOT NULL,
    shaparent VARCHAR(50) NOT NULL
);

ALTER TABLE repodata.parents
    ADD CONSTRAINT fk_parents_commitid_commits
    FOREIGN KEY (commitid)
    REFERENCES repodata.commits (commitid)
    ON UPDATE RESTRICT ON DELETE RESTRICT;

-- Таблиця файлів
CREATE TABLE repodata.files (
    fileid BIGSERIAL PRIMARY KEY,
    commitid BIGINT NOT NULL,
    filename VARCHAR(1024) NOT NULL,
    status VARCHAR(40) NOT NULL,
```

```

        additions INT          NOT NULL,
        deletions INT          NOT NULL,
        rawurl    VARCHAR(1024) NULL,
        diff      TEXT          NULL,
        procst    SMALLINT      NOT NULL DEFAULT 0,
        instime   TIMESTAMP     NOT NULL,
        updttime  TIMESTAMP     NOT NULL
    );

CREATE INDEX commitid_idx
    ON repodata.files (commitid ASC);
CREATE INDEX files_procst_idx
    ON repodata.files (procst ASC);

ALTER TABLE repodata.files
    ADD CONSTRAINT fk_files_commitid_commits
    FOREIGN KEY (commitid)
    REFERENCES repodata.commits (commitid)
    ON UPDATE RESTRICT ON DELETE RESTRICT;

-- Таблиця тексту файлів
CREATE TABLE repodata.filetext (
    fileid BIGSERIAL PRIMARY KEY,
    ftext  TEXT NOT NULL
);

ALTER TABLE repodata.filetext
    ADD CONSTRAINT fk_filetext_fileid_files
    FOREIGN KEY (fileid)
    REFERENCES repodata.files (fileid)
    ON UPDATE RESTRICT ON DELETE RESTRICT;

```

Лістинг Е.2 -- Спільний модуль CommonTypes.py

```

# -*- coding: utf-8 -*-
"""
CommonTypes.py -- спільний модуль для сценаріїв
завантаження даних із GitHub REST API.

Містить параметри підключення до БД, перелік
станів обробки та функцію виконання HTTP-запитів
з автоматичним перемиканням між токенами.
"""

import requests
import json
import time
from datetime import datetime
from enum import IntEnum

# Параметри підключення до бази даних PostgreSQL
connParams = {
    'hostip': '192.168.0.111',
    'login': 'semonov',
    'pwd': '****',
    'db': 'phd'
}

class ProcStType(IntEnum):
    """Стани обробки записів у БД."""
    WAIT = 0 # Чекає на обробку
    OK = 1 # Успішно оброблено
    FAIL = 2 # Помилка під час обробки
    EMPTY_REPO = 3 # Порожній репозиторій

# Токени для доступу до GitHub REST API
# (4 облікові записи для збільшення ліміту

```

```

# з 5000 до 20000 запитів/год)
tokens = ['***', '***', '***', '***']

def makeGitReq(url, isParseJson=True):
    """Виконання GET-запиту до GitHub REST API.

    Автоматично перемикає токени у разі вичерпання
    ліміту запитів (HTTP 403, rate limit exceeded).

    Args:
        url: URL-адреса запиту до GitHub API.
        isParseJson: True для розбору відповіді як JSON.

    Returns:
        (isError, resp): кортеж з прапорцем помилки
        та тілом відповіді.
    """
    isError = False
    resp = None
    isReachedLimit = True

    while isReachedLimit:
        for token in tokens:
            try:
                headers = {
                    'Accept': 'application/vnd.github+json',
                    'Authorization': f'Bearer {token}',
                    'X-GitHub-API-Version': '2022-11-28'
                }
                response = requests.get(url, headers=headers)
                headersResp = response.headers

                isError = (response.status_code != 200)
                resp = (json.loads(response.text)
                        if isError or isParseJson
                        else response.text)

                isReachedLimit = (
                    isError and 'message' in resp
                    and 'API rate limit exceeded'
                    in resp['message']
                )

                if isError and not isReachedLimit:
                    print('=' * 40)
                    print(datetime.now())
                    print(response.text)
                    if 'X-RateLimit-Remaining' in headersResp:
                        print('Remaining: '
                              + headersResp['X-RateLimit-Remaining'])
                    print('=' * 40)

            except Exception as e:
                print(e)
                isError = True
                isReachedLimit = False

        if not isReachedLimit:
            break

    if isReachedLimit:
        print('All tokens reached limits')
        time.sleep(300) # Очікування 5 хвилин

    return isError, resp

```

Лістинг Е.3 -- Сценарій завантаження репозиторіїв

```
# -*- coding: utf-8 -*-
```

```

"""
Сценарій завантаження переліку репозиторіїв
із GitHub REST API у таблицю repodata.repos.
"""

from CommonTypes import *
import psycopg2

def main():
    global connParams
    conn = psycopg2.connect(
        dbname=connParams['db'],
        user=connParams['login'],
        password=connParams['pwd'],
        host=connParams['hostip']
    )
    cursor = conn.cursor()

    # Визначення останнього завантаженого ID
    cursor.execute(
        "SELECT MAX(repoid) FROM repodata.repos"
    )
    repoid = cursor.fetchone()[0]
    isReqError = False

    try:
        while not isReqError:
            url = ("https://api.github.com/repositories"
                   f"?since={repoid}")
            isReqError, repos = makeGitReq(url)

            if (not isReqError
                and type(repos) is list
                and 'full_name' in repos[0]):
                for repo in repos:
                    repoid = int(repo['id'])
                    cursor.execute(
                        'INSERT INTO repodata.repos '
                        '(repoid, fullname, descr, '
                        ' instime, updttime) '
                        'VALUES (%s, %s, %s, '
                        ' CURRENT_TIMESTAMP, '
                        ' CURRENT_TIMESTAMP)',
                        (repoid,
                         repo['full_name'],
                         repo['description'])
                    )
                    cursor.execute('COMMIT')

    except Exception as e:
        print(f'{e}. repoid: {repoid}')

    cursor.close()
    conn.close()

main()

```

Лістинг Е.4 -- Сценарій завантаження повідомлень

```

# -*- coding: utf-8 -*-
"""
Сценарій завантаження повідомлень про внесені зміни
із GitHub REST API у таблицю repodata.commits.
"""

from CommonTypes import *
import psycopg2

```

```

cursorIns = None

def setProcst(repoid, procst):
    """Оновлення стану обробки репозиторію."""
    global cursorIns
    cursorIns.execute(
        'UPDATE repodata.repos '
        'SET procst = %s, '
        '    uptime = CURRENT_TIMESTAMP '
        'WHERE repoid = %s',
        (procst, repoid)
    )
    cursorIns.execute('COMMIT')

def makeGitReqForRepo(repoid, url):
    """HTTP-запит з обробкою помилок репозиторію."""
    isError, jsonResp = makeGitReq(url)
    isError = (isError or type(jsonResp) is not list)

    if isError:
        print(f'repoid = {repoid}')
        # Порожній або недоступний репозиторій
        if ('message' in jsonResp
            and any(msg in jsonResp['message']
                    for msg in [
                        'Git Repository is empty',
                        'Not Found',
                        'Repository access blocked',
                    ])):
            setProcst(repoid, ProcStType.EMPTY_REPO)
        else:
            setProcst(repoid, ProcStType.FAIL)

    return isError, jsonResp

def main():
    global connParams, cursorIns

    connFetch = psycopg2.connect(
        dbname=connParams['db'],
        user=connParams['login'],
        password=connParams['pwd'],
        host=connParams['hostip']
    )
    cursorFetch = connFetch.cursor()

    connIns = psycopg2.connect(
        dbname=connParams['db'],
        user=connParams['login'],
        password=connParams['pwd'],
        host=connParams['hostip']
    )
    cursorIns = connIns.cursor()
    connIns.autocommit = False

    # Вибірка необроблених репозиторіїв
    cursorFetch.execute(
        'SELECT repoid, fullname '
        'FROM repodata.repos '
        'WHERE procst = 0 '
        'ORDER BY repoid LIMIT 50000'
    )

    repoRec = cursorFetch.fetchone()
    while repoRec:
        repoid = repoRec[0]
        url = ('https://api.github.com/repos/'
              f'{repoRec[1]}/commits?per_page=100')

```

```

# Пошук останнього SHA для продовження
cursorIns.execute(
    'SELECT shacommith FROM repodata.commits '
    f'WHERE repoid = {repoid} '
    'ORDER BY commitid DESC LIMIT 1'
)
commitRec = cursorIns.fetchone()
shacommith = commitRec[0] if commitRec else ''

reqUrl = (f'{url}&sha={shacommith}'
          if commitRec else url)
isReqError, commits = makeGitReqForRepo(
    repoid, reqUrl
)

while (not isReqError
       and (len(commits) != 1
            or commits[0]['sha'] != shacommith)):
    for commit in commits:
        if commit['sha'] == shacommith:
            continue
        shacommith = commit['sha']

    # Збереження повідомлення
    cursorIns.execute(
        'INSERT INTO repodata.commits '
        '(repoid, shacommith, message, '
        ' cmtdate, instime, updttime) '
        'SELECT %s, %s, %s, %s, '
        ' CURRENT_TIMESTAMP, '
        ' CURRENT_TIMESTAMP '
        'WHERE NOT EXISTS ('
        ' SELECT 1 FROM repodata.commits '
        ' WHERE repoid=%s AND shacommith=%s)',
        (repoid, commit['sha'],
         commit['commit']['message'],
         commit['commit']['committer']['date'],
         repoid, commit['sha'])
    )

    # Збереження батьківських SHA
    parents = commit.get('parents', [])
    for parent in parents:
        cursorIns.execute(
            'INSERT INTO repodata.parents '
            '(commitid, shaparent) '
            'SELECT t.commitid, %s '
            'FROM repodata.commits t '
            'WHERE t.repoid=%s '
            ' AND t.shacommith=%s '
            ' AND NOT EXISTS ('
            ' SELECT 1 '
            ' FROM repodata.parents '
            ' WHERE commitid=t.commitid '
            ' AND shaparent=%s)',
            (parent['sha'], repoid,
             commit['sha'], parent['sha'])
        )

    isReqError, commits = makeGitReqForRepo(
        repoid, f'{url}&sha={shacommith}'
    )

if not isReqError:
    setProcst(repoid, ProcStType.OK)

repoRec = cursorFetch.fetchone()

cursorFetch.close()
connFetch.close()

```

```

        cursorIns.close()
        connIns.close()

main()

```

Лістинг Е.5 -- Сценарій завантаження файлів

```

# -*- coding: utf-8 -*-
"""
Сценарій завантаження файлів для внесених змін
із GitHub REST API у таблицю repodata.files.
"""

from CommonTypes import *
import psycopg2

cursorIns = None

def setProcst(commitid, procst):
    """Оновлення стану обробки повідомлення."""
    global cursorIns
    cursorIns.execute(
        'UPDATE repodata.commits '
        'SET procst = %s, '
        '    updttime = CURRENT_TIMESTAMP '
        'WHERE commitid = %s',
        (procst, commitid)
    )
    cursorIns.execute('COMMIT')

def main():
    global connParams, cursorIns

    connFetch = psycopg2.connect(
        dbname=connParams['db'],
        user=connParams['login'],
        password=connParams['pwd'],
        host=connParams['hostip']
    )
    cursorFetch = connFetch.cursor()

    connIns = psycopg2.connect(
        dbname=connParams['db'],
        user=connParams['login'],
        password=connParams['pwd'],
        host=connParams['hostip']
    )
    cursorIns = connIns.cursor()
    connIns.autocommit = False

    cursorFetch.execute(
        'SELECT c.commitid, r.fullname, c.shacommit '
        'FROM repodata.commits c '
        'JOIN repodata.repos r ON c.repoid = r.repoid '
        'WHERE c.procst = 0 AND r.procst = 1 '
        'ORDER BY commitid LIMIT 1000000'
    )

    commits = cursorFetch.fetchone()
    while commits:
        commitid = commits[0]
        url = ('https://api.github.com/repos/'
              f'{commits[1]}/commits/{commits[2]}')
        try:
            isReqError, commit = makeGitReq(url)
            if (not isReqError
                and 'files' in commit

```

```

        and type(commit['files']) is list):
    for file in commit['files']:
        diff = (file['patch']
                if 'patch' in file
                else None)
    cursorIns.execute(
        'INSERT INTO repodata.files '
        '(commitid, filename, status, '
        ' additions, deletions, rawurl, '
        ' diff, instime, updttime) '
        'SELECT %s,%s,%s,%s,%s,%s,%s, '
        ' CURRENT_TIMESTAMP, '
        ' CURRENT_TIMESTAMP '
        'WHERE NOT EXISTS ( '
        ' SELECT 1 FROM repodata.files '
        ' WHERE commitid=%s '
        ' AND filename=%s)',
        (commitid, file['filename'],
         file['status'],
         file['additions'],
         file['deletions'],
         file['raw_url'], diff,
         commitid, file['filename'])
    )
    setProcst(commitid, ProcStType.OK)

except Exception as e:
    print(f'{e}. commitid={commitid}')
    setProcst(commitid, ProcStType.FAIL)

commits = cursorFetch.fetchone()

cursorFetch.close()
connFetch.close()
cursorIns.close()
connIns.close()

main()

```

Лістинг Е.6 -- Сценарій завантаження тексту файлів

```

# -*- coding: utf-8 -*-
"""
Сценарій завантаження тексту файлів
із GitHub REST API у таблицю repodata.filetext.
"""

from CommonTypes import *
import psycopg2

def main():
    global connParams

    connFetch = psycopg2.connect(
        dbname=connParams['db'],
        user=connParams['login'],
        password=connParams['pwd'],
        host=connParams['hostip']
    )
    cursorFetch = connFetch.cursor()

    global cursorIns
    connIns = psycopg2.connect(
        dbname=connParams['db'],
        user=connParams['login'],
        password=connParams['pwd'],
        host=connParams['hostip']
    )

```

```

cursorIns = connIns.cursor()
connIns.autocommit = False

cursorFetch.execute(
    'SELECT fileid, rawurl '
    'FROM repodata.files '
    'WHERE procst = 0 '
    'ORDER BY fileid LIMIT 1000000'
)

files = cursorFetch.fetchone()
while files:
    fileid = files[0]
    url = files[1]

    isReqError, fileTxt = makeGitReq(url, False)
    if not isReqError:
        fileTxt = fileTxt.replace('\x00', '')
        cursorIns.execute(
            'INSERT INTO repodata.filetext '
            '(fileid, ftext) '
            'SELECT %s, %s '
            'WHERE NOT EXISTS ('
            ' SELECT 1 FROM repodata.filetext '
            ' WHERE fileid = %s)',
            (fileid, fileTxt, fileid)
        )
        cursorIns.execute(
            'UPDATE repodata.files '
            'SET procst = %s, '
            '    uptime = CURRENT_TIMESTAMP '
            'WHERE fileid = %s',
            (ProcStType.OK, fileid)
        )
        cursorIns.execute('COMMIT')

    files = cursorFetch.fetchone()

cursorFetch.close()
connFetch.close()
cursorIns.close()
connIns.close()

main()

```

ДОДАТОК Ж. МОБІЛЬНИЙ ЗАСТОСУНОК CMTSELECTOR ДЛЯ РОЗМІТКИ НАВЧАЛЬНОЇ ВИБІРКИ

Лістинг Ж.1 -- MainActivity.kt (Kotlin)

```
package com.example.cmtselector

import android.content.Context
import android.graphics.Color
import android.os.Bundle
import android.widget.Button
import android.widget.EditText
import android.widget.TextView
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import com.android.volley.Request
import com.android.volley.toolbox.JsonArrayRequest
import com.android.volley.toolbox.StringRequest
import com.android.volley.toolbox.Volley
import com.jaredrummler.android.device.DeviceName
import org.json.JSONObject
import java.text.SimpleDateFormat
import java.util.Calendar

val NOT_FOUND_MSG: String = "Не знайдено"

enum class ProcStEnum(val procSt: Int) {
    PROCESSED(1),
    OK_MSG(10),
    BAD_MSG(11),
    NOT_SURE_BUT_OK(12),
    NOT_SURE_BUT_BAD(13);

    companion object {
        fun fromInt(procSt: Int) = ProcStEnum.values().first { it.procSt == procSt }
    }
}

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val ipEditTxt = findViewById<EditText>(R.id.ipEditText)
        val cmtMsgTxtMultiLine = findViewById<TextView>(R.id.cmtMsgTextMultiLine)
        val prevCmtMsgBtn = findViewById<Button>(R.id.prevCmtMsgButton)
        val nextCmtMsgBtn = findViewById<Button>(R.id.nextCmtMsgButton)
        val okCmtMsgBtn = findViewById<Button>(R.id.okCmtMsgButton)
        val badCmtMsgBtn = findViewById<Button>(R.id.badCmtMsgButton)
        val nSureOkCmtMsgBtn = findViewById<Button>(R.id.nSureOkCmtMsgButton)
        val nSureBadCmtMsgBtn = findViewById<Button>(R.id.nSureBadCmtMsgButton)

        val sharedPreferences = this.getSharedPreferences(Context.MODE_PRIVATE)
        var ipAddress = sharedPreferences.getString("ipAddress", ipEditTxt.text.toString())
        ipEditTxt.setText(ipAddress)

        val volleyQueue = Volley.newRequestQueue(this)
        var commitId: Int? = null

        fun initCmtMsgText() {
            val jsonReq = JsonArrayRequest(
                Request.Method.GET,
                "http://${ipEditTxt.text}/commits?procst=eq.1&order=commitid.asc&limit=1",
                null,
                { response ->
                    if (response.optJSONObject(0) != null &&

```

```

        response.getJSONObject(0).has("commitid")) {
            commitId = response.getJSONObject(0).get("commitid") as Int
            cmtMsgTxtMultiLine.text
        } else {
            Toast.makeText(this, NOT_FOUND_MSG, Toast.LENGTH_LONG).show()
        }
    },
    { error ->
        Toast.makeText(this, error.localizedMessage, Toast.LENGTH_LONG).show()
    }
)
volleyQueue.add(jsonReq)
}

fun getNextCmtMsg(isNext: Boolean) {
    val jsonReq = JsonRequest(
        Request.Method.GET,
        "http://${ipEditTxt.text}/commits?commitid=${if (isNext) "gt" else "lt"}." +
            "${commitId}&order=commitid.${if (isNext) "asc" else
"desc"}&limit=1",
        null,
        { response ->
            if (response.optJSONObject(0) != null &&
response.getJSONObject(0).has("commitid")) {
                commitId = response.getJSONObject(0).get("commitid") as Int
                cmtMsgTxtMultiLine.text
            } else {
                Toast.makeText(this, NOT_FOUND_MSG, Toast.LENGTH_LONG).show()
            }
        },
        { error ->
            Toast.makeText(this, error.localizedMessage, Toast.LENGTH_LONG).show()
        }
    )
    volleyQueue.add(jsonReq)
}

fun setCmtMsgText(isNext: Boolean) {
    if (commitId == null) {
        initCmtMsgText()
    } else {
        getNextCmtMsg(isNext)
    }
}

fun updateProcSt(procSt: Int) {
    DeviceName.init(this)
    val deviceName = DeviceName.getDeviceName()

    val stringRequest = object: StringRequest(

```

```

        Request.Method.PATCH,
        "http://${ipEditTxt.text}/commits?commitid=eq.${commitId}",
        null,
        { error ->
            Toast.makeText(this, error.toString(), Toast.LENGTH_LONG).show()
        }
    )
    {
        override fun getHeaders(): MutableMap<String, String> {
            val headers = HashMap<String, String>()
            headers["Authorization"] = "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJyb2xlIjoiaG9kb191c2VyIn0.2xdouu9BzGxe-NLMFm20s9kNaT4e_M5kDLjZ4MPHpp8"
            headers["Content-Type"] = "application/json"
            return headers
        }

        override fun getBody(): ByteArray {
            val jsonBody = JSONObject()
            jsonBody.put("procst", procSt)
            jsonBody.put("username", deviceName)

            val currDt = Calendar.getInstance().time
            val formatter = SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
            jsonBody.put("uptime", formatter.format(currDt))
            return jsonBody.toString().encodeToByteArray()
        }
    }
    volleyQueue.add(stringRequest)
}

prevCmtMsgBtn.setOnClickListener() {
    setCmtMsgText(false)
}

nextCmtMsgBtn.setOnClickListener() {
    setCmtMsgText(true)
}

okCmtMsgBtn.setOnClickListener() {
    updateProcSt(ProcStEnum.OK_MSG.procSt)
    setCmtMsgText(true)
}

badCmtMsgBtn.setOnClickListener() {
    updateProcSt(ProcStEnum.BAD_MSG.procSt)
    setCmtMsgText(true)
}

nSureOkCmtMsgBtn.setOnClickListener() {
    updateProcSt(ProcStEnum.NOT_SURE_BUT_OK.procSt)
    setCmtMsgText(true)
}

nSureBadCmtMsgBtn.setOnClickListener() {
    updateProcSt(ProcStEnum.NOT_SURE_BUT_BAD.procSt)
    setCmtMsgText(true)
}
}

override fun onDestroy() {
    super.onDestroy()

    val editor = getPreferences(Context.MODE_PRIVATE).edit()
    editor.putString("ipAddress",
        findViewById<EditText>(R.id.ipEditText).text.toString())
    editor.commit()
}
}

```

Лістинг Ж.2 -- activity_main.xml (Android Layout)

```

<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".MainActivity">

    <Button
        android:id="@+id/okCmtMsgButton"
        android:layout_width="160dp"
        android:layout_height="90dp"
        android:layout_marginTop="24dp"
        android:layout_marginEnd="16dp"
        android:backgroundTint="#009688"
        android:text="Повідомлення відповідає вимогам"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/prevCmtMsgButton"
        tools:ignore="MissingConstraints" />

    <Button
        android:id="@+id/badCmtMsgButton"
        android:layout_width="160dp"
        android:layout_height="90dp"
        android:layout_marginStart="16dp"
        android:layout_marginTop="24dp"
        android:backgroundTint="#F44336"
        android:text="Повідомлення НЕ відповідає вимогам"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/prevCmtMsgButton"
        tools:ignore="MissingConstraints" />

    <Button
        android:id="@+id/nSureOkCmtMsgButton"
        android:layout_width="160dp"
        android:layout_height="90dp"
        android:layout_marginTop="12dp"
        android:layout_marginEnd="16dp"
        android:backgroundTint="#8BC34A"
        android:text="Не впевнений, але беремо для навчання"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/okCmtMsgButton"
        tools:ignore="MissingConstraints" />

    <Button
        android:id="@+id/nSureBadCmtMsgButton"
        android:layout_width="160dp"
        android:layout_height="90dp"
        android:layout_marginStart="16dp"
        android:layout_marginTop="12dp"
        android:backgroundTint="#F5F386"
        android:text="Не впевнений, і не беремо для навчання"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/badCmtMsgButton"
        tools:ignore="MissingConstraints" />

    <Button
        android:id="@+id/prevCmtMsgButton"
        android:layout_width="160dp"
        android:layout_height="55dp"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:text="Попереднє повідомлення"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/cmtMsgTextMultiLine" />

```

```

<EditText
    android:id="@+id/cmtMsgTextMultiLine"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="19dp"
    android:ems="10"
    android:focusableInTouchMode="false"
    android:gravity="start|top"
    android:inputType="textMultiLine"
    android:lines="5"
    android:scrollbars="vertical"
    android:text="Тут відображатимуться повідомлення про внесені зміни"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/ipEditText" />

<Button
    android:id="@+id/nextCmtMsgButton"
    android:layout_width="160dp"
    android:layout_height="55dp"
    android:layout_marginTop="16dp"
    android:layout_marginEnd="16dp"
    android:text="Наступне повідомлення"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/cmtMsgTextMultiLine" />

<EditText
    android:id="@+id/ipEditText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="19dp"
    android:ems="10"
    android:inputType="text"
    android:minHeight="48dp"
    android:text="185.102.187.79:3000"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>

```

ДОДАТОК 3. СЕРТИФІКАТ УЧАСТІ В МІЖНАРОДНІЙ
КОНФЕРЕНЦІЇ COIA-2024 (М. СТАМБУЛ, ТУРЕЧЧИНА, 27–29
СЕРПНЯ 2024 Р.)



**ДОДАТОК И. СЕРТИФІКАТ УЧАСТІ В МІЖНАРОДНІЙ
КОНФЕРЕНЦІЇ ITS-2025 (М. КИЇВ, УКРАЇНА, 11 ГРУДНЯ 2025 Р.)**



CERTIFICATE OF PARTICIPATION

The Organizing Committee of the XXV International Scientific and Practical Conference

"Information Technologies and Security" (ITS-2025) certifies that:

Sergiy Pogorilyy and Bohdan Semonov

have participated in the ITS-2025 Conference, Kyiv, Ukraine, December 11, 2025, with the paper titled:

*LANGUAGE-SPECIFIC FINE-TUNING OF TRANSFORMER MODELS FOR
COMMIT MESSAGE GENERATION IN VERSION CONTROL SYSTEMS*

Conference Chairman

Deputy Director of the Institute for Information Recording of the National Academy of Sciences of Ukraine

Honored Worker of Science and Engineering of Ukraine, Dr.Eng., Professor

Oleksandr Dodonov



«ЗАТВЕРДЖУЮ»



Декан факультету комп'ютерних
наук та кібернетики Київського
національного університету імені
Тараса Шевченка
_____ доктор фізико-
математичних наук КАШПУР О. Ф.
«24» червня 2026 р.

АКТ

впровадження в навчальний процес результатів дисертаційної роботи Семьонова Богдана Олександровича «Проектування і застосування нейронних мереж для створення методів оцінювання якості та генерації повідомлень про внесені зміни в системах контролю версій», поданої на здобуття наукового ступеня доктора філософії за спеціальністю 123 – Комп'ютерна інженерія

Цей акт засвідчує, що наступні результати дисертаційної роботи Семьонова Богдана Олександровича:

1. Метод оцінювання якості повідомлень про внесені зміни у системах контролю версій на основі рекурентних та трансформерних нейронних мереж (розділи 2 та 3 роботи);
2. Метод автоматичної генерації повідомлень про внесені зміни на основі моделей архітектури кодувальник-декодувальник із мовно-специфічним донавчанням та семантичним збагаченням ідентифікаторів вихідного коду (розділ 4 роботи)

використано в курсі «Актуальні проблеми обробки інформації в комп'ютерних системах» освітньої програми «Інформатика» для студентів освітнього рівня «Магістр» факультету комп'ютерних наук та кібернетики спеціальності 122 «Комп'ютерні науки» Київського національного університету імені Тараса Шевченка.

Завідувач кафедри
математичної інформатики
доктор фіз.-мат. наук, професор

ТЕРЕЩЕНКО В. М.

№ KHO-12.3/28 від 30.06.2026

ДОВІДКА

Даним листом АТ «ПУМБ» підтверджує впровадження результатів дисертаційної роботи Семьонова Богдана Олександровича «Проектування і застосування нейронних мереж для створення методів оцінювання якості та генерації повідомлень про внесені зміни в системах контролю версій», поданої на здобуття наукового ступеня доктора філософії за спеціальністю 123 – Комп'ютерна інженерія

Результати дисертаційної роботи Семьонова Богдана Олександровича, а саме:

1. Метод оцінювання якості повідомлень про внесені зміни (commit messages) у системах контролю версій на основі трансформерних нейронних мереж;
2. Метод автоматичної генерації повідомлень про внесені зміни на основі моделей архітектури кодувальник-декодувальник із семантичним збагаченням ідентифікаторів вихідного коду через автоматичне розкриття аббревіатур,

впроваджено у процеси розробки програмного забезпечення АТ «ПУМБ» у вигляді програмного засобу контролю якості повідомлень про внесені зміни, інтегрованого до системи контролю версій Git.

Директор Департаменту інформаційних технологій
АТ «ПУМБ»

Безруков А. Б.

