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

БІЛЕЦЬКИЙ ПАВЛО ВОЛОДИМИРОВИЧ

УДК 004.8

ДИСЕРТАЦІЯ

СТВОРЕННЯ МЕТОДІВ ПОШУКУ КОРЕФЕРЕНТНИХ ОБ'ЄКТІВ В УКРАЇНОМОВНИХ ТЕКСТАХ НА ОСНОВІ ДЕРЕВ РІШЕНЬ, НЕЙРОННИХ МЕРЕЖ ТА ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ

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

________________ Білецький Павло Володимирович

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

Київ - 2025

АНОТАЦІЯ

Білецький П.В. Створення методів пошуку кореферентних об'єктів в україномовних текстах на основі дерев рішень, нейронних мереж та великих мовних моделей. – 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, 2025.

В роботі досліджуються, проектуються та створюються методи автоматизованого пошуку кореферентних об’єктів в україномовних текстах з використанням дерев рішень, нейронних мереж та великих мовних моделей.

Пошук кореферентних об’єктів є важливою задачею обробки природної мови (NLP), що полягає в ідентифікації всіх виразів (слів або словосполучень), які посилаються на один і той же об’єкт у тексті. Вирішення цієї задачі критично важливе для розуміння тексту, виділення інформації, машинного перекладу, оцінки когерентності та розв'язування інших задач обробки природної мови. Особливістю української мови є вільний порядок слів, що ускладнює задачу порівняно з мовами з фіксованим порядком та вимагає методів, здатних враховувати глибокі семантичні та контекстуальні зв’язки. Розробка ефективних методів для української мови є актуальною через обмеженість існуючих ресурсів та досліджень у цій галузі, особливо з огляду на стрімкий розвиток технологій штучного інтелекту.

Актуальність роботи зумовлена необхідністю створення спеціалізованих інструментів для аналізу україномовних текстів, які б враховували лінгвістичні особливості мови та могли б ефективно використовувати сучасні досягнення в галузі машинного навчання. Робота досліджує застосування різноманітних підходів для вирішення поставленої задачі. Розглядаються як класичні методи на основі дерев рішень, що дозволяють створювати інтерпретовані моделі з використанням лінгвістичних ознак, так і сучасні нейромережеві архітектури (зокрема,

згортково-рекурентні мережі), здатні автоматично виявляти складні закономірності. Особливу увагу приділено застосуванню великих мовних моделей (LLM) на базі архітектури Transformer, які демонструють передові результати в галузі NLP завдяки здатності обробляти великі контексти та використовувати знання, отримані під час попереднього навчання на великих наборах даних.

У роботі здійснено аналіз існуючих методів пошуку кореферентних об'єктів, включаючи підходи на основі правил, дерев рішень, різних нейромережевих архітектур та сучасних мовних моделей. Визначено їх переваги та недоліки, зокрема для застосування до україномовних текстів. На основі аналізу сформульовано мету та завдання дослідження, що полягають у створенні та дослідженні ефективності нових методів пошуку кореферентних об'єктів в україномовних текстах з використанням дерев рішень, згортково-рекурентних нейронних мереж та великих мовних моделей, а також адаптації сучасних LLM для цієї задачі. Далі приведені **основні результати та наукова новизна.**

Розроблено метод пошуку кореферентних об'єктів на основі дерев рішень, що використовує набір лінгвістичних та семантичних ознак (включаючи векторні представлення ELMo та морфологічні характеристики). Показано ефективність зведення задачі до бінарної класифікації пар потенційно кореферентних об'єктів та досягнуто високих показників точності, зокрема за метрикою B-cubed. Забезпечено інтерпретованість моделі шляхом можливості візуалізації дерева рішень. Вперше проведено формальну верифікацію властивостей розробленої моделі на основі дерев рішень з використанням розмічених транзиційних систем та автоматів Бюхі для доведення її семантичної коректності та підвищення надійності.

Запропоновано метод пошуку кореферентних об'єктів на основі згортково-рекурентної нейронної мережі з довгою та короткочасною пам'яттю (ConvLSTM). Досліджено ефективність поєднання згорткових

шарів для виділення локальних ознак із векторних представлень слів та рекурентних шарів LSTM для врахування контекстуальних залежностей при класифікації кореферентних пар в українських текстах.

Вперше розроблено та системно досліджено два методи використання великих мовних моделей (LLM) для пошуку кореферентних об'єктів в україномовних текстах:

- на основі бінарної класифікації пар об'єктів за допомогою спеціально сформованого запиту (prompt);
- на основі генерації повного кластеру кореферентних об'єктів безпосередньо моделлю у відповідь на запит. Проведено порівняльний аналіз ефективності низки сучасних LLM (Llama 3, Llama 3.1, Llama 3.2, Llama 3.3, DeepSeek R1, Gemma 3, Gemini 2.0 Flash, Gemini 2.0 Thinking) для цієї задачі. Встановлено, що метод генерації кластерів є значно ефективнішим за кількістю необхідних звернень до LLM. Виявлено позитивний вплив здатності моделей до міркування (на прикладі Gemini 2.0 Thinking) на якість розв'язання задачі.

Вперше здійснено донавчання (fine-tuning) відкритої великої мовної моделі Llama 3.2 (3 мільярди параметрів) з використанням техніки QLoRA спеціально для задачі пошуку кореферентних об’єктів в україномовних текстах на основі методу генерації кластерів. Донавчена модель демонструє суттєве покращення якості порівняно з базовою моделлю і досягає результатів, співставних з потужними пропрієтарними моделями (наприклад, Gemini 2.0 Flash), підтверджуючи високу ефективність та доцільність адаптації відкритих LLM для специфічних завдань обробки української мови.

Досліджено ефективність використання високопродуктивних обчислень для прискорення роботи методів на основі LLM. Експериментально підтверджено значний приріст швидкодії (у ~6.7-6.8 разів) при використанні графічних прискорювачів (GPU) порівняно з центральними

процесорами (CPU). Показано переваги квантизації (зокрема, 4-бітної порівняно з 8-бітною, прискорення ~47%) для зменшення вимог до ресурсів та подальшого прискорення обчислень, що є критичним для практичного застосування LLM.

Ключові слова: обробка природної мови, пошук кореферентних об’єктів, українська мова, дерева рішень, нейронні мережі, згортково-рекурентна нейронна мережа, LSTM, великі мовні моделі, Transformer, RoBERTa, Llama, Gemma, Gemini, донавчання моделі, формальна верифікація, високопродуктивні обчислення, GPU, квантизація, кластеризація.

SUMMARY

Biletskyi P.V. Creation of methods for coreference resolution in Ukrainian-language texts based on decision trees, neural networks, and large language models. – Qualification scientific work on the rights of the manuscript.

The PhD thesis on competition of a scientific degree of the doctor of philosophy on a specialty 123 “Computer Engineering”. – Taras Shevchenko National University of Kyiv, Kyiv, 2025.

In the work, methods for automated coreference resolution in Ukrainian-language texts based on decision trees, neural networks, and large language models are investigated, designed and created.

Coreference resolution is an important natural language processing (NLP) task, consisting in identifying all expressions (words or phrases) that refer to the same entity in a text. Solving this task is critically important for text understanding, information extraction, machine translation, coherence evaluation, and other NLP applications. A specific feature of the Ukrainian language is its free word order, which complicates the task compared to languages with fixed order and requires methods capable of capturing deep semantic and contextual links. The development of effective methods for Ukrainian is relevant due to the limited existing resources and research in this area, especially considering the rapid development of artificial intelligence technologies.

The relevance of the work stems from the need to create specialized tools for analyzing Ukrainian-language texts that would consider the linguistic specifics of the language and could effectively leverage modern achievements in machine learning. The paper investigates the application of various approaches to solve the posed task. Both classic methods based on decision trees, which allow creating interpretable models using linguistic features, and modern neural network architectures (specifically, convolutional-recurrent networks), capable of automatically detecting complex patterns, are considered. Particular attention is given to the application of large language models (LLMs) based on the

Transformer architecture, which demonstrate state-of-the-art results in NLP due to their ability to process large contexts and utilize knowledge gained during pre-training on massive datasets.

In the paper, an analysis of existing coreference resolution methods has been performed, including approaches based on rules, decision trees, various neural network architectures, and modern large language models. Their advantages and disadvantages, particularly for application to Ukrainian-language texts, have been identified. Based on this analysis, the purpose and tasks of the research were formulated, consisting in the creation and investigation of the effectiveness of new methods for coreference resolution in Ukrainian-language texts using decision trees, convolutional-recurrent neural networks, and large language models, as well as the adaptation of modern LLMs for this task. **The main results and scientific novelty** are listed below.

A method for coreference resolution based on decision trees has been developed, utilizing a set of linguistic and semantic features (including ELMo vector representations and morphological characteristics). The effectiveness of reducing the task to binary classification of potentially coreferent pairs has been shown, and high accuracy metrics, particularly B-cubed, have been achieved. Interpretability of the model is ensured through the possibility of visualizing the decision tree. For the first time, formal verification of the properties of the developed decision tree-based model was performed using labeled transition systems and Büchi automata to prove its semantic correctness and enhance reliability.

A method for coreference resolution based on a convolutional-recurrent neural network with long short-term memory (ConvLSTM) has been proposed. The effectiveness of combining convolutional layers for extracting local features from word vector representations and recurrent LSTM layers for considering contextual dependencies in classifying coreferent pairs in Ukrainian texts has been investigated.

For the first time, two methods for using large language models (LLMs) for coreference resolution in Ukrainian-language texts have been developed and systematically investigated:

- based on binary classification of object pairs using a specially crafted prompt;
- based on the generation of the complete coreferent cluster directly by the model in response to a prompt. A comparative analysis of the effectiveness of several modern LLMs (Llama 3, Llama 3.1, Llama 3.2, Llama 3.3, DeepSeek R1, Gemma 3, Gemini 2.0 Flash, Gemini 2.0 Thinking) for this task was conducted. It was established that the cluster generation method is significantly more efficient in terms of the number of required calls to the LLM. A positive impact of the models' reasoning capabilities (exemplified by Gemini 2.0 Thinking) on the quality of task resolution was identified.

For the first time, fine-tuning of the open large language model Llama 3.2 (3 billion parameters) was successfully performed using the QLoRA technique specifically for the task of coreference resolution in Ukrainian-language texts, based on the cluster generation method. The fine-tuned model demonstrates a significant improvement in quality compared to the baseline "zero-shot" model and achieves results comparable to powerful proprietary models (e.g., Gemini 2.0 Flash), confirming the high effectiveness and feasibility of adapting open LLMs for specific Ukrainian language processing tasks.

The effectiveness of using high-performance computing (HPC) to accelerate the operation of methods based on LLMs has been investigated. A significant performance increase (by ~6.7-6.8 times) when using graphics processing units (GPUs) compared to central processing units (CPUs) for the LLM inference stage has been experimentally confirmed. The advantages of quantization (reducing the bit representation of model weights, particularly 4-bit compared to 8-bit, speedup ~47%) for reducing memory requirements and further accelerating computations

have also been demonstrated, which is critical for the practical deployment and use of LLMs in natural language processing tasks.

Keywords: natural language processing, coreference resolution, Ukrainian language, decision trees, neural networks, convolutional-recurrent neural network, LSTM, large language models, Transformer, RoBERTa, Llama, Gemma, Gemini, model fine-tuning, QLoRA, formal verification, high-performance computing, GPU, quantization, clustering.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Наукові праці, в яких опубліковані основні наукові результати дисертації:

1. Погорілий, С. Д., & Білецький, П. В. (2021). Використання графічного процесора для прискорення пошуку кореферентних об'єктів з використанням моделі RoBERTa. Наукові праці Донецького національного технічного університету, Серія: Інформатика, кібернетика та обчислювальна техніка, 33(2), (С. 4–9). https://doi.org/10.31474/1996-1588-2021-2-33-4-9.
2. Погорілий, С., & Білецький, П. В. (2022). Алгоритм пошуку кореферентних об’єктів в україномовних текстах із використанням дерев рішень. Problems in programming, (3-4), (С. 85–91). https://doi.org/10.15407/pp2022.03-04.085.
3. Погорілий, С. Д., Крамов, А. А., & Білецький, П. В. (2019). Метод оцінки когерентності україномовних текстів з використанням згорткової нейронної мережі. Збірник Наукових Праць Військового Інституту Київського Національного Університету Імені Тараса Шевченка, 65, (С. 63–71). https://doi.org/10.17721/2519-481X/2019/65-08.
4. Погорілий, С. Д., Слинько, М. С., & Білецький, П. В. (2024). Формальна верифікація властивостей моделі пошуку кореферентних об’єктів на основі дерев рішень. Problems in programming, (2-3), (С. 319–328). https://doi.org/10.15407/pp2024.02-03.319.

Наукові праці, які засвідчують апробацію матеріалів дисертації:

5. Pogorilyy, S., & Biletskyi, P. (2022). Coreference resolution based on pretrained RoBERTa language model. In Proceedings of the 8th International Conference on Control and Optimization with Industrial

Applications, COIA 2022, (pp. 354–357). Baku, Azerbaijan. http://coia-conf.org/upload/editor/files/COIA2022_V1_abs.pdf

6. Pogorilyy, S., & Biletskyi, P. (2022). Coreference resolution algorithm for Ukrainian-language texts using decision trees. Proceeding of the UkrProg 2022, (pp. 81-90). https://ceur-ws.org/Vol-3501/s8.pdf
7. Pogorilyy, S., & Biletskyi, P. (2023). Analysis of Decision Trees for Coreference Resolution Task in Ukrainian Language. In Proceedings of the Workshops at the X International Scientific Conference "Information Technology and Implementation" (IT&I-WS 2023) CEUR Workshop Proceedings Vol. 3646, (pp. 255–262). https://ceur-ws.org/Vol-3646/Short_7.pdf
8. Biletskyi, P., & Tkach Y. (2024) Coreference resolution in Ukrainian-language texts using convolutional long short-term memory neural networks. XXIV International young scientists conference on applied physics, (pp. 129-130). https://icap.knu.ua/index.php/icap2024
9. Погорілий, С., & Білецький, П. (2018). Development and analysis of the tool for verification graphic isomorphism. Proceedings of the XVII International Young Scientists' Conference on Applied Physics (pp. 108–109). Kyiv National University. https://indico.knu.ua/event/2/contributions/276/contribution.pdf
10. Pogorilyy, S., Slynko, M., & Biletskyi, P. (2024). The verification of decision tree model for coreference resolution using marked transition systems, Petri nets and Büchi automata. In Proceedings of the 14th International Scientific and Practical Conference on Programming (UkrPROG 2024), CEUR Workshop Proceedings Vol. 3806, (pp. 361–371). https://ceur-ws.org/Vol-3806/S_26_Pogorilyy_Slynko_Biletskyi.pdf
11. Pogorilyy, S., & Biletskyi, P. (2024). Coreference Resolution in Ukrainian-language texts using Llama 3 large language model. In Proceedings of the 9th International Conference on Control and

Optimization with Industrial Applications (COIA 2024), (pp. 538–542). Istanbul, Turkey.
http://coia-conf.org/upload/editor/files/Proc_COIA2024.pdf

ЗМІСТ

АНОТАЦІЯ 2

SUMMARY 6

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА 10

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 18

ВСТУП 19

РОЗДІЛ 1. РОЗГЛЯД МЕТОДІВ АВТОМАТИЗОВАНОГО ПОШУКУ КОРЕФЕРЕНТНИХ ОБ’ЄКТІВ 25

1.1 Визначення поняття кореферентних об’єктів 25

1.2 Особливості пошуку кореферентних об’єктів в україномовних текстах 26

1.3 Корпуси текстів для пошуку кореферентних об’єктів 28

1.4 Підходи до пошуку кореферентних об’єктів 30

1.5 Методи пошуку кореферентних об’єктів 31

1.5.1 Методи на основі наборів правил 31

1.5.2 Метод на основі дерев рішень 32

1.5.3 Методи, що використовують нейронні мережі 34

1.5.4 Методи, засновані на використанні моделей природної мови 37

1.5.5 Порівняння методів пошуку кореферентних об’єктів 39

1.6 Програмні засоби, що використовуються в алгоритмах пошуку кореферентних об’єктів 40

1.7 Використання високопродуктивних обчислень при розв’язанні задачі знаходження кореферентних об’єктів 42

1.8 Висновки з 1 розділу 46

РОЗДІЛ 2. СТВОРЕННЯ МЕТОДІВ ПОШУКУ КОРЕФЕРЕНТНИХ ОБ'ЄКТІВ В УКРАЇНОМОВНИХ ТЕКСТАХ НА ОСНОВІ НЕЙРОННИХ МЕРЕЖ ТА ДЕРЕВ РІШЕНЬ 50

2.1 Метод пошуку кореферентних об'єктів на основі дерев рішень 50

2.1.1 Лінгвістичні характеристики, що використовуються для пошуку кореферентних об'єктів методами на основі дерев рішень 50

2.1.2 Підготовка текстів для аналізу деревом рішень 51

2.1.3 Створення дерева рішень 53

2.1.4 Використання дерева рішень для пошуку кореферентних об'єктів 55

2.2 Формальна верифікація властивостей моделей пошуку кореферентних об'єктів, що використовують дерева рішень 56

2.2.1 Створення розмічених транзиційних моделей на високому рівні абстракції 57

2.2.2 Верифікація властивостей моделі автоматами Бюхі 60

2.3 Метод пошуку кореферентних об'єктів на основі згортково-рекурентних нейронних мереж з довгою та короткочасною пам'яттю 62

2.3.1 Архітектура згортково-рекурентних нейронних мереж з довгою та короткочасною пам'яттю 62

2.3.2 Підготовка текстів для аналізу згортково-рекурентною мережею з довгою та короткочасною пам'яттю 65

2.3.3 Реалізація методу пошуку кореферентних об'єктів на основі згортково-рекурентних нейронних мереж з довгою та короткочасною пам'яттю 66

2.4 Висновки до розділу 2 68

РОЗДІЛ 3. СТВОРЕННЯ МЕТОДІВ ПОШУКУ КОРЕФЕРЕНТНИХ ОБ’ЄКТІВ В УКРАЇНОМОВНИХ ТЕКСТАХ З ВИКОРИСТАННЯМ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ 71

3.1 Архітектура моделей природних мов Transformers 71

3.2 Метод пошуку кореферентних об’єктів з використанням мовної моделі RoBERTa 72

3.3 Розгляд великих мовних моделей, використаних для розробки застосунку для пошуку кореферентних об’єктів. 74

3.3.1 Особливості сучасних моделей природної мови 74

3.3.2 Моделі сімейства Llama (Llama 3, 3.1, 3.2, 3.3) 75

3.3.3 Модель DeepSeek R1 76

3.3.4 Модель Gemma 3 77

3.3.5 Моделі Gemini 2.0 Flash та Gemini 2.0 Thinking 77

3.3.6 Порівняння великих мовних моделей, обраних для розв'язання задачі пошуку кореферентних об'єктів 78

3.4 Розробка методів пошуку кореферентних об’єктів на основі великих мовних моделей 83

3.4.1 Загальна архітектура застосунку пошуку кореферентних об’єктів на основі великих мовних моделей 83

3.4.2 Метод пошуку кореферентних об’єктів з використанням бінарної класифікації на етапі генерації результатів мовною моделлю . 84

3.4.3 Метод створення кластеру кореферентних об’єктів на етапі генерації результатів мовною моделлю 86

3.5 Донавчання великої мовної моделі Llama 3.2 89

3.6 Висновки до розділу 3 91

РОЗДІЛ 4. ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ МЕТОДІВ ОЦІНКИ КОРЕФЕРЕНТНОСТІ В УКРАЇНОМОВНИХ ТЕКСТАХ 94

4.1 Дослідження прискорення роботи моделей пошуку кореферентних об’єктів з використанням графічних прискорювачів......... 94

4.2 Метрики оцінки ефективності алгоритмів 98

4.3 Дослідження використання дерев рішень, нейронних мереж та малих моделей природних мов для пошуку кореферентних об’єктів....... 100

4.4 Дослідження використання великих мовних моделей для пошуку кореферентних об’єктів. 103

4.5 Висновки до розділу 4 106

ВИСНОВКИ.. 109

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 112

ДОДАТОК А. ЛІСТИНГ ПРОГРАМНОГО КОДУ РЕАЛІЗАЦІЇ МЕТРИКИ B-CUBED ТА MUC ДЛЯ ОЦІНКИ ЯКОСТІ ЗНАХОДЖЕННЯ КОРЕФЕРЕНТНИХ ОБ’ЄКТІВ.. 121

ДОДАТОК Б. ЛІСТИНГ ПРОГРАМНОГО КОДУ СТВОРЕННЯ ДЕРЕВА РІШЕНЬ ДЛЯ ПОШУКУ КОРЕФЕРЕНТНИХ ОБ’ЄКТІВ В УКРАЇНОМОВНИХ ТЕКСТАХ... 125

ДОДАТОК В. ЛІСТИНГ ПРОГРАМНОГО КОДУ РЕАЛІЗАЦІЇ МЕТОДУ ПОШУКУ КОРЕФЕРЕНТНИХ ОБ’ЄКТІВ З ВИКОРИСТАННЯМ БІНАРНОЇ КЛАСИФІКАЦІЇ НА ЕТАПІ ГЕНЕРАЦІЇ РЕЗУЛЬТАТІВ МОВНОЮ МОДЕЛЛЮ 148

ДОДАТОК Г. ЛІСТИНГ ПРОГРАМНОГО КОДУ РЕАЛІЗАЦІЇ МЕТОДУ СТВОРЕННЯ КЛАСТЕРУ КОРЕФЕРЕНТНИХ ОБ’ЄКТІВ НА ЕТАПІ ГЕНЕРАЦІЇ РЕЗУЛЬТАТІВ МОВНОЮ МОДЕЛЛЮ...................... 154

ДОДАТОК Д. ЛІСТИНГ ПРОГРАМНОГО КОДУ РЕАЛІЗАЦІЇ ДОНАВЧАННЯ НЕЙРОННОЇ МЕРЕЖІ LLAMA 3.2 161

ДОДАТОК Е. СЕРТИФІКАТ ПРО УЧАСТЬ В МІЖНАРОДНІЙ НАУКОВІЙ КОНФЕРЕНЦІЇ COIA-2022 (БАКУ, АЗЕРБАЙДЖАН, 24-26 СЕРПНЯ 2022).. 170

ДОДАТОК Є. СЕРТИФІКАТ ПРО УЧАСТЬ В МІЖНАРОДНІЙ НАУКОВІЙ КОНФЕРЕНЦІЇ COIA-2024 (СТАМБУЛ, ТУРЕЧЧИНА, 27-29 СЕРПНЯ 2024).. 171

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

BiLSTM	Двостороння довга та короткочасна пам'ять (Bidirectional Long Short-Term Memory)
CNN	Згорткова нейронна мережа (Convolutional Neural Network)
ConvLSTM	Згортково-рекурентна нейронна мережа з довгою та короткочасною пам’яттю (Convolutional-Recurrent Neural Network with Long Short-Term Memory)
CPU	Центральний процесор (Central Processing Unit)
GPU	Графічний процесор (Graphics Processing Unit)
LSTM	Довга та короткочасна пам'ять (Long Short-Term Memory)
MoE	Mixture of Experts (Mixture of Experts)
QLoRA	Техніка QLoRA (Quantized Low-Rank Adaptation)
SMT	Satisfiability Modulo Theories
TOPS	Трильйон операцій за секунду (Trillions of Operations Per Second)
TPU	Тензорний процесор (Tensor Processing Unit)
Transformer	Архітектура Transformer (Transformer architecture)
РНН	Рекурентна нейронна мережа (Recurrent Neural Network)
РТС	Розмічена транзиційна система (Labeled Transition System)

ВСТУП

Обґрунтування вибору теми дослідження. У зв’язку зі стрімким зростанням обсягів текстової інформації в цифровому просторі зростає потреба в її глибокому автоматизованому аналізі. Одним із фундаментальних завдань обробки природної мови (англ. Natural Language Processing, NLP), ключовим для розуміння семантики тексту, є пошук кореферентних об'єктів. Це завдання полягає в ідентифікації всіх згадок (слів або словосполучень) у тексті, які посилаються на одну й ту саму сутність реального чи уявного світу [1, 2, 3]. Успішне вирішення цієї задачі є критично важливим для багатьох прикладних напрямків NLP, таких як виділення інформації, машинний переклад, відповіді на запитання, аналіз тональності, оцінка когерентності тексту та побудова діалогових систем. Знаходження кореферентних об'єктів допомагає в створенні графів знань, об'єднання яких з великими мовними моделями вважають однією зі складових для створення систем штучного інтелекту загального призначення (AGI).

Особливої актуальності задача пошуку кореферентних об'єктів набуває для мов зі складними морфологічними та синтаксичними властивостями, до яких належить українська мова. Вільний порядок слів, характерний для української мови, значно ускладнює ідентифікацію кореферентних зв'язків порівняно з мовами з фіксованим порядком слів (наприклад, англійською). Це вимагає розробки методів, здатних враховувати не лише поверхневі ознаки, а й глибокі семантичні та контекстуальні залежності між елементами тексту. Незважаючи на значний прогрес у галузі NLP, для української мови досі спостерігається обмеженість у доступних ресурсах (корпусах даних) та спеціалізованих методах для вирішення задачі пошуку кореферентних об'єктів.

Розвиток технологій штучного інтелекту (ШІ), зокрема методів машинного навчання, відкриває нові можливості для вирішення складних лінгвістичних задач. Історично для пошуку кореферентних об'єктів

використовувалися методи на основі правил [4], проте вони вимагають значних зусиль експертів-лінгвістів і погано адаптуються до різних контекстів. Згодом поширення набули методи на основі класичних алгоритмів машинного навчання, таких як дерева рішень [5], що дозволяють створювати інтерпретовані моделі на основі лінгвістичних ознак. Подальший розвиток пов'язаний із застосуванням нейронних мереж різних архітектур (рекурентних [2, 6, 7], згорткових [3]), які здатні автоматично виділяти складні патерни з даних.

Останнім часом революційний вплив на NLP справили великі мовні моделі (англ. Large Language Models, LLM) на базі архітектури Transformer [8]. Ці моделі, навчені на величезних масивах текстових даних, демонструють вражаючі результати у широкому спектрі завдань, включаючи пошук кореферентних об'єктів, завдяки своїй здатності обробляти довгі контексти та використовувати попередньо засвоєні знання. Однак застосування LLM для української мови, особливо для такої специфічної задачі, як пошук кореферентних об'єктів, потребує системного дослідження ефективності різних підходів та порівняння різних моделей.

Актуальність даного дослідження зумовлена необхідністю створення та порівняльного аналізу ефективності різноманітних методів пошуку кореферентних об'єктів саме для україномовних текстів. Існує потреба в дослідженні як класичних, інтерпретованих підходів (дерева рішень), так і сучасних нейромережевих архітектур та передових великих мовних моделей. Важливим аспектом є не лише досягнення високої точності, але й забезпечення надійності (через формальну верифікацію) та ефективності (через використання високопродуктивних обчислень та програмних технік покращення продуктивності, таких як квантизація). Недостатня дослідженість цих питань для української мови формує наукову проблему, вирішенню якої присвячена дана робота.

Мета і завдання дослідження. Метою дисертаційного дослідження є створення та дослідження методів автоматизованого пошуку кореферентних об'єктів в україномовних текстах на основі дерев рішень, нейронних мереж та великих мовних моделей, включаючи їх адаптацію до української мови та оцінку ефективності роботи.

Для досягнення поставленої мети сформовано наступні завдання:

Розробити метод пошуку кореферентних об'єктів в україномовних текстах на основі дерев рішень з використанням лінгвістичних ознак та векторних представлень слів; виконати формальну верифікацію властивостей розробленої моделі.

Розробити метод пошуку кореферентних об'єктів на основі згортково-рекурентної нейронної мережі з довгою та короткочасною пам’яттю (ConvLSTM) та дослідити його ефективність для української мови.

Розробити та системно дослідити методи використання великих мовних моделей (LLM) для пошуку кореферентних об'єктів в україномовних текстах, порівнявши підходи бінарної класифікації пар та генерації кластерів, а також ефективність різних сучасних LLM.

Здійснити донавчання (fine-tuning) відкритої великої мовної моделі для задачі пошуку кореферентних об’єктів в україномовних текстах та оцінити приріст якості.

Дослідити ефективність використання високопродуктивних обчислень (графічних прискорювачів) та технік квантизації для прискорення роботи методів пошуку кореферентних об'єктів на основі великих мовних моделей.

Об’єктом дослідження є методи автоматизованого пошуку кореферентних об'єктів у текстах природної мови на основі підходів машинного навчання.

Предметом дослідження є процес проектування, реалізації та порівняльного аналізу моделей на основі дерев рішень, згортково-рекурентних нейронних мереж та великих мовних моделей для пошуку кореферентних об'єктів в україномовних текстах; адаптація та донавчання великих мовних моделей для української мови; оцінка впливу архітектурних рішень, методів взаємодії з моделями та обчислювальних ресурсів на ефективність розв'язання задачі.

Методи дослідження. У дисертаційній роботі застосовуються методи машинного навчання, дерева рішень, нейронні мережі, великі мовні моделі (LLM), методи обробки природної мови, методи формальної верифікації алгоритмів, методи високопродуктивних обчислень, методи кластеризації. Для програмної реалізації методів та відповідних моделей використовується мова програмування Python та спеціалізовані бібліотеки.

Наукова новизна отриманих результатів. Впродовж розв’язання сформованих завдань дослідження вперше отримано наступні результати:

1. Створено метод пошуку кореферентних об'єктів в україномовних текстах на основі дерев рішень з використанням комбінації лінгвістичних ознак та векторних представлень слів; вперше виконано формальну верифікацію властивостей такої моделі за допомогою розмічених транзиційних систем та автоматів Бюхі для доведення її семантичної коректності.
2. Створено метод пошуку кореферентних об'єктів на основі згортково-рекурентної нейронної мережі з довгою та короткочасною пам'яттю (ConvLSTM) та досліджено його ефективність для обробки україномовних текстів.
3. Вперше розроблено та системно досліджено два методи використання великих мовних моделей (LLM) для пошуку кореферентних об'єктів в україномовних текстах (бінарна класифікація пар та генерація кластерів); проведено порівняльний

аналіз ефективності низки сучасних великих мовних моделей для цієї задачі та визначено переваги методу генерації кластерів.

4. Вперше здійснено донавчання (fine-tuning) відкритої великої мовної моделі Llama 3.2 (3 мільярди параметрів) спеціально для задачі пошуку кореферентних об’єктів в україномовних текстах на основі методу генерації кластерів, що дозволило досягти результатів, співставних з потужними пропрієтарними моделями.
5. Досліджено ефективність використання високопродуктивних обчислень, зокрема графічних процесорів (GPU) та квантизації для прискорення роботи методів пошуку кореферентних об'єктів на основі LLM в контексті української мови, кількісно оцінено приріст швидкодії.

Особистий внесок здобувача. Дисертаційна робота є результатом самостійних розробок автора. В роботах, виконаних у співавторстві, автору належать наступні результати: [9] – дослідження використання моделі RoBERTa для пошуку кореферентних об’єктів; [10] – дослідження використання GPU для прискорення пошуку кореферентних об’єктів з моделлю RoBERTa; [11] – розробка інструменту верифікації ізоморфізму графів; [12, 13, 14] – розробка, дослідження та детальний аналіз алгоритму на основі дерев рішень для пошуку кореферентних об’єктів в україномовних текстах; [15, 16] – вибір характеристик та робота над верифікацією моделі на основі дерев рішень; [17] – дослідження ефективності згорткової нейронної мережі для оцінки когерентності; [18] – формулювання методу пошуку кореферентних об’єктів з використанням згортково-рекурентної нейронної мережі з довгою та короткочасною пам’яттю (ConvLSTM); [19] – дослідження використання Llama 3 для пошуку кореферентних об’єктів в україномовних текстах.

Апробація матеріалів дисертації. Основні теоретичні та практичні результати дисертаційної роботи доповідались та обговорювались на міжнародних наукових конференціях та семінарах:

1. XVII International Young Scientists' Conference on Applied Physics (м. Київ, 2018 р.).
2. The 8th International Conference on Control and Optimization with Industrial Applications, COIA 2022 (м. Баку, Азербайджан, 24-26 серпня 2022 р.).
3. XIII-та міжнародна науково-практична конференція з програмування УкрПРОГ'2022 (м. Київ, 2022 р.).
4. X International Scientific Conference "Information Technology and Implementation" (IT&I-WS 2023) (м. Київ, 2023 р.).
5. XXIV International young scientists conference on applied physics (м. Київ, 2024 р.).
6. XIV-та міжнародна науково-практична конференція з програмування УкрПРОГ'2024 (м. Київ, 2024 р.).
7. The 9th International Conference on Control and Optimization with Industrial Applications, COIA 2024 (м. Стамбул, Туреччина, 27-29 серпня 2024 р.).

Публікації. За темою дисертаційної роботи опубліковано 11 наукових праць: 4 статті у фахових наукових виданнях України та 7 тез доповідей на міжнародних конференціях. Тези 3 конференцій внесено в наукометричні бази даних Scopus та DBLP.

Структура та обсяг дисертаційної роботи. Дисертаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг роботи складає 171 сторінку, з яких основний зміст викладено на 111 сторінках. Дисертація містить 15 рисунків, 10 таблиць та список використаних джерел із 84 найменувань.

РОЗДІЛ 1. РОЗГЛЯД МЕТОДІВ АВТОМАТИЗОВАНОГО ПОШУКУ КОРЕФЕРЕНТНИХ ОБ’ЄКТІВ

1.1Визначення поняття кореферентних об’єктів

В лінгвістиці під кореферентними розуміються об’єкти (слова або словосполучення), що вказують на один і той же об’єкт в тексті [1, 2, 3, 12]. Іменник, що позначає об’єкт (референт), має назву антецедента чи постцедента в залежності від його розміщення відносно займенника. Іменник може стояти перед займенником, що вказує на об’єкт – у такому випадку іменник називається антецедентом, а займенник - анафорою. Якщо іменник стоїть після займенника, іменник називають постцедентом, а займенник - катафорою. Інколи іменник чи займенник є складеними - при цьому потрібно враховувати значення декількох слів. Також, референт може включати одразу декілька іменників, при цьому займенник буде вказувати на цю множину об’єктів. Окрім того, інколи декілька займенників вказують на один іменник, що також може бути складеним.

Пошук кореферентних об’єктів входить у сферу обробки природних мов (natural language processing, NLP). Пошук кореферентних об’єктів дозволяє знаходити зв’язки в тексті, виділяти інформацію з текстів із складною структурою (відстежувати зв’язок між дійовими особами та їхніми діями та описом), допомагає в оцінці когерентності (зв’язності) тексту. Також такий пошук уможливлює правильний переклад між мовами, оскільки зіставлення кореферентних об’єктів дозволяє правильно зіставляти члени речення, узгоджувати їхні роди та числа, в чому сучасні методи автоматизованого перекладу все ще часто допускають помилки. Окрім того, знаходження кореферентних об’єктів разом у комбінації з перекладом текстів дозволяє провести адаптацію лінгвістичних підходів з однієї мови в іншу, таким чином в мови, для яких не створено специфічних засобів розв’язання задач їх обробки або для яких не існує розмічених наборів даних для

розв'язання лінгвістичних задач відносно просто можуть бути адаптовані такі методи.

Далі наведені приклади кореферентності (референт виділений **жирним шрифтом**, займенники - підкреслені).

Приклад простої анафори:

*Він перейшов через **гору**. Вона була високою.*

Приклад простої катафори:

*Вона вийшла на дорогу, що вела праворуч. **Марія** сьогодні мала гарний настрій.*

Приклад складеного антецедента.

***Іван, Михайло та Остап** – всі вони працювали під землею.*

1.2Особливості пошуку кореферентних об'єктів в україномовних текстах

У порівнянні з іншими мовами Європи – англійською, німецькою, французькою, італійською, іспанською – для української мови властивий довільний порядок слів, як і для інших слов'янських мов – польської, російської, сербської, хорватської. Наприклад [12], для української мови:

Карпо прикинув таке слівце, що батько перестав стругати і почав прислухатись. Він глянув на синів через хворостяну стіну. Сини стояли без діла й балакали, поспиравшись на заступи (Іван Нечуй-Левицький, «Кайдашева сім'я», оригінальний порядок слів).

З другого речення в прикладі можна перестановкою слів утворити інші граматично-коректні речення з дуже близькими значеннями, але різними наголосами, в залежності від задуму автора:

- *Через хворостяну стіну він глянув на синів.*

- *На синів він глянув через хворостяну стіну.*

- *Через стіну хворостяну він глянув на синів.*

- *Через хворостяну стіну глянув він на синів.*

- *Через хворостяну стіну на синів глянув він.*

В той же час в англійській мові можлива тільки одна комбінація: «He looked at his sons through the twig wall», оскільки в ній використовується стандартний порядок слів суб'єкт – дієслово – об'єкт (SVO, subject verb object). В інших мовах також поширений порядок суб'єкт – об'єкт – дієслово (SOV, subject object verb). У зв'язку з цим, алгоритм для української мови має передбачати роботу з різним порядком слів.

Також, метод для правильного визначення кореферентних об'єктів має враховувати відношення і між іншими об'єктами в тексті (присудками, обставинами, означеннями). Наприклад, в цьому уривку близькість слів «батько» та «він» підвищує імовірність їх кореферентності, але можлива й інша ситуація, коли слово «Карпо» може знаходитись ближче до слова «він», тому для правильного визначення потрібно співвіднести інші елементи речення. Зокрема, відношення підметів, дієслів та додатків «Карпо прикинув слівце» та «Сини стояли й балакали» має віднести підмет «Карпо» до групи «сини», далі частина «Він глянув на синів» має вказати на ще один суб'єкт, після цього розрізнивши слова «Карпо», «сини» та «батько» та їх дії, можна правильно зробити підстановку та класифікувати слова «батько» та «він» кореферентними.

Виявлення референтів та відповідних їм катафори та анафори дозволяє виділяти значущу інформацію з тексту. Так, знайшовши кореферентні об'єкти алгоритм може замінити катафори та анафори відповідними їм референтами. Це дозволяє робити речення незалежними від контексту, в якому вони вживаються. Як наслідок, речення можуть аналізуватися самостійно, без зв'язку з основним текстом. Також, виявлення

кореферентних об’єктів дозволяє полегшити переклад з однієї мови на іншу, особливо, якщо в них відрізняється порядок слів у реченні, що робить необхідним перестановку членів речення. Окрім того, знаходження кореферентних об’єктів дозволяє підвищити точність розв’язання інших задач обробки природної мови, наприклад, оцінки когерентності текстів.

1.3 Корпуси текстів для пошуку кореферентних об’єктів

Для розробки методів пошуку кореферентних об’єктів (навчання нейронних мереж, включаючи моделі природної мови, створення дерев рішень) в поширених у світі мовах, таких як англійська, існує велика кількість корпусів даних. Для української мови наразі наявні тільки 2 таких набори даних.

Корпус текстів №1, створений в університеті Тараса Шевченка, містить 2500 текстів. Формування набору проводилось напівавтоматично, з ручною розміткою кореферентних об’єктів, що значно обмежило об’єм корпусу. Для автоматичного виділення додаткових характеристик тексту використана бібліотека UDpipe [20] (детальніше в розділі 1.6).

Цей корпус текстів містить такі характеристики для кожного слова:

ID – ідентифікатор слова;

RawText – саме слово;

DocumentID – ідентифікатор тексту до якого входить слово;

WordOrder – порядковий номер слова в межах тексту;

PartOfSpeech – частина мови;

Lemmatized – початкова (лематизована) форма слова;

IsPlural – чи множина;

IsProperName – чи власна назва;

IsHeadWord – чи є слово головне у словосполученні;

Gender – рід;

EntityID – ідентифікатор об’єкта до якого входить слово (окреме чи входить в словосполучення);

CoreferenceGroupID – ідентифікатор кореферентної групи, до якої входить слово.

Корпус текстів №2 [21] було автоматично адаптовано з англомовного набору даних. Цей корпус містить в собі 36600 текстів. Кожен текст містить наступні поля:

doc_key – ідентифікатор документа;

clusters – список кластерів, де кожному кластеру відповідає список кореферентних об’єктів. Кожен кореферентний об’єкт складається зі списку, що містить два значення: перше – номер першого слова в кореферентному об’єкті, друге – номер останнього слова в кореферентному об’єкті;

sentences – список речень, де кожне речення представлене списком слів;

tokens – список окремих слів з речень.

Для адаптації набору використовувався підхід зіставлення слів, який дозволив мінімізувати похибку при перенесенні набору даних між мовами.

Для корпусу текстів важливими показниками є якість сформованих даних та його об’єм, чим вони більші, тим вищих результатів можна буде досягти при розв’язанні задачі. Великі та різноманітні корпуси дозволяють моделям краще узагальнювати знання та працювати на нових даних, забезпечуючи покриття різних ситуацій, стилів і контекстів. Корпуси текстів з низькою якістю (наприклад, з граматичними та синтаксичними помилками, некогерентними текстами з відсутністю логічного зв’язку) можуть

призводити до поганого узагальнення характеристик, важливих для аналізу об’єктів, оскільки модель навчається на неправильних прикладах. Окрім того, довші корпуси текстів дозволяють ефективно застосовувати глибші нейронні мережі з більшими наборами параметрів, які можуть виділяти складніші закономірності, ніж нейронні мережі менших розмірів.

1.4 Підходи до пошуку кореферентних об’єктів

Є чотири основних підходи до пошуку кореферентних об’єктів імовірнісними методами [22]:

- Підхід **Mention-pair** розглядає попарно всі комбінації об’єктів, що теоретично можуть бути кореферентними, та для кожної пари оцінює імовірність їх кореферентності.
- Підхід **Entity-mention** концентрується на знаходженні сутностей референтів в тексті, при цьому кореферентні об’єкти об’єднуються в кластери. Таким чином, на відмінну від підходу Mention-pair він використовує попередню інформацію для знаходження кореферентних об’єктів до об’єкта, аналізується в даний момент. Також, цей підхід має меншу обчислювальну складність, адже не розглядає попарно комбінації всіх об’єктів.
- Підхід **Mention-ranking** для знаходження кореферентних об’єктів обирає тільки найбільш імовірних кандидатів з відранжованих, що відрізняє її від моделі Mention-pair, яка розглядає на кожному кроці тільки два об’єкти, визначаючи чи вони кореферентні.
- Підхід **Cluster-ranking** комбінує підходи Mention-ranking та Entity-mention, використовуючи ранжування з Mention-ranking та кластеризацію з Entity-mention, об’єднуючи переваги обох підходів.

1.5 Методи пошуку кореферентних об’єктів

Для розв’язання задачі пошуку кореферентних об’єктів використовують декілька основних підходів. На початкових етапах досліджень використовувалися набори правил, що дозволяли виділяти кореферентні об’єкти. Набори правил можуть формуватися висококваліфікованими лінгвістами для конкретної мови, але їх створення вимагає значних зусиль і часу.

Згодом стали застосовуватися імовірнісні методи, що використовували дерева рішень, та, ще пізніше, штучний інтелект (генетичний алгоритм, нейронні мережі, автоматизована кластеризація). Спочатку алгоритми, що засновані на наборах правил були більш ефективними за методи з використанням штучного інтелекту, але після появи великих промаркованих наборів даних для англійської та інших мов вони перевершили попередні підходи до аналізу[1].

Великі мовні моделі стали новим етапом у розв’язанні задач пошуку кореферентних об’єктів. Ці моделі навчаються на величезних обсягах тексту, які охоплюють сотні природних мов та містять трильйони токенів (слів або їх частин, розділових знаків). Завдяки цьому, вони здатні враховувати широкий контекст при визначенні кореферентних зв’язків. Великі мовні моделі показали високу ефективність у різних задачах обробки природної мови, включаючи розпізнавання сутностей, аналіз тексту та машинний переклад.

1.5.1 Методи на основі наборів правил

Набори правил [4] (також часто називають фільтруючими ситами) дозволяють виділяти потенційно кореферентні об'єкти послідовно використовуючи правила до вхідних даних. Ці правила можуть бути створені кваліфікованими лінгвістами вручну, з урахуванням особливостей конкретної мови. Методи, що використовують тільки набори правил зазвичай містять велику кількість таких правил та використовують складну логіку, що

ґрунтується на використанні попередньо визначених лінгвістичних конструкцій та семантичних критеріїв для кореферентних об’єктів.

Алгоритми на основі простих правил (наприклад: належність до певної частини мови, співпадіння роду, числа та інші) використовуються як частина більш складних алгоритмів на етапі попереднього аналізу та утворення початкових кластерів кореферентних об’єктів.

1.5.2 Метод на основі дерев рішень

Дерево рішень — це ієрархічна структура, яка складається з вузлів (кореня – початкового вузла, внутрішніх та кінцевих - листкових вузлів) та ребер, що з'єднують ці вузли. Кожен вузол представляє собою певний елемент процесу прийняття рішення:

- Кореневий вузол - початковий вузол, з якого починається дерево; він містить увесь набір даних і розділяється на підмножини на основі певного критерію.
- Внутрішні вузли - представляють собою пункти прийняття рішень, де виконується перевірка на певну характеристику об’єктів з набору даних. Кожен некінцевий вузол такого дерева посилається на два піддерева (або вузла-нащадка), які відповідають можливим результатам такої перевірки.
- Листові вузли - кінцеві вузли, які містять остаточне передбачення, за яким класифікуються приклади, які досягли цього вузла.

Ребра (гілки) дерева рішень з'єднують вузли та показують шлях від одного вузла до іншого, відображаючи послідовність прийняття рішень. На кожному ребрі вказується умова або значення ознаки, яке визначає напрямок руху по дереву. Така структура дозволяє наочно представити процес прийняття рішень та забезпечує зрозумілий механізм для класифікації або прогнозування на основі вхідних даних.

Дерева рішень є потужним інструментом для моделювання процесів прийняття рішень та аналізу даних. Вони використовуються для прогнозування результатів на основі заданих вхідних даних шляхом ієрархічного розгалуження можливих варіантів. Кожен внутрішній вузол дерева представляє перевірку або тест на певну ознаку, гілки відповідають можливим результатам цього тесту, а листові вузли (або листя) вказують на кінцеве рішення або прогноз.

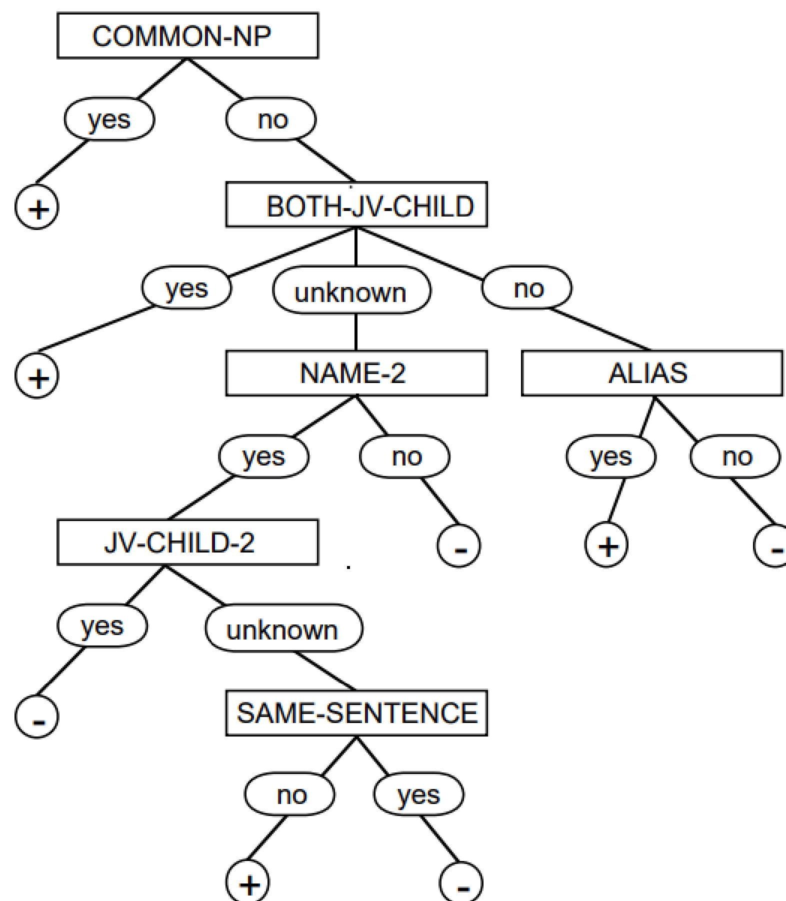

Рис. 1.1 - Представлення дерева рішень для пошуку кореферентних об’єктів[5]

Для формування дерев рішень використовується ряд алгоритмів, що спрямовані на досягнення найбільш ефективного розділення елементів множин на підмножини на кожному кроці.

Дерева рішень дозволяють розв'язувати задачу пошуку кореферентних об'єктів завдяки можливості опису пар кореферентних об'єктів певними характеристиками [5] (детальніше в розділі 2.1). При цьому задачу пошуку кореферентних об'єктів можна звести до задачі класифікації пар потенційно кореферентних об'єктів. Приклад дерева рішень для пошуку кореферентних об'єктів зображено на рис. 1.1.

1.5.3 Методи, що використовують нейронні мережі

Для пошуку кореферентних об'єктів в текстах можна використовувати нейронні мережі[2, 3, 6, 7]. Алгоритми на основі нейронних мереж адаптивні до особливостей різних мов та не вимагають у інженера наявності глибоких знань у лінгвістиці. Нейронні мережі, як і дерева рішень, можуть використовувати лінгвістичні характеристики для пошуку кореферентних об'єктів, але разом із тим також можуть використовувати векторні представлення слів і аналізувати речення в цілому, а не тільки пари об'єктів. Як і для дерев рішень, задачу пошуку кореферентних об'єктів можна звести до бінарної класифікації. Для пошуку кореферентних об'єктів використовуються різні архітектури нейронних мереж.

Повнозв'язна архітектура нейронних мереж (Feedforward neural network) – це перша архітектура, що дозволила формувати глибокі нейронні мережі. Вона складається з набору шарів, кожен з яких містить певну кількість нейронів, при цьому кожен нейрон попереднього шару пов'язаний з кожним нейроном наступного шару. Робота нейрона визначається його прихованими параметрами. Окремий нейрон містить ваги, що визначають, на скільки сильно сигнал від попереднього нейрона впливає на формування вихідного сигналу, та параметр зміщення (bias), які підсумовуються для отримання вихідного сигналу. Також, для уможливлення апроксимації нелінійних функцій до вихідного сигналу кожного нейрону застосовується нелінійна функція активації перед подачею в наступний шар. Для навчання нейронної мережі застосовують алгоритм зворотного розповсюдження

помилки, що при наявності зразків, які містять вхідні та вихідні дані, уможливлює автоматичну корекцію ваг нейронної мережі для отримання правильних передбачень вихідної інформації залежно від вхідних даних. Повнозв’язні нейронні мережі наразі рідко самостійно використовуються для розв’язання задач обробки природної мови, але застосовуються у комбінації з іншими типами штучних нейронних мереж, наприклад як шари фінальної обробки.

Рекурсивна архітектура нейронних мереж (Recursive neural network, RvNN) – архітектура, в якій виходи рекурсивних шарів пов’язані з входами того ж шару, таким чином сигнал оброблюється рекурсивно. Такі мережі дозволяють обробляти дані змінної довжини та успішно використовуються при вирішенні задач обробки природних мов. Рекурсивні нейронні мережі включають в себе рекурентні нейронні мережі (Recurrent neural network, RNN), до яких входять мережі з довгою та короткочасною пам’яттю (Long short-term memory, LSTM) та мережі GRU (Gated Recurrent Unit).

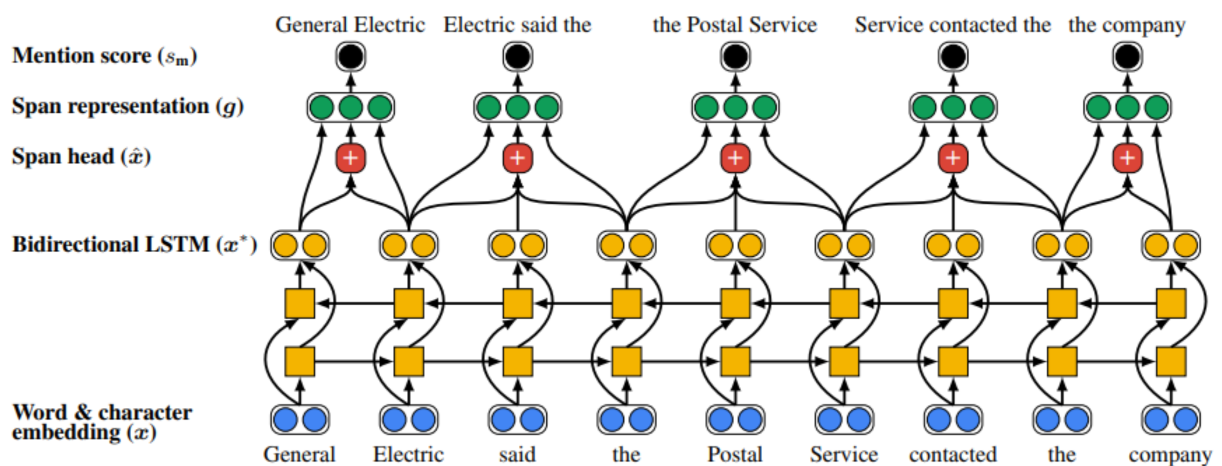

Рис. 1.2 Двонаправлена нейронна мережа із довгою та короткочасною пам’яттю для пошуку кореферентних об’єктів[7]

Рекурсивні нейронні мережі підходять для аналізу послідовностей, до яких належать тексти – послідовності слів, що можуть бути представлені в векторному вигляді. Для пошуку кореферентних об’єктів в попередніх

роботах розглядалися рекурентні нейронні мережі (RNN) [6] та різновид LSTM – двонаправлену нейронну мережу з довгою та короткочасною пам'яттю (BiLSTM)[2, 7] використану для пошуку кореферентних об'єктів. Її архітектуру зображено на рис. 1.2.

Згорткові архітектура нейронних мереж (Convolutional neural network, CNN) містить в собі 2 типи шарів: згорткові шари та шари субдискретизації (Pooling). В згорткових шарах набори ядер (матриць розмірності 3x3, 5x5 …) застосовуються до вхідних даних, з якими виконується операція згортки, що формує вихідні карти характеристик. До цих карт застосовується операція субдискретизації, що зменшує розмірність даних. Після шарів згортки використовують шар для об'єднання даних з карт характеристик в одновимірний вектор, що в подальшому обробляється повнозв'язними шарами.

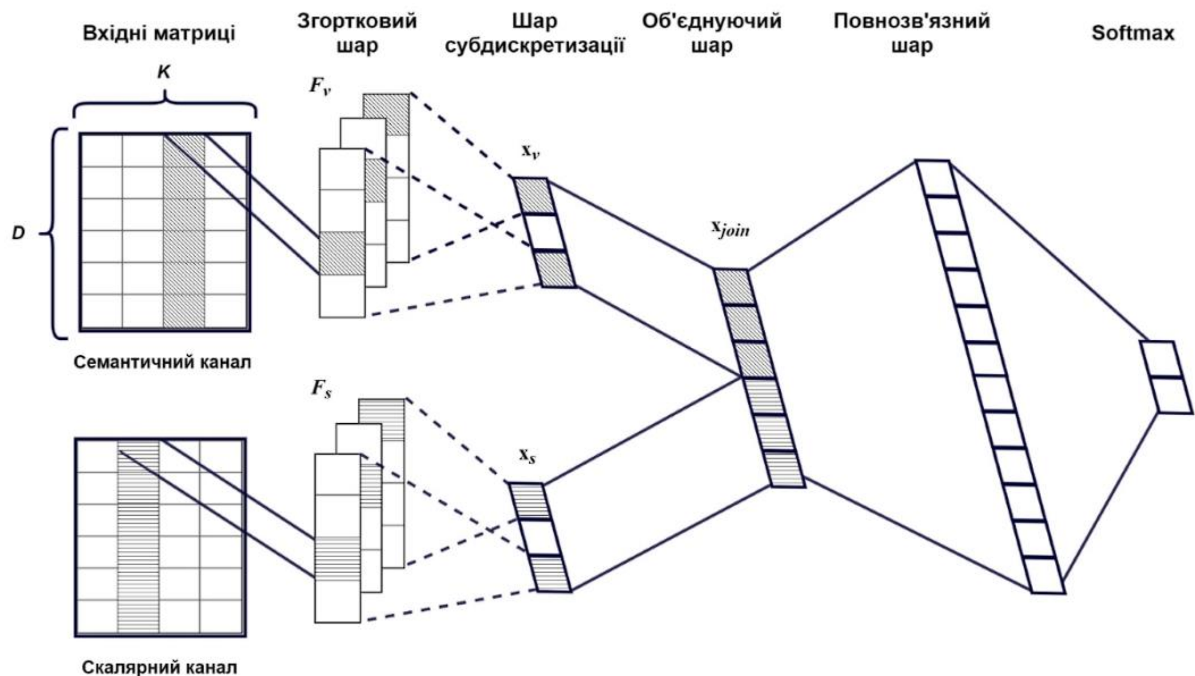

Рис. 1.3. Згорткова нейронна мережа [3]

Завдяки особливостям архітектури, згорткові нейронні мережі підходять для обробки великих об'ємів даних, таких як фото, відео та

векторні представлення слів. Згорткові шари нейронної мережі дозволяють виділяти значущі характеристики, які далі обробляються повнозв'язними шарами. Приклад використання згорткової нейронної мережі для пошуку кореферентних об'єктів[3] зображено на рис. 1.3.

Методи на основі двонаправленої нейронної мережі з довгою та короткочасною пам'яттю (BiLSTM) [2] та згорткової нейронної мережі (CNN) [3] використовувалися для пошуку кореферентних об'єктів в україномовних текстах у попередніх дослідженнях. Для їх навчання використовувався корпус №1, описаний в розділі 1.3. Результати їх роботи приведені в розділі 4.3 для порівняння.

1.5.4 Методи, засновані на використанні моделей природної мови

Моделі природної мови – це спеціалізовані моделі, засновані на архітектурі Transformer [8] та призначені для обробки та генерації текстів. Архітектура Transformer складається з двох основних блоків: *кодера (encoder)* та *декодера (decoder)* та призначена для перетворення вхідної послідовності слів, знаків (або токенів – частин слова, розділових знаків) у вихідну послідовність. Кожен блок складається з декількох шарів, організованих послідовно.

Блок кодера: Обробляє вхідну послідовність та перетворює її в контекстуалізоване представлення.

Декодер: Використовує контекстуалізоване представлення, отримане від кодера, для генерування вихідної послідовності (наприклад, переклад, відповідь на запитання).

Ключовою особливістю архітектури Transformer є використання механізму уваги, який дозволяє враховувати значення всіх попередніх слів та визначати найбільш значущі з них для формування наступного слова у вихідній послідовності.

При розв'яанні задачі пошуку кореферентних моделей, великі мовні моделі використовуються двома способами:

Попередньо навчені на текстових корпусах з трильйонами токенів, великі мовні моделі генерують значно точніші та контекстуально чутливіші векторні представлення порівняно з попередніми підходами (наприклад, Word2Vec [23]). Це дозволяє моделі пошуку кореферентних об'єктів, яка використовує проміжні векторні представлення слів великої мовної моделі, краще вловлювати семантичні та синтаксичні відношення між потенційно кореферентними об'єктами, таким чином покращуючи якість розв'язання задачі пошуку кореферентних об'єктів.

Великі мовні моделі можуть безпосередньо використовуватися для пошуку кореферентних об'єктів, тобто аналізувати запит, що містить завдання і текст, та генерувати відповідь (детальніше пункт 3.4). Завдяки навчанню на різноманітних корпусах текстів, вони мають знання про реальність, які необхідні для розв'язання складних випадків знаходження кореферентних об'єктів та здатні слідувати інструкціям, що задані в запиті (prompt), повертаючи відповідь на питання в заданому форматі. Окрім того, великі мовні моделі можна донавчати на обмежених маркованих корпусах текстів, що дозволяє їм завдяки попереднім знанням навчатися значно швидше та краще, ніж спеціалізованим моделям, навченим з початку. Також великі мовні моделі можуть використовуватися і для створення корпусів текстів, оскільки здатні автоматизовано генерувати марковані набори даних, що далі можуть бути використані для донавчання інших моделей.

Пошук кореферентних об'єктів є важливою складовою оцінки правильності роботи великих мовних моделей. Він може застосовуватися для оцінки якості її роботи. Також, для оцінки правильності роботи великих мовних моделей існують спеціалізовані тести - Winograd Schema Challenge, що полягають у використанні пар речень, які відрізняються декількома словами, але це повністю змінює значення займенника, що вживається в

тексті, наприклад (займенник підкреслений, кореферентний йому об’єкт виділено жирним):

Шапка не вмістилася у валізу, оскільки вона була завеликою.

Шапка не вмістилася у **валізу**, оскільки вона була замалою.

Такі приклади дозволяють оцінити здатність моделі враховувати значення слів у складних, контекстнозалежних випадках. Розширені модифікації цього тесту, такі як WinoGrande Schema Challenge містять великий набір складних випадків кореферентності та використовуються для оцінки найсучасніших мовних моделей.

Знаходження кореферентних об’єктів допомагає формувати формалізовану структуру речень, що є корисним при створенні графів знань (Knowledge Graphs). Інтеграція великих мовних моделей з графами знань дозволить значно розширити можливості великих мовних моделей та наблизитися до створення штучного інтелекту загального призначення (Artificial General Intelligence, AGI) [25, 26].

1.5.5 Порівняння методів пошуку кореферентних об’єктів

В табл. 1 наведено порівняння методів пошуку кореферентних об’єктів. Інтелектуальні методи на основі нейронних мереж, дерев рішень та моделей мови в цілому показують вищу продуктивність у порівнянні з алгоритмами на наборах правил [4]. Докладніше їх ефективність буде проаналізована в наступних розділах.

Таблиця 1.1 Переваги та недоліки різних методів пошуку кореферентних об’єктів

Методи	Автоматична адаптація під особливості мови	Швидкодія	Можливість автоматичного створення додаткових проміжних характеристик	Можливість аналізу внутрішніх правил роботи користувачем	Необхідність використання наборів текстів для створення рішення
Набори правил	Ні	Порівняно висока	Ні	Так	Ні
Дерева рішень	Так	Порівняно висока	Ні	Так	Так
Нейронні мережі	Так	Бажане прискорення для навчання	Так	Ні	Так
Моделі природних мов	Так	Бажане прискорення для навчання	Так	Опціональна	Так

1.6 Програмні засоби, що використовуються в алгоритмах пошуку кореферентних об’єктів

Мова програмування Python є основною при написанні застосунків для автоматизованого пошуку кореферентних об’єктів. Вона широко використовується для наукових досліджень та написання практичних застосунків завдяки зручності роботи та наявності великої кількості бібліотек.

Для реалізації алгоритмів з пошуку кореферентних об’єктів необхідно отримати векторні представлення слів, що дозволяє бібліотека ELMo

(Embeddings from Language Models) [27]. Ця бібліотека дозволяє перетворювати слова у вектори, що відповідають їх семантичному, лексичному, синтаксичному значенню. На відмінну від інших бібліотек, що дозволяють формувати векторні представлення слів, таких як Word2Vec [23], векторні представлення сформовані в ELMo враховують не тільки значення окремого слова, але й значення навколишніх слів, що дозволяє краще знаходити зв'язки між окремими словами та реченнями. ELMo використовує нейронні мережі для отримання векторних представлень слів та вимагає навчання. Використана версія адаптована для української мови.

Використовується бібліотека Scikit-learn [28]. Ця бібліотека включає багато засобів для розв'язання задач регресії, кластеризації, класифікації, зокрема містить високоефективну реалізацію дерев рішень з можливістю налаштування параметрів побудови дерева.

Однією з головних переваг дерев рішень над іншими алгоритмами є можливість візуалізації побудованого дерева. Для цього використано бібліотеку Graphviz [29].

Для отримання додаткової інформації при обробці текстів (рід, число лематизована (початкова) версія слова) використано бібліотеку UDpipe [20], що використовує нейронні мережі та навчена на україномовному корпусі текстів.

Для роботи з корпусами текстів та обробки масивів використовувалися бібліотеки Json [30], HuggingFace Datasets [31] та Numpy [32].

Бібліотеки TensorFlow [33] та PyTorch [34] пропонують високоефективні реалізації шарів нейронних мереж та використовуються як для реалізації власних нейронних мереж, так і для забезпечення внутрішнього функціонування та донавчання великих мовних моделей. Також, для донавчання моделей використано бібліотеку Unsloth [35], що

включає в себе більш ефективні реалізації алгоритму донавчання, ніж ті, що за замовчуванням включені в бібліотеки TensorFlow та PyTorch.

Для генерації відповідей (inference) великими мовними моделями використано бібліотеку OpenAI [36]. Вона надає інтерфейс, що широко використовується для віддаленого підключення до великих мовних моделей, запущених як локально, так і на віддаленому сервері.

Більшість мовних моделей, використаних в досліджені, виконувались локально використовуючи застосунок LM Studio [37], що заснована на бібліотеці llama.cpp [38]. LM Studio надає зручний графічний користувацький інтерфейс, що допомагає в швидкому розгортанні мовних моделей, їх тестуванні як в графічному інтерфейсі, так і з функціональністю локального сервера, до якого можна підключитися використовуючи бібліотеку OpenAI.

Для генерації відповідей великими мовними моделями, розміщеними на віддалених серверах компанії Google використано бібліотеку GenAI [39].

1.7 Використання високопродуктивних обчислень при розв’язанні задачі знаходження кореферентних об’єктів

Обробка великих об’ємів даних нейронною мережею вимагає наявності потужних обчислювальних ресурсів, особливо на етапі навчання. Структура нейронних мереж дозволяє реалізувати паралельну обробку даних. Для обробки даних за допомогою нейронних мереж використовують наступні пристрої [10]:

- Центральні процесори (CPU) підходять для обробки невеликих об’ємів даних на простих нейронних мережах. Це пов’язано з використанням невеликої кількості потужних ядер, що пристосовані до швидкого виконання послідовних алгоритмів. Хоча сучасні процесори мають значно більшу кількість ядер, ніж їх попередники (так, найпотужніші процесори для настільних комп’ютерів 8 років тому мали до 4 обчислювальних ядер, впродовж трьох наступних років число ядер виросло до 16, а зараз досягає

24) та використовують спеціалізовані інструкції (AVX256, AVX512) для роботи з матричними обчисленнями, із ростом складності даних та структури нейронних мереж їх навчання на центральних процесорах займає дуже багато часу, що ускладнює процес розробки. Також, хоча сучасні настільні процесори підтримують відносно великий об'єм ОЗП (до 192 ГБ DDR5), її швидкість (до 102 ГБ/с DDR5-6400) обмежує швидкодію при роботі з великими мовними моделями.

- Графічні процесори (GPU), у зв'язку із своєю паралельною структурою, адаптованою для обробки графічної інформації, добре підходять і для прискорення навчання нейронних мереж. Сучасні графічні процесори містять на порядки більше обчислювальних ядер, ніж процесори загального призначення. Наприклад, непрофесійна відеокарта RTX 5090 (2025) має 21760 ядер на архітектурі Blackwell (2.0, для настільних GPU) та 32ГБ пам'яті GDDR7, до 1.7 ТБ/с. Так як нейронні мережі, зазвичай, не вимагають високої точності представлення даних, для обчислень часто використовують числа з нижчою точністю, наприклад, FP16, FP8/Int8, FP4/Int4 замість FP32, що дозволяє отримати додаткове прискорення та економити пам'ять. Окрім того, графічні процесори на архітектурах Turing та новіші містять у своєму складі виділені тензорні ядра (Tensor Cores), що дозволяють графічному процесору досягти вищої продуктивності у порівнянні з ядрами CUDA під час аналізу даних нейронною мережею. Удосконалення тензорних ядер, що введені в архітектурі Ampere разом із можливістю не обчислювати нульові коефіцієнти в нейронних мережах, дозволили підвищити продуктивність в 2.7 рази у порівнянні з попередньою архітектурою для відеокарт однакового рівня [40]. Наступні архітектури графічних прискорювачів Nvidia додавали підтримку обчислень з FP8/Int8 (Ada Lovalace), FP4 (Blackwell), що призвело до подвоєння продуктивності для чисел з нижчою точністю між поколіннями. Спеціалізований прискорювач B200 [41] на архітектурі Nvidia

Blackwell (версія для прискорювачів) містить 33 792 ядра CUDA та 192 ГБ пам’яті HBM3e зі швидкістю до 8 ТБ/с.

- Програмовані логічні інтегральні схеми (ПЛІС, FPGA) мають у своєму складі велику кількість логічних елементів, які можуть окремо програмуватися під потреби та особливості конкретної архітектури нейронної мережі. Корпорація Intel розробляє ПЛІС [42], призначені спеціально під прискорення обчислень нейронних мереж.

- Мікросхеми спеціального призначення (ASIC) адаптовані для операцій з нейронними мережами. До них належать Google Tensor Processing Unit (TPU) [43], що через використання спеціалізованої архітектури дозволяють досягти високої продуктивності на Ватт, у порівнянні з традиційними GPU, хоча найновіші графічні процесори також мають спеціальні ядра призначені для нейронних мереж. Google TPU доступний для покупки тільки у вигляді малопотужного копроцесора, призначеного для обчислень з низькою затримкою (Edge Computing). Завдяки технології хмарних обчислень, Google надає доступ дослідникам до потужніших тензорних прискорювачів.

- Аналогові мікросхеми з підтримкою обчислень в пам’яті (Analog in-Memory Computing, AiMC) – це новий тип аналогових мікросхем, що використовують технологію обчислень в пам’яті, яка дозволяє обійти основне обмеження архітектури Фон-Неймана – необхідність очікувати інформацію з пам’яті для її обробки подальшої обробки. Так, як нейронні мережі не вимагають високої точності обчислень, для них в якості обчислювальних елементів підходять і аналогові помножувачі, що значно простіші за будовою, як наслідок, займають значно менше місця на схемі та споживають менше енергії. Дослідження таких мікросхем активно проводять в IBM [44, 45] та IMEC [46]. Працюючі дослідні чіпи досягли дуже високих показників енергоефективності [47] – до 2900 TOPS/Вт.

- Cerebras Wafer-Scale Engine 3 (WSE-3) [48]– це різновид спеціалізованого цифрового прискорювача нейронних мереж, який є єдиною інтегральною схемою розміру кремнієвої пластини, що містить 900 000 обчислювальних ядер, 44 ГБ кеш пам’яті (до 20 ПБ/с) та має споживання в 23 кВт [41]. Вони використовуються для навчання особливо великих моделей мов завдяки підтримці підключення до 1.2 ПБ зовнішньої пам’яті на чіп та можливості швидкого навчання та генерації відповідей завдяки розміщенню ваг мовної моделі прямо в кеш пам’яті одного чи декількох таких прискорювачів.

Таблиця 1.2 Порівняння швидкодії різних типів обчислювальних пристроїв

Тип обчислювального пристрою	**Модель**	**Швидкодія, TOPS**	**Енергоефективність, TOPS/Вт**
CPU	AMD Ryzen 9950X	5 (FP32, FP16, INT8)	0.04
GPU	Nvidia RTX 5090	1676 (INT8), 3352 (INT4)	2, 4
GPU (ASIC)	Nvidia DGX B200	4500 (FP16), 9000 (FP8), 18000 (FP4)	2.5, 5 10
FPGA	Intel Stratix 10 NX	143(INT8)	1
ASIC	Google Edge TPU	4 (INT8)	2
ASIC	Cerebras WSE-3	125 000 (FP16, FP8)	5

AiMC	Прототип (IMEC)	5.8 (23.5 max)	2900 (1000-1500)

Швидкодія обчислювальних пристроїв, що використовуються в галузі штучного інтелекту для прискорення нейронних мереж вимірюється в трильйонах операцій за секунду (TOPS) [49]. Для метрики TOPS важливо уточнювати формат чисел, що використовуються при обчисленнях. Також, для різних архітектур нейронних мереж відносні показники продуктивності можуть сильно відрізнятися в залежності від обчислювальних пристроїв, тобто, деякі з них будуть ефективними для одних типів нейронних мереж, деякі - для інших.

1.8 Висновки з 1 розділу

У першому розділі дисертаційного дослідження представлено огляд та аналіз предметної області – автоматизованого пошуку кореферентних об'єктів у текстах, з особливим акцентом на специфіку української мови. Цей розділ виконує ключову роль у означенні проблеми, обґрунтуванні її актуальності та визначенні теоретичного й практичного підґрунтя для подальших досліджень, представлених у наступних розділах.

Ключові аспекти розділу:

Сформульовано визначення кореферентності в лінгвістиці, включаючи поняття антецедента, постцедента, анафори та катафори, а також складні випадки (складені референти, множинні зв'язки).

Обґрунтовано важливість та актуальність задачі пошуку кореферентних об'єктів для широкого спектру завдань обробки природної мови (NLP), таких як виділення інформації, оцінка когерентності тексту, машинний переклад (особливо для узгодження граматичних категорій) та адаптація лінгвістичних підходів між мовами. Знаходження кореферентних

об'єктів допомагає в створенні графів знань, об'єднання яких з великими мовними моделями вважають одною із складових для створення систем штучного інтелекту загального призначення (AGI).

Виокремлено та проаналізовано ключові особливості української мови, що ускладнюють вирішення задачі пошуку кореферентних об’єктів у порівнянні з мовами з фіксованим порядком слів (наприклад, англійською). Головний акцент зроблено на вільному порядку слів, що вимагає від алгоритмів здатності працювати з різними синтаксичними структурами та враховувати ширший контекст і семантичні зв'язки між різними членами речення.

Проведено аналіз наявних корпусів текстів для української мови, придатних для розробки та тестування моделей пошуку кореферентних об’єктів. Підкреслено обмеженість обсягу та різноманітності доступних даних, що є суттєвим викликом для навчання моделей, особливо нейромережевих. Обговорено важливість якості та обсягу даних для ефективності моделей.

Систематизовано та класифіковано основні підходи до пошуку кореферентних об'єктів (Mention-pair, Entity-mention, Mention-ranking, Cluster-ranking), що дає уявлення про різні стратегії моделювання.

Здійснено детальний огляд еволюції методів: від наборів правил (фільтруючих сит) до імовірнісних методів на основі дерев рішень та сучасних підходів із застосуванням штучного інтелекту, зокрема різних архітектур нейронних мереж - повнозв'язних, рекурсивних (RNN, LSTM, BiLSTM), згорткових (CNN).

Особливу увагу приділено новітньому етапу – використанню великих мовних моделей (LLM) на базі архітектури Transformer, як для генерації якісних векторних представлень, так і для безпосереднього вирішення задачі кореференції, а також для оцінки їх роботи (Winograd Schema Challenge).

Наведено порівняльну таблицю методів, що узагальнює їх переваги та недоліки стосовно адаптивності, швидкодії, інтерпретованості та потреби в даних.

Представлено огляд ключових програмних засобів та бібліотек, що використовуються для реалізації сучасних методів NLP та машинного навчання, зокрема для вирішення задачі кореференції.

Обґрунтовано необхідність використання високопродуктивних обчислень (HPC) для навчання та застосування складних моделей, в особливості нейронних мереж та великих мовних моделей (LLM). Проаналізовано різні типи обчислювальних пристроїв, їх характеристики та придатність для прискорення обчислень в галузі штучного інтелекту (ШІ).

З наведеного можна зробити наступні висновки:

1. Знаходження кореферентних об’єктів – актуальна задача, яка недостатньо досліджена для української мови. Її рішення дозволить полегшити розв’язання інших задач обробки природної мови. Рішення задачі пошуку кореферентних об’єктів для української мови має свої особливості.
2. Використання наборів правил для пошуку кореферентних об’єктів має багато недоліків, зокрема, необхідність розробки правил висококваліфікованими спеціалістами вручну, адаптованість під особливості конкретної мови, що ускладнює переносимість методу на інші мови, нижча ефективність у порівнянні з алгоритмами, що використовують методи штучного інтелекту. Незважаючи на це, прості набори правил використовуються для попередньої обробки текстів перед їх подальшим аналізом методами штучного інтелекту.
3. Для пошуку кореферентних об’єктів варто дослідити використання дерев рішень. На відмінну від інших алгоритмів для пошуку кореферентних об’єктів, алгоритми на основі дерев рішень дозволяють

формувати дерево рішень, яке може бути візуалізоване. Це дозволяє відносно легко перевіряти логіку роботи дерева. Дерева рішень формуються в автоматичному режимі, що значно зменшує об'єми ручної праці у порівнянні з алгоритмами на основі наборів правил.

4. Алгоритми на основі нейронних мереж дозволяють автоматизовано формувати власні математичні функції в проміжних шарах, таким чином інтерпретуючи дані на вході.
5. Моделі мови, також засновані на використанні нейронних мереж, але при цьому використовують спеціальну архітектуру Transformers, що підходить для розв'язання широкого спектру проблем обробки природних мов. Такі моделі показують високу результативність, розробка нових моделей, таких як GPT4, наразі є передовим напрямком в дослідженнях автоматизованої обробки природних мов.

Висновки дозволяють сформувати мету дисертаційного дослідження: створення методів автоматизованого пошуку кореферентних об'єктів в україномовних текстах.

Для досягнення мети необхідно:

1. Дослідити та сформувати метод пошуку кореферентних об'єктів в україномовних текстах.
2. Використати нейронні мережі для створення методу пошуку кореферентних об'єктів в україномовних текстах.
3. Створити метод пошуку кореферентних об'єктів в україномовних текстах на основі великих мовних моделей.
4. Донавчити велику мовну модель на корпусі україномовних текстів для застосування в методі пошуку кореферентних об'єктів.
5. Дослідити використання високопродуктивних обчислень для прискорення роботи алгоритмів пошуку кореферентних об'єктів.

РОЗДІЛ 2. СТВОРЕННЯ МЕТОДІВ ПОШУКУ КОРЕФЕРЕНТНИХ ОБ’ЄКТІВ В УКРАЇНОМОВНИХ ТЕКСТАХ НА ОСНОВІ НЕЙРОННИХ МЕРЕЖ ТА ДЕРЕВ РІШЕНЬ

2.1 Метод пошуку кореферентних об’єктів на основі дерев рішень

2.1.1 Лінгвістичні характеристики, що використовуються для пошуку кореферентних об’єктів методами на основі дерев рішень

Для опису кореферентних об’єктів для алгоритмів на основі дерев рішень використані наступні лінгвістичні характеристики [12, 13, 14]:

- Косинусна схожість векторів семантичного представлення об’єктів, що розглядаються. Для отримання векторних представлень використовувалася попередньо навчена модель ELMo (детальніше 1.6).

- Кількість слів між обраними об’єктами.

- Кількість об’єктів між обраними потенційно кореферентними об’єктами.

- Булеве значення, правдиве, якщо перший об’єкт займенник.

- Булеве значення, правдиве, якщо другий об’єкт займенник.

- Булеве значення, правдиве, якщо лематизовані версії (початкові форми слів) об’єктів збігаються. В алгоритмі співпадіння лематизованих версій визначено як співпадіння хоча б одного слова в обох об’єктах, що розглядаються.

- Булеве значення, правдиве, якщо в обох об’єктів однакове число (однина чи множина).

- Булеве значення, правдиве, якщо в обох об’єктів однаковий рід.

- Булеве значення, правдиве, якщо обидва об’єкти власні назви.

Їх отримання проводиться автоматично, використовуючи спеціальні бібліотеки (детальніше п 1.6) або вручну розроблені засоби. Використання

лінгвістичних характеристик, разом із деякими іншими даними уможливлює пошук кореферентних об'єктів.

2.1.2 Підготовка текстів для аналізу деревом рішень

Для аналізу текстів за допомогою дерев рішень, їх потрібно подати у правильному вигляді. У роботі для створення дерева рішень використано характеристики для опису кореферентних об'єктів з розділу 2.1

На вхід алгоритму подається текст, що складається із окремих слів, пунктуації, та додаткової інформації, підготовленої за допомогою бібліотеки UDpipe. Сюди входить рід, число, лематизовані версія слова, частина мови. Також, для слів, що входять в кореферентні групи вказується ідентифікатор, що дозволяє віднести конкретне слово або словосполучення до кореферентної групи (підготовлений вручну).

Для роботи алгоритму для кожного тексту, що розглядається, формуються список (Python list), що вміщуєш індекси слів, що входять в словосполучення, які є потенційно кореферентними об'єктами, до їх об'єднання в кореферентні кластери, але у форматі, що полегшує подальше їх об'єднання (2.1), w – окреме слово, $\left[w_{i,1}, w_{i,2} \dots\right]$ - словосполучення.

$$ObjectsList = [\dots \left[\left[w_{i,1}, w_{i,2} \dots\right]\right], \left[\left[w_{i+1,1}, w_{i+1,2} \dots\right]\right] \dots] \qquad (2.1)$$

Також, створюються списки, що вміщають в себе правильно сформовані кластери кореферентних об'єктів. Ці списки необхідні на етапі порівняння кластерів, отриманих в результаті передбачень дерева рішень та вірних кластерів (2), C_1 , $\left[\left[w_{i,1}, w_{i,2}\right], \left[w_{j,1}, w_{j,2}\right]\right]$ – окремий кластер.

$$CorrectObjectsList =$$
$$[C_1, C_2, \dots \left[\left[w_{i,1}, w_{i,2} \dots\right], \left[w_{j,1}, w_{j,2} \dots\right] \dots\right], \left[\left[w_{i+1,1}, w_{i+1,2} \dots\right] \dots\right] \dots] \qquad (2.2)$$

Оскільки задача кластеризації зведена в алгоритмі до задачі класифікації, для створення дерев рішень та передбачень з їх допомогою

необхідні списки з параметрами кожної пари потенційно кореферентних об'єктів (2.3), де $\cos Sim$ – косинусна схожість об'єктів, $nWBtw$– кількість слів між об'єктами, що розглядаються, $nObjBetw$ – кількість об'єктів між об'єктами, що розглядаються, $len1$ – довжина (кількість слів) першого об'єкту, $len2$ – довжина другого об'єкту, $1pron$ – чи перший об'єкт – займенник, $2pron$ – чи другий об'єкт – займенник, $1prp$– чи перший об'єкт – власна назва, $2prp$ - чи другий об'єкт – власна назва, $lemS$ – чи лематизовані версії об'єктів збігаються, $gendS$ – чи об'єкти мають однаковий рід, $numS$– чи об'єкти мають однакове число.

$$X = [\begin{bmatrix} \cos Sim_1, nWBtw_1, nObjBtw_1, len1_1, len2_1, 1pron_1 \\ , 2pron_1, 1prp_1, 2prp_1, lemS_1, gendS_1, numS_1 \end{bmatrix}, \ldots] \quad (2.3)$$

Також необхідні мітки, що вказують, чи є пара об'єктів, що розглядаються, є кореферентними (2.4).

$$Y = [y_1,\ y_2,\ y_3, \ldots] \quad (2.4)$$

Після створення дерева рішень під час його використання для передбачень із списку (1) утворюється список (2.5), що містить список передбачених кластерів.

$$PredictedObjectsList = [C_1, C_2, \ldots \left[[w_{i,1}, w_{i,2} \ldots], [w_{j,1}, w_{j,2} \ldots] \ldots\right], \left[[w_{i+1,1}, w_{i+1,2} \ldots] \ldots\right] \ldots] \quad (2.5)$$

Отримані списки з характеристиками груп кореферентних об'єктів (2.3) а також їх маркування (2.4), що містять 2400000 зразків кореферентних та некореферентних об'єктів, діляться на дві частини – перша (1500 текстів, ~ 60%) використовується для формування дерева рішень, друга (1015 текстів, ~ 40%) – для перевірки результативності алгоритму (аналіз отриманих результатів).

2.1.3 Створення дерева рішень

Дерево рішень [12] реалізоване з використанням бібліотеки Scikit-learn[28], яка має клас дерева рішень sklearn.tree.DecisionTreeClassifier. При створенні екземпляру класу визначаються параметри дерева рішень – критерії розділення на піддерева під час формування, максимальна глибина дерева, мінімальна кількість елементнів для розділення, мінімальна кількість елементів в одному листку дерева, ваги для класів, що передбачаються та інші. За замовчуванням параметри дерева вказані для отримання безпомилкової конфігурації дерева на вибірці, що використовується для його формування. Такий підхід веде до надмірної адаптації дерева до даних, що використовуються при його формуванні та знижує точність роботи на наборах, які раніше не аналізувалися алгоритмом. Також, для великих об’ємів даних внаслідок використання такої конфігурації створюється надміру велике дерево, формування якого займає багато часу.

Таким чином постає необхідність обмежити розміри дерева для досягнення вищих результатів на даних, що не використовувалися для формування дерева. Для цього використано параметр min_impurity_decrease, що дозволяє визначати мінімально достатнє значення зменшення неоднорідності в наступних піддеревах при розділенні. На відмінну від інших способів обмеження розмірів дерева, таких як обмеження глибини або обмеження на кількість елементів в листях, цей показник дозволяє більш рівномірно обмежувати розміри дерева.

Для формування дерева підготовлені дані пар потенційно кореферентних об’єктів з маркуванням, чи є вони кореферентними подаються в функцію fit. На виході отримується дерево, здатне аналізувати пари кореферентних об’єктів. Таким чином, задачу кластеризації кореферентних об’єктів зведено до задачі класифікації деревом рішень.

Частина отриманого дерева показана на Рис. 2.1. Для ілюстрації глибина дерева та кількість текстів, що аналізуються, штучно обмежена, щоб мати змогу помістити отримане дерево на екрані. Як видно з рисунка, розбиття на кореферентні та некореферентні об'єкти починається із характеристики, що дозволяє найкраще розділити поточну групу об'єктів – співпадіння лематизованих версій об'єктів. Далі, завдяки використанню дерев рішень, в піддереві дерева (Рис. 2.2) можна прослідкувати наступну логіку – якщо лематизовані версії об'єктів співпадають, довжина першого та другого об'єкту – 1 та перший об'єкт є власною назвою, тоді з високою імовірністю пара об'єктів є кореферентними.

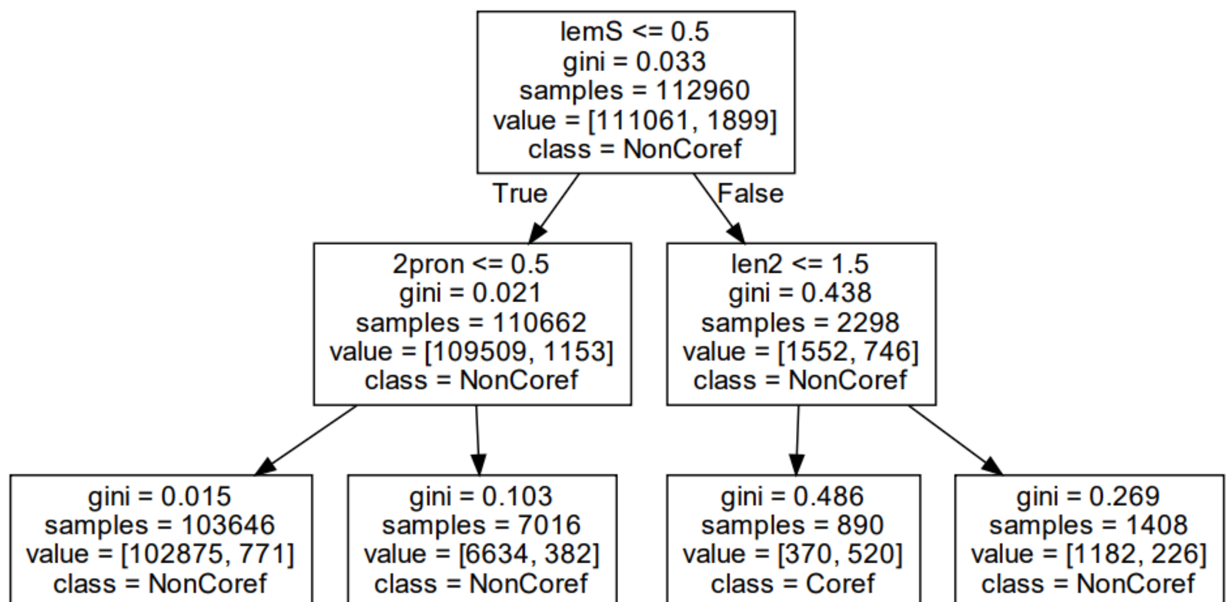

Рис. 2.1. Дерево рішень

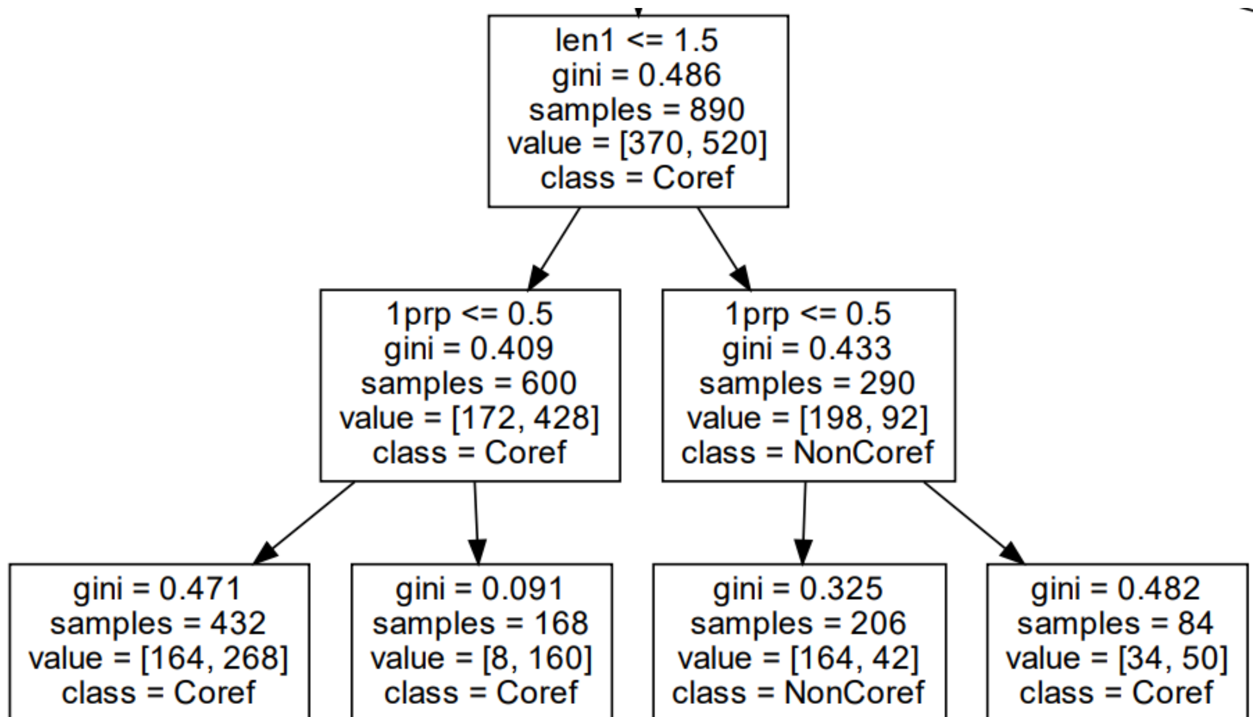

Рис. 2.2. Піддерево дерева рішень

2.1.4 Використання дерева рішень для пошуку кореферентних об'єктів

Після формування дерева рішень створене дерево використовується для отримання кластерів кореферентних об'єктів: відбувається перебір пар об'єктів – кандидатів на кореферентність (2.1). Для кожної пари об'єктів визначаються параметри (2.3), необхідні для класифікації об'єктів деревом рішень. Якщо пара класифікована як кореферентна, відбувається злиття об'єктів. Після утворення кластерів, що містять декілька об'єктів їх злиття відбувається, якщо хоча б одна пара об'єктів із першого та другого кластеру розпізнана кореферентною. В ході експериментальних досліджень виявлено, що при використанні декількох циклів проходів поки можливе злиття кластерів результати на метриках (наступний пункт) виявлялися на 2-5% вищими, ніж без використання циклічних проходів, тому в фінальній версії алгоритму використовується декілька проходів, допоки в циклі відбувається хоча б одне злиття. В результаті роботи алгоритму утворюється список кластерів, що містить кореферентні об'єкти всередині спільних кластерів (2.5).

2.2Формальна верифікація властивостей моделей пошуку кореферентних об'єктів, що використовують дерева рішень

В процесі створення великих, складних програмних систем часто виникають помилки. Тому актуальним є питання забезпечення надійності таких систем. Зазвичай для пошуку помилок в процесі розробки програмних продуктів виконується його тестування. Воно дозволяє знаходити помилки в програмі, але не гарантує їх виявлення. Для детальнішого дослідження надійності застосунків використовують методи формальної верифікації системи.

Для проведення формальної верифікації необхідно побудувати формальну модель системи [15]. Подати систему формально можна використовуючи розмічені транзиційні системи [50]. Вони дозволяють описати систему набором дискретних станів, між якими існують переходи, що здійснюються при виконанні певних умов та позначають операції, які виконуються системою під час переходу між станами.

Наступним кроком після створення формальної моделі необхідно визначити властивості системи, які будуть аналізуватися та також подати їх формально.

Після цього, отримавши модель системи та властивості для її перевірки, використовуються формальні методи доведення виконання (чи невиконання) властивостей. Для цього використовують автомати Бюхі [51].

Як показано в оглядовій статті [52], в існуючих дослідженнях використання формальної верифікації для моделей машинного навчання використовуються апарат розв'язності формул у теоріях (Satisfiability Modulo Theories, SMT) та лінійної оптимізації (Linear Programming, LP). В роботі використано підхід до верифікації з використанням автоматних моделей та лінійно-темпоральної логіки, що, на відміну від SMT, дозволяє досліджувати темпоральні характеристики моделі на потенційно

нескінченній послідовності станів. Як правило, математична модель дискретної системи являє собою граф, вершини якого відповідають станам (або класам станів), в яких може перебувати система в різні моменти часу; а ребрам – переходи між станами, які можуть мати позначки, що зображують дії чи події, які виконує система. Функціонування системи при такому зображенні представляється послідовностями переходів від одного стану до іншого. Якщо ребро має позначку, то ця позначка являє собою дію системи, що виконується при переході від стану на початку ребра до стану в його кінці. Далі в розділі використовуються розмічені транзиційні системи (розмічені ТС, або РТС) як дискретна модель обчислень загального типу [50].

2.2.1 Створення розмічених транзиційних моделей на високому рівні абстракції

З точки зору моделювання застосунок, що використовує дерева рішень для пошуку кореферентних об’єктів, можна представити у вигляді взаємодії таких систем:

- ТС 1, або “керуюча” система: відповідає за взаємодію з зовнішніми ресурсами;
- ТС 2, або “ядро”: система що представляє прохід по дереву рішень.

Керуюча система моделюється ТС

$$M = (\{v_0, v_1, v_2\}, \{a_1, a_2, a_3\}, \alpha, \beta, v_0), \tag{2.6}$$

де v0 - система в стані доступності; v1 - стан, в якому система опрацьовує вхідні дані; v2 - стан, в якому система видає результат. Переходи інтерпретуються наступним чином: a1 - отримання нового набору вхідних даних; a2 - формування результату класифікації; a3 - перехід в стан доступності.

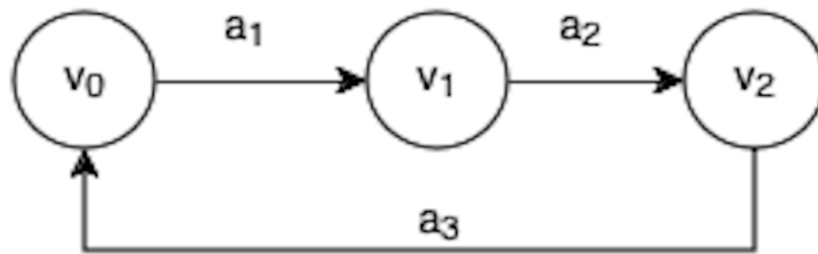

Рис. 2.3. Представлення ТС моделі керуючої системи

Для наочності, використаємо як ядро піддерево дерева рішень, отриманого у [12]. Варто зазначити, що в процесі створення моделі до піддерева було додано 2 сурогатні стани: початковий s0 та кінцевий s6; а також перехід t11 між ними. Це необхідно для представлення дерева рішень як неперервно функціонуючої системи, що дозволяє використання темпоральної логіки для подальшого аналізу. Кінцева модель визначається як

$$S_K = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}, \tag{2.7}$$

$$T_K = \{t_1, t_2, \dots, t_{11}\}, \tag{2.8}$$

$$K = (S_K, T_K, \alpha, \beta, s_0), \tag{2.9}$$

а множина пропозиційних формул, асоційованих зі станами, та функція позначок для кожного стану наведені на рис. 2.4.

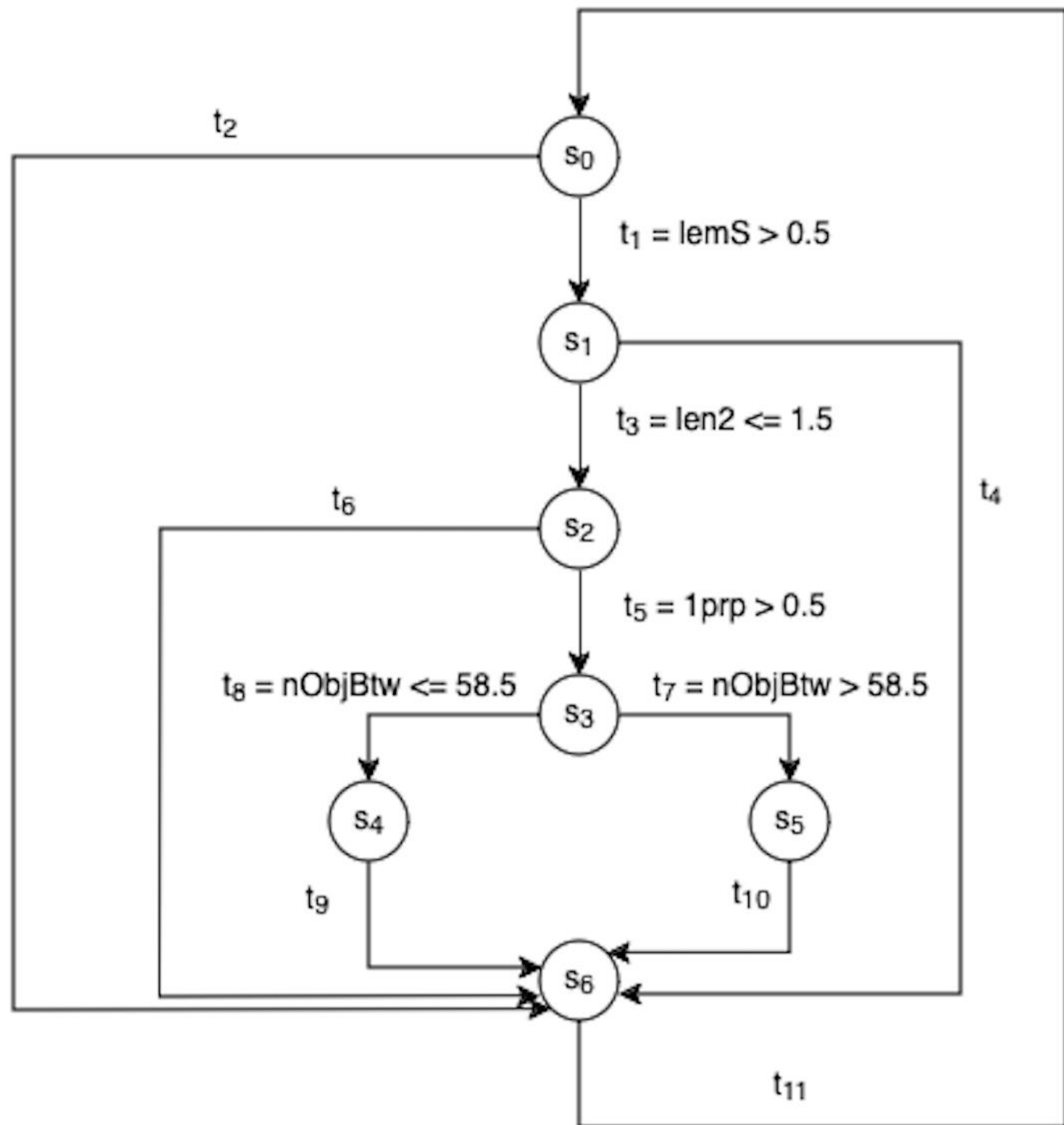

Рис. 2.4. Представлення ТС підмоделі ядра

Важливо зазначити, що у обраному для моделювання піддереві кожен із підмножини станів *{s2, s3, s4}* фіксує кореферентність вхідних даних. Відповідно, кожен зі станів *{s0, s1, s5}* фіксує відсутність кореферентності, а стан s_6 зберігає клас кореферентності, визначений раніше.

Для вищенаведених ТС будуємо синхронізовану паралельну композицію (синхронний добуток) з глобальними переходами, що моделює роботу застосунку в цілому. Множина обмежень синхронізації містить наступні елементи (ε - тотожна дія, що означає відсутність переходу в ТС):

$$T = \{(a_1, t_1), (a_1, t_2), (\varepsilon, t_3), (\varepsilon, t_4),$$

$$(\varepsilon, t_5), (\varepsilon, t_6), (\varepsilon, t_7), (\varepsilon, t_8), (\varepsilon, t_9),$$

$$(\varepsilon, t_{10}), (a_2, \varepsilon), (a_3, t_{11})\}. \qquad (2.10)$$

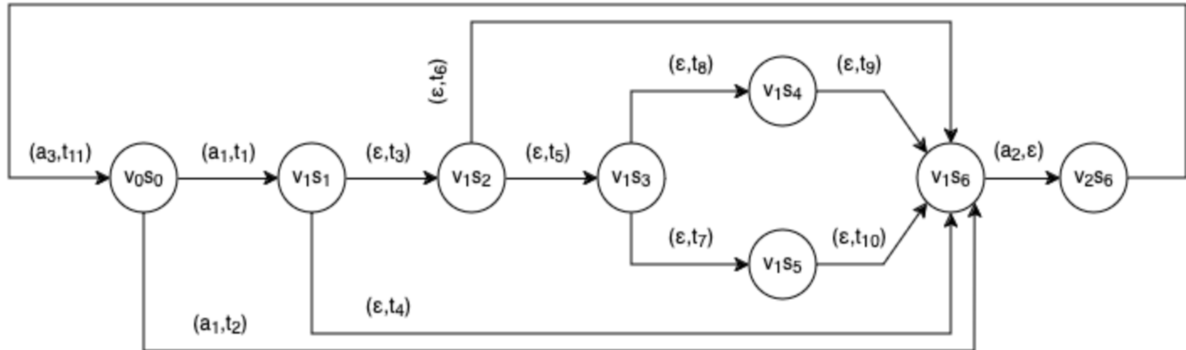

Рис. 2.5. Синхронний добуток ТС 1 і 2

2.2.2 Верифікація властивостей моделі автоматами Бюхі

Пропонується використання наступного алгоритму перевірки лінійно-темпоральної формули P, що представляє властивість, яка визначає семантичну коректність роботи системи:

Створити автомат Бюхі, що акцептує слова, які підтверджують P.

Побудувати добуток автомата і ТС, що моделює вихідну систему.

Знайти переріз шляхів, які генерує загальна транзиційна система (ЗТС), і шляхів, які акцептує автомат. Подальший аналіз досяжних станів перерізу дозволяє знайти як приклади, так і контрприклади формули P [51].

Розглянемо використання алгоритму на прикладі: нехай існує гіпотеза, що, якщо при аналізі кореферентності двох об’єктів довжина другого об’єкту є малою, то об’єкти кореферентні. Така властивість представляється формулою лінійно-темпоральної логіки

$$P_1 = (len2 \leq 1.5) \rightarrow G(F(coref)), \qquad (2.11)$$

тобто якщо умова справджується, то система рано чи пізно перейде в стан, що фіксує клас кореферентності, і залишиться в такому стані.

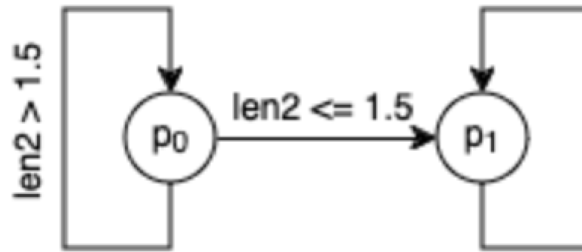

Рис. 2.6. Зображення автомата Бюхі, що відповідає LT формулі P1

Автомат Бюхі, що допускає слова, що відповідають формулі P_1, містить два стани: початковий p_0 та кінцевий p_1, перейшовши в який, автомат вже не змінює свій стан незалежно від поданих слів.

Побудуємо перетин автомата Бюхі (рис. 2.6) та ТС, що моделює синхронний добуток ТС 1 і 2 (рис. 2.5). Спрощене візуальне представлення отриманого перетину наведено на рис. 2.7. Для наочності на рис. 2.7 приховані недосяжні стани.

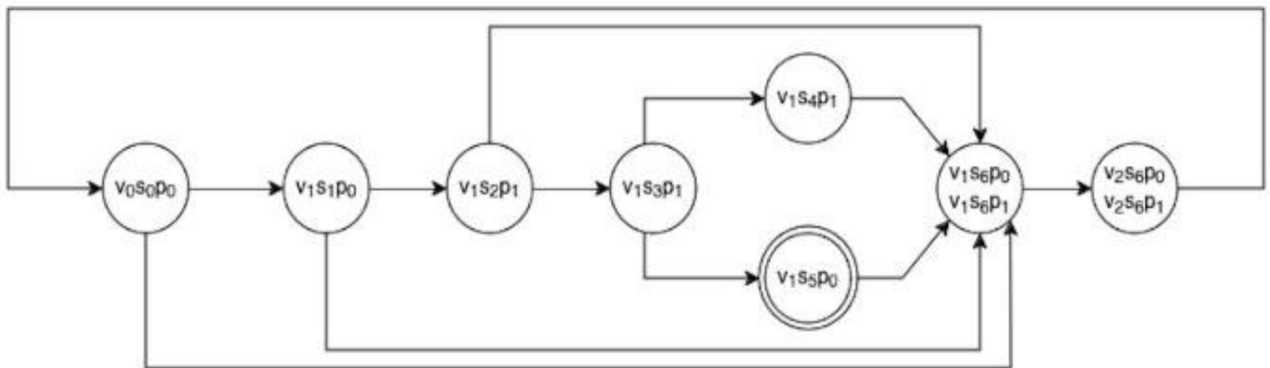

Рис. 2.7 Добуток автомата Бюхі та синхронного добутку ТС

Отриманий перетин дозволяє провести низку аналітичних дослідів можливих шляхів, циклів і трас. Зокрема, він дозволяє стверджувати, що можливий шлях, при якому відбувається перехід із стану p1 в стан p0 (кінцевий стан такого переходу виділено двома концентричними колами), тобто знайдено контрприклад властивості P. Дійсно, лише інформації про довжину одного з об’єктів мало, щоб стверджувати про їх кореферентність: обхід дерева з вказаним обмеженням може завершитися як стані s4, що

фіксує кореферентність, так і в стані s5, що фіксує її відсутність. Якщо ж розглянути іншу формулу лінійно-темпоральної логіки, наприклад,

$$P_2 = (lemS \wedge \underline{1prp} \wedge len2 \leq 1.5 \wedge$$

$$\wedge\, len1 \ \leq \ 1.5 \ \wedge \ nObjBtw \leq 58.5) \rightarrow G(F(coref)), \quad (2.12)$$

то аналіз добутку автомата Бюхі та синхронного добутку ТС покаже відсутність шляху-циклу такий, що є доступним із початкового стану, та який включає в себе стан із множини недопустимих станів, тобто формула P_2 є істинною, а отже властивість, представлена нею, виконується завжди.

2.3Метод пошуку кореферентних об'єктів на основі згортково-рекурентних нейронних мереж з довгою та короткочасною пам'яттю

2.3.1 Архітектура згортково-рекурентних нейронних мереж з довгою та короткочасною пам'яттю

Згортково-рекурентні нейронні мережі поєднують властивості згорткових нейронних мереж та рекурентних нейронних мереж, що дозволяє ефективно обробляти дані з просторово-часовими залежностями, такі як відео або виділяти значущу інформацію та аналізувати вхідні дані, що необхідно при аналізі текстів [18].

На початку, до вхідних даних застосовуються згорткові шари. Згорткові нейронні мережі складаються з шарів згортки, які застосовують фільтри (ядра) до вхідних даних, виявляючи такі особливості, як контури, текстури або інші закономірності. Після виконання операції згортки над вхідними даними та згортковими ядрами отримується ряд карт характеристик. Ваги ядер згортки визначаються автоматично в процесі навчання, використовуючи алгоритм зворотного розповсюдження помилки.

Наступним йде шар активації. Він представляє нелінійну функцію, що застосовується до карт характеристик для уможливлення апроксимації нелінійної функції нейронною мережею.

Шар підвибірки дозволяє зменшити розмірність вихідних даних шляхом вибору максимального значення з певної області вхідних даних. Цей шар дозволяє значно зменшити об'єм обчислень, відсікаючи несуттєві значення.

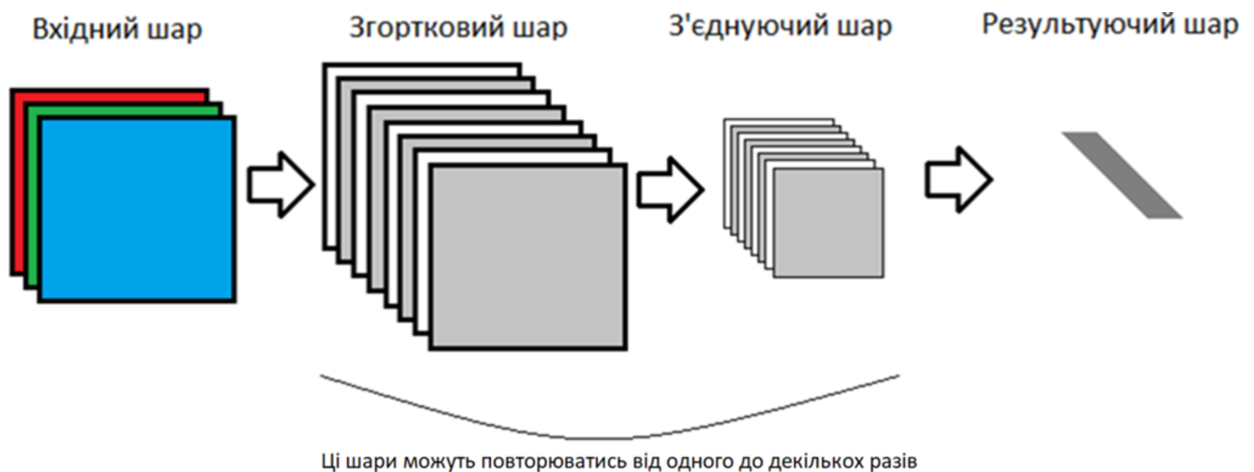

Рис. 2.8 – архітектура згорткової нейронної мережі

Рекурентні нейронні мережі з довгою та короткочасною пам'яттю (LSTM) призначені для обробки послідовних даних, де поточний стан залежить від попередніх.

Для обробки послідовностей даних із збереженням контексту використовуються рекурентні шари типу LSTM. Їхня ключова особливість — LSTM-комірка, керована трьома вентилями: забування (forget gate), вхідним (input gate) та вихідним (output gate). Стан комірки (cell state) акумулює та передає інформацію впродовж ланцюжка послідовностей.

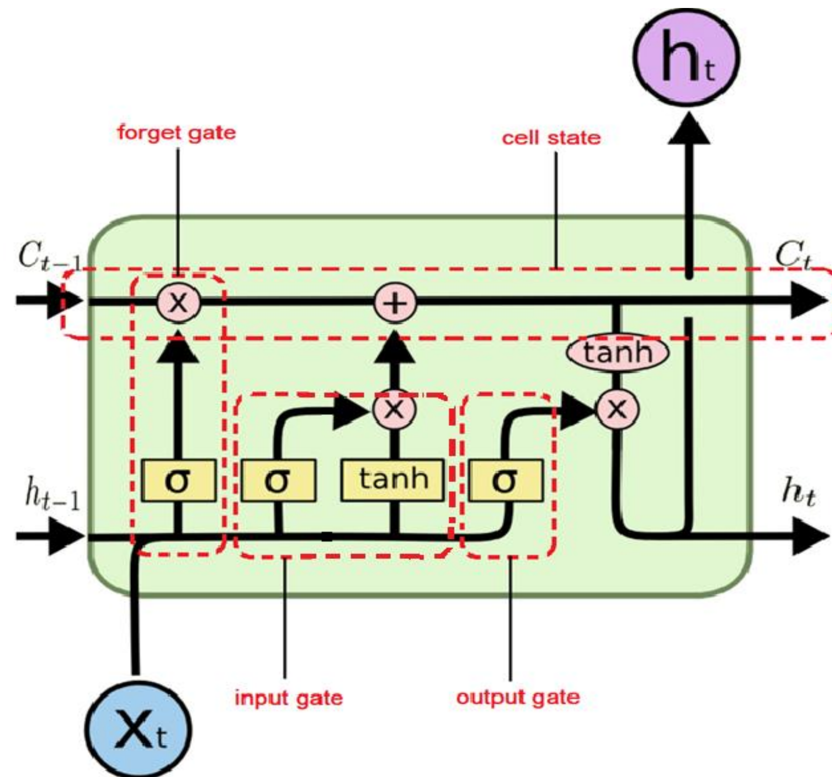

Рис. 2.9 – окрема комірка мережі з довгою та корткочасною пам'яттю 53]

Робота комірки відбувається наступним чином:

Керування забуванням: Вентиль забування аналізує попередній вихід (прихований стан) та поточний вхід, щоб визначити, які частини збереженої в стані комірки інформації слід відкинути.

Керування оновленням: Вхідний вентиль дозволяє додавати нову інформацію до стану комірки. Це двохетапний процес: визначається, що оновити (сигмоїдна функція) та які нові значення додати (функція tanh), після чого стан комірки оновлюється.

Керування виходом: Вихідний вентиль визначає, що саме з оновленого стану комірки сформує наступний прихований стан (вихід поточного кроку). Він формує вихідний стан комірки на основі попереднього прихованого стану та поточного входу.

Архітектура згортково-рекурентних нейронних мереж з довгою та короткочасною пам'яттю поєднує ці два підходи. Спочатку застосовуються згорткові шари для виділення значущих ознак із вхідних даних. Отримані ознаки передаються до рекурентних шарів з довгою та короткочасною пам'яттю, які моделюють часові залежності між послідовними входами.

Основними перевагами згортково-рекурентних нейронних мереж з довгою та короткочасною пам'яттю при роботі з текстами є зменшення кількості параметрів, необхідних для навчання мережі за рахунок використання згорткових шарів та як наслідок, аналіз тільки значущої для розв'язання задачі інформації шарами з довгою та короткочасною пам'яттю.

2.3.2 Підготовка текстів для аналізу згортково-рекурентною мережею з довгою та короткочасною пам'яттю

Для навчання та оцінки якості роботи алгоритму пошуку кореферентних об'єктів, використано набір текстів №1, описаний в пункті 1.3.

Першим етапом є отримання векторних представлень слів ELMo, які дозволяють враховувати порядок слів та контекст, в якому вони вжиті (пункт 1.6). Отримані в результаті використання бібліотеки вектори містять 1024 дійсних значення. До них додаються ознаки WordOrder, PartOfSpeech, IsPlural, IsProperName, IsHeadWord, Gender з корпусу текстів №1, які будуть доцільними для пошуку кореферентних об'єктів, та досліджені в статтях [12, 13, 14].

Також, для аналізу пари кореферентних об'єктів необхідно вказати, які саме об'єкти зі списку слів, що утворюють речення, аналізуються на наявність кореферентного зв'язку. Для цього використовується додатковий параметр – мітка пари, що приймає значення 1 для слів, що входять потенційно кореферентні об'єкти, які перевіряються на кореферентність, для всіх інших об'єктів значення мітки встановлене в 0.

2.3.3 Реалізація методу пошуку кореферентних об'єктів на основі згортково-рекурентних нейронних мереж з довгою та короткочасною пам'яттю

Для ідентифікації кореферентних пар об'єктів застосовується бінарний класифікатор, який визначає з певною ймовірністю, чи є пара кореферентною. Якщо ймовірність перевищує 0.5, пара класифікується як кореферентна, в іншому випадку — як некореферентна.

Бінарна класифікація реалізована на основі згортково-рекурентної нейронної мережі з довгою та короткочасною пам'яттю [18]. Модель ConvLSTM дозволяє працювати з текстами довільної довжини, враховуючи як локальні особливості слів, так і контекстуальні залежності. Архітектура мережі детально описана в розділі 2.3.1, її структура зображена на рис. 2.10.

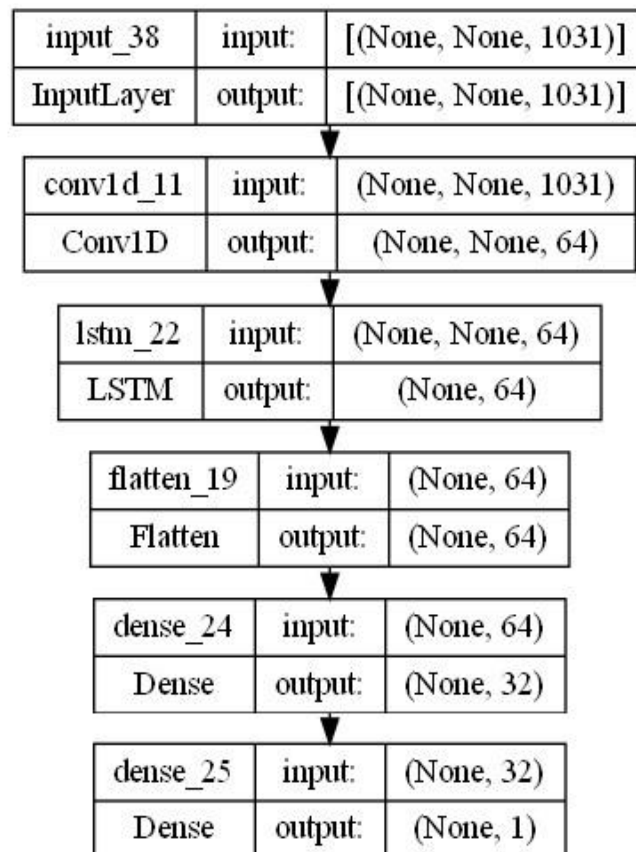

Рис. 2.10 – структура згортково-рекурентної нейронної мережі з довгою та короткочасною пам'яттю

Нейронна мережа складається з 6 шарів. *Вхідний шар* приймає на вхід текст з довільною кількістю слів, представлених у векторній формі. Кожне слово представлено вектором довжиною 1031 значень: 1024 значення - векторне представлення, отримане в результаті застосування бібліотеки ELMo, 6 значень – характеристики, отримані в результаті використання бібліотеки UDPipe та мітка пари, що позначає пару об’єктів, які перевіряються на кореферентність.

Згортковий шар Conv1D виділяє значущі характеристики з векторів ознак слів, зменшуючи розмірність вихідного вектора до 64. Це підвищує швидкість навчання мережі та точність класифікації за рахунок аналізу тільки значущої для розв’язання задачі інформації.

Рекурентний шар LSTM дозволяє аналізувати довгострокові залежності в послідовності , отриманих від згорткового шару. Це підвищує точність визначення кореферентних пар завдяки врахуванню контексту. Крім того, LSTM дозволяє обробляти тексти з довільною кількістю слів та нормувати вихідні дані, що зменшує їхню розмірність для наступного шару.

Шар Flatten перетворює дані в одновимірний вектор, придатний для аналізу повнозв'язними шарами.

Кожен нейрон *повнозв’язного шару* отримує інформацію від усіх нейронів попереднього шару та застосовує функцію активації для перетворення вхідних значень.

Вихідний шар складається з одного нейрона, який формує результат розв'язання задачі бінарної класифікації, і показує, чи кореферентною є пара об’єктів, позначених міткою пари.

Наступним етапом у пошуку кореферентних об'єктів у текстах є процес кластеризації. Він передбачає об'єднання знайдених кореферентних об'єктів в кластери на основі рішення бінарного класифікатора. Кожен новий кластер порівнюється лише з попередніми кластерами; якщо злиття не відбувається,

алгоритм переходить до наступного кластеру. Якщо нейронна мережа класифікує вхідні потенційно кореферентні об’єкти як кореферентні, відбувається їхнє злиття. Якщо пара не кореферентна, злиття не відбувається.

Для кластерів, що містять більше одного кореферентного об'єкту, злиття відбувається на основі аналізу першого об'єкту попереднього кластеру та одиничного потенційно кореферентного об'єкта кластеру, що розглядається.

2.4 Висновки до розділу 2

В розділі 2 дисертаційного дослідження представлено розробку та детальний опис двох методів для вирішення задачі пошуку кореферентних об'єктів в україномовних текстах: перший метод – на основі дерев рішень, другий – на основі згортково-рекурентних нейронних мереж з довгою та короткочасною пам’яттю. В розділі висвітлено теоретичні засади та практичні аспекти реалізації запропонованих підходів.

Розроблено метод на основі дерев рішень:

Визначення ознак: обрано лінгвістичні характеристики (9 ознак), що використовуються для опису потенційно кореферентних пар об'єктів. Ці характеристики включають семантичну схожість (ELMo), відстань, граматичні категорії (рід, число), тип іменника (власна назва, займенник), збіг лем тощо. Наголошено на автоматизованому отриманні характеристик за допомогою бібліотек ELMo та UDpipe.

Підготовка даних: описано процес перетворення даних з корпусів текстів у структуровані формати (списки об'єктів, кластерів, векторів ознак та міток), придатні для створення та тестування моделі дерева рішень. Задача кластеризації зведена до бінарної класифікації пар.

Реалізація та налаштування: використанно бібліотеку Scikit-learn для побудови дерева рішень, обґрунтовано необхідність обмеження складності

дерева (для уникнення перенавчання) через параметр min_impurity_decrease. Проілюстровано візуалізацію дерева та логіку його роботи на прикладі піддерева, що демонструє важливість певних ознак (наприклад, збіг лематизованих форм слова).

Алгоритм кластеризації: деталізовано процес використання навченого дерева рішень для формування кластерів кореферентних об'єктів шляхом ітеративного перебору пар, класифікації та злиття.

Проведено формальну верифікація методу пошуку кореферентних об'єктів з використанням дерев рішень:

Обґрунтування: підкреслено актуальність забезпечення надійності моделей машинного навчання, що виходить за межі традиційного тестування.

Запропоновано: використання апарату формальної верифікації на основі розмічених транзиційних систем (РТС), лінійно-темпоральної логіки та автоматів Бюхі для аналізу властивостей моделі дерев рішень. Цей підхід дозволяє досліджувати темпоральні характеристики та семантичну коректність роботи моделі.

Моделювання та верифікація: продемонстровано побудову формальної моделі системи та її верифікацію на прикладі двох формул лінійно-темпоральної логіки. Показано, як метод дозволяє виявляти контрприклади для невірних гіпотез та підтверджувати істинність коректних властивостей. Це дозволяє забезпечувати надійність розробленого методу.

Розроблено метод на основі згортково-рекурентної нейронної мережі з довгою та короткочасною пам'яттю:

Архітектура: описано гібридну архітектуру, що поєднує переваги згорткових шарів (CNN) для виділення локальних ознак та зменшення розмірності, та рекурентних шарів LSTM для моделювання довгострокових залежностей та контексту в послідовностях даних (текстах).

Підготовка даних: Описано використання векторних представлень ELMo, доповнених лінгвістичними ознаками та спеціальною міткою потенційно кореферентної пари для вказання аналізованих об'єктів.

Реалізація: Представлено 6-шарову архітектуру мережі що реалізує бінарну класифікацію пар об'єктів на кореферентність.

Алгоритм кластеризації: Описано процес формування кластерів на основі виходу нейромережевого класифікатора із зазначенням особливостей злиття кластерів.

РОЗДІЛ 3. СТВОРЕННЯ МЕТОДІВ ПОШУКУ КОРЕФЕРЕНТНИХ ОБ'ЄКТІВ В УКРАЇНОМОВНИХ ТЕКСТАХ З ВИКОРИСТАННЯМ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ

3.1Архітектура моделей природних мов Transformers

Модель Transformer, запропонована у статті[8], стала провідною моделлю в галузі обробки природної мови (NLP), замінивши рекурентні та згорткові нейронні мережі як основний підхід. Її архітектура, заснована на механізмі уваги (attention mechanism), який дозволяє враховувати значення всіх попередніх слів та визначати найбільш значущі з них для формування наступного слова у вихідній послідовності, обробляючи ці залежності паралельно, що значно прискорює навчання.

Архітектура Transformer складається з двох основних блоків: *кодера (encoder)* та *декодера (decoder)*. Кожен блок складається з декількох шарів, організованих послідовно.

Блок кодера: обробляє вхідну послідовність (наприклад, речення) та перетворює її в контекстуалізоване представлення.

Декодер: використовує контекстуалізоване представлення, отримане від кодера, для генерування вихідної послідовності (наприклад, переклад, відповідь на запитання).

Архітектура Transformer використовується для створення алгоритмів в різних галузях – обробка природної мови, фото, відео, аудіо інформації, створення інтерфейсів людина-комп'ютер. Transformer для створення великих мовних моделей (LLM) — це тип моделі штучного інтелекту, заснований на глибокому навчанні, що тренується на величезних обсягах текстових даних для розуміння та генерації природної мови. Ці моделі зазвичай мають мільярди параметрів (змінних, значення яких визначаються під час тренування і які визначають поведінку моделі) та здатні виконувати широкий спектр завдань з обробки природної мови (NLP), таких як переклад, узагальнення, відповіді на запитання, генерація коду та творчого тексту.

Графічне представлення архітектури Transformers показано на рис. 3.1.

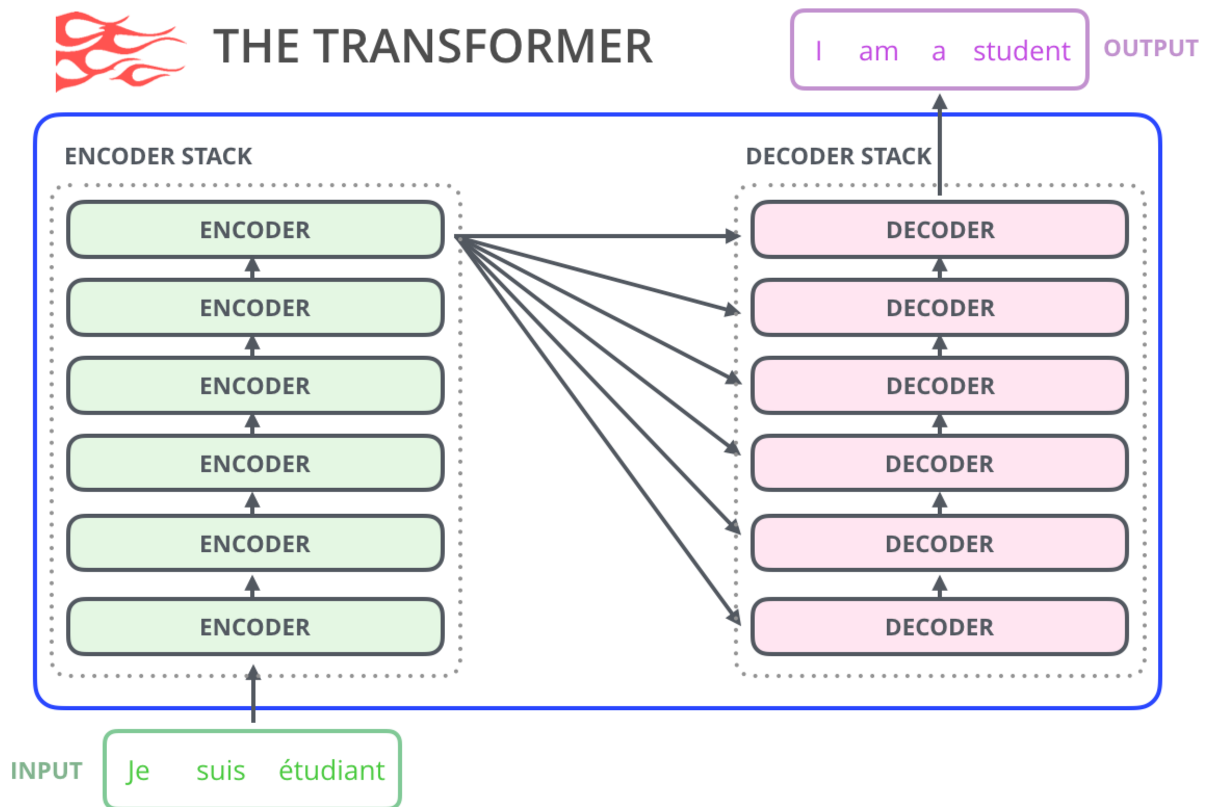

Рис. 3.1 Представлення мовної моделі на архітектурі Transformers[54]

До великих мовних моделей належать GPT3 [55], GPT4 [56], які уможливлюють генерацію текстів, коду програм і можуть вести розмову з людиною. Наразі використання таких моделей набуло широкого розповсюдження та продовжує розвиватися. В поширених іноземних мовах мовні моделі використовуються для пошуку кореферентних об'єктів (детальніше в розділі 1.5.4), але в українській мові їх використання досліджено в недостатній мірі.

3.2 Метод пошуку кореферентних об'єктів з використанням мовної моделі RoBERTa

Мовна модель RoBERTa [57] реалізована з використанням архітектури Transformers. В роботі застосована попередньо навчена мовна модель [58]

містить в собі 125 мільйонів параметрів та навчена на корпусі, що містить 2.6 мільярда слів (85.8 мільйонів рядків). Для навчання цієї моделі не вимагається розмічений набір даних, тому були використані великі набори текстів зібрані з україноомовної Вікіпедії, соціальних мереж та набору текстів OSCAR[59]. На вхід алгоритму подається текст T, що складається з набору речень:

$$T = \{s1, s2 \dots, sn\} \quad (3.1)$$

Кожне речення s складається з набору слів та умовних знаків (3.2):

$$s = \{w1, w2 \dots, wk\} \quad (3.2)$$

Окремі слова чи словосполучення формують потенційно кореферентні об’єкти на основі належності до певних частин мови та співпадіння їх роду. Однією з функцій бібліотеки Transformers є оцінка імовірності знаходження

конкретного слова чи словосполучення в обраному місці у тексті. Ця функція використовується для пошуку кореферентних об’єктів в тексті. Виділивши потенційно кореферентні об’єкти, визначається імовірність підстановки обраного референтну замість катафори чи анафори. Для цього спочатку обрана катафора чи анафора замінюється на спеціальний тег <mask>, що показує місце відповідної заміни. Далі проводиться токенізація – окремі слова, їх складові та знаки замінюються на спеціальні токени, необхідні алгоритму для подальшої роботи.

$$s = \{w1, w2 \dots < mask >.., wk\} \rightarrow \{t1, t2 \dots, tz\} \quad (3.3)$$

Токенізований текст подається в функцію pipe бібліотеки Transformers, в якій вказується як параметр target потенційно кореферентний об’єкт. Функція повертає числове значення P∈[0,1] - імовірність знаходження даного референту на місці катафори або анафори. Референтом до обраної катафори чи анафори буде вважатися об’єкт, імовірність знаходження якого в позиції обраної катафори чи анафори вища певного порогового значення $P > Pmin$

(для відсікання об'єктів, що не належать до жодного з кореферентних кластерів). Корпус для перевірки точності роботи алгоритму включає в себе 2500 текстів, з яких 2000 використовуються для перевірки точності роботи та ще 500 для оцінки порогового значення. Порогове значення $Pmin$ підбирається шляхом максимізації середнього арифметичного значень метрик (детальніше в розділі 4.1). Виходячи з отриманих імовірностей, потенційно кореферентні об'єкти об'єднуються в кластери, що є результатом роботи алгоритму.

3.3 Розгляд великих мовних моделей, використаних для розробки застосунку для пошуку кореферентних об'єктів.

3.3.1 Особливості сучасних моделей природної мови

Хоча всі великі мовні моделі базуються на Transformers, вони містять модифікації для підвищення ефективності, продуктивності та можливостей:

Grouped-Query Attention (GQA) **:** удосконалений варіант механізму Multi-Head Attention (MHA). GQA групує кілька груп *запитів (query heads)*, які ділять спільні *ключі (keys)* та *значення (values)*, зменшуючи обчислювальні витрати під час генерації відповідей (inference) [60]. Використовується в *Llama 3* [61, 62] та *Gemma 3* [63, 64].

Rotary Position Embeddings (RoPE): Метод кодування позиційної інформації токенів шляхом "обертання" векторів слів, що дозволяє поєднувати переваги відносного та абсолютного кодування позиційної інформації про слова [65]. Такий підхід покращує швидкодію та якість обробки довгих текстових послідовностей. RoPE використовується в *Llama 3* [61, 62], *Gemma 3* [63, 64], та *DeepSeek R1* [66, 67] (через його удосконалений вид – YaRN [68]).

Mixture of Experts (MoE): Архітектурний підхід, де використовуються кілька "експертів" (менших повнозв'язних нейронних мереж) та шлюзову мережу (gating network) для динамічного вибору експертів для обробки

кожного токена [68, 69]. Це дозволяє збільшити кількість параметрів без пропорційного збільшення обчислень на етапі генерації відповідей (inference). Використовується в *DeepSeek R1* [67, 68].

Всі сучасні великі мовні моделі підтримують *квантизацію* (quantization) [70]– представлення ваг моделі з пониженою розмірністю, яка дозволяє знизити вимоги до об'єму пам'яті, необхідної для збереження ваг моделі (які для сучасних моделей можуть складати десятки та сотні гігабайт) та прискорити роботу великих мовних моделей.

3.3.2 Моделі сімейства Llama (Llama 3, 3.1, 3.2, 3.3)

Llama (Large Language Model Meta AI) — це сімейство великих мовних моделей (LLM), розроблених та випущених компанією Meta AI [61, 62].

Особливості роботи та архітектура: Моделі Llama 3 є продовженням сімейства Llama, побудовані на удосконаленій архітектурі Transformers (детальніше пункт 3.3.4). Вони використовують GQA для ефективності генерації відповідей (inference) та RoPE для позиційного кодування. Модель Llama 3 з 8 та 70 мільярдами параметрів (8B та 70B) стала доступною у квітні 2024, Llama 3.1 (8B, 70B, 405B) - у липні 2024, Llama 3.2 (1B, 3B, 11B, 90B) – у вересні 2024, Llama 3.3 70B - у грудні 2024. Llama 3.3 показує покращені результати у міркуванні та математиці, наближаючись до Llama 3.1 405B [61, 62]. Llama 3 має контекстне вікно (кількість попередніх токенів, які велика мовна модель здатна враховувати для генерації наступного токена в послідовності) в 8 192 токени, Llama 3.1, 3.2, 3.3 – 131 072 токени.

Моделі сімейства Llama 3 є відкритими (open source). Ваги моделей доступні для завантаження. Модель можна донавчати та запускати локально.

Модель Llama 3 першочергово адаптована для роботи з англійською, але має підтримку української мови.

3.3.3 Модель DeepSeek R1

DeepSeek R1 – велика мовна модель, розроблена китайською компанією DeepSeek AI [66, 67].

Особливості роботи та архітектура: DeepSeek R1 випущена у січні 2025 та позиціонується як відкрита модель із здібністю до міркування, подібними до першої моделі з підтримкою міркування OpenAI o1 [71]. Вона базується на DeepSeek-V3 [72]. Ця модель має 671 мілярд параметрів та архітектуру *MoE*, де лише близько 37 мільярд параметрів активні під час генерації відповідей (inference). Використовує покращену версію RoPE - *YaRN* для розширення контекстного вікна до 131 072 токени. Також використовує *Multi-head Latent Attention (MLA)* [73] замість стандартного механізму Multi-Head Attention в перших шарах для зменшення розміру *KV-кешу (Key-Value cache)* (*KV-кеш* — це механізм кешування проведених обчислень, що зберігає обчислені значення *ключів (keys)* та *значень (values)* для попередніх токенів, щоб уникнути їх повторного обчислення при генерації кожного наступного токена, що значно прискорює процес генерації відповідей (inference), особливо для довгих послідовностей). Для навчання міркуванню на початку навчання цiє моделі використовувалося навчання з підкріпленням (reinforcement learning), а назавершаючому етапі – донавчання з вчителем (supervised fine-tuning).

Дистильовані версії моделі DeepSeek R1 створені шляхом дистиляції знань з R1 у менші моделі (1.5B-70B) на базі архітектур Qwen [74] та Llama 3 [61, 62]. Це робить передові можливості моделей з міркуванням доступнішими широкому колу користувачів.

DeepSeek R1 та дистильовані версії є відкритими (open source). Ваги доступні, можливе донавчання та локальний запуск.

Модель DeepSeek R1 першочергово адаптована для роботи з китайською та англійською мовами, але здатна працювати з українською.

3.3.4 Модель Gemma 3

Gemma 3 – велика мовна модель, розроблена підрозділом компанії Google – DeepMind [62, 63].

Особливості роботи та архітектура: Gemma — це сімейство відкритих моделей від Google. Gemma 3 випущена в березні 2025 та доступна у розмірах 1B, 4B, 12B, 27B. Заявлено, що вона побудована на тих же дослідженнях, що й Gemini 2.0 [63]. Gemma 3 має покращені можливості міркування у порівнянні з моделлю Gemma 2, підтримує мультимодальність - обробка тексту, зображень, коротких відео для моделей розміром >4B, контекстне вікно 131 072 токени,. Gemma 3 підтримує структурований вивід в заданому форматі та виклик функцій для реалізації агентних систем [75] (систем, що використовують складне планування та міркування для автоматизованого вирішення складних задач) та роботи з додатковими засобами, наприклад, доступу в мережу інтернет та обробки документів. Google заявляє, що Gemma 3 27B є найкращою моделлю для запуску на одному прискорювачі та перевершує Llama 3 405B та DeepSeek-V3 на LMArena [76].

Моделі сімейства Gemma є відкритими під ліцензією Google, що дозволяє їх комерційне використання та дослідження. Ваги моделі представлені в відкритому доступі, моделі можна донавчати та запускати локально.

Модель Gemma 3 фокусується на підтримці понад 140 мов, включаючи українську.

3.3.5 Моделі Gemini 2.0 Flash та Gemini 2.0 Thinking

Моделі Gemini 2.0 Flash [77] та Gemini 2.0 Thinking [78] розроблені підрозділом компанії Google - DeepMind.

Gemini 2.0 Flash: мультимодальна модель, що підтримує обробку текстів, зображень, аудіо та відео, а також генерацію зображень і аудіо. У порівнянні з попреденіми моделями, Gemini 2.0 Flash перевершує 1.5 Pro за якістю, будучи швидшою за 1.5 Flash [77]. Підтримує агентивні можливості.

Gemini 2.0 Flash Thinking: спеціалізована на міркуванні модель, що підтримує обробку текстів та зображень. На відмінну від інших закритих моделей з підтримкою міркування, таких як OpenAI o1 [71] та o3 [79], Gemini 2.0 Flash Thinking дозволяє переглядати процес міркування *("chain of thought")*, що уможливлює аналіз процесу прийняття рішень моделлю.

Моделі Gemini 2.0 є закритими (closed source). Доступні безкоштовно через API при обмеженні на кількість запитів (до 15 в хвилину). Не доступні для завантаження/локального запуску. Використання регулюється умовами Google Cloud. Обидві моделі підтримують довжину контексту в 1 048 576 токенів. Моделі Gemini мають вбудовану багатомовну підтримку, включаючи українську.

3.3.6 Порівняння великих мовних моделей, обраних для розв'язання задачі пошуку кореферентних об'єктів

Розглянуті моделі демонструють значний прогрес в LLM. Llama 3, DeepSeek R1 та Gemma 3 є найбільш поширеними відкритими моделями, пропонуючи якісні, доступні для завантаження та донавчання моделі. Усі вони використовують RoPE (або його варіанти) для кращої обробки довгих послідовностей, а DeepSeek R1 виділяється архітектурою MoE та використанням MLA для зменшення розмірів KV-кешу. На відмінну від попередніх відкритих моделей, Gemma 3 має підтримку більше ніж 140 мов та вбудовані можливості структурованого виводу. Gemini 2.0, хоч і є закритою моделлю, доступна через API обмежено безкоштовно, та є передовою у мультимодальності та агентних можливостях. Здатність LLM генерувати структурований вивід є важливою для інтеграції в реальні

системи, і ця функція все активніше впроваджується як у відкритих (Gemma 3), так і закритих (Gemini 2.0) моделях. Порівняння великих мовних моделей, використаних в роботі, наведено в таблиці 3.1

Таблиця 3.1 – порівняння великих мовних моделей, використаних в роботі

Характеристика\Модель	**Llama 3 (3.1, 3.2, 3.3)**	**DeepSeek R1**	**Gemma 3**	**Gemini 2.0 (Flash, Thinking)**
Розробник	Meta AI	DeepSeek AI	Google DeepMind	Google DeepMind
Дата випуску	Квітень-Грудень 2024	Січень 2025	Березень 2025	Грудень 2024
Розмір (Параметри)	Llama 3 8B, 70B Llama 3.1 8B, 70B, 405B Llama 3.2 1B, 3B, 11B, 90B Llama 3.3 70B	671B (всього), 37B (активні), дистильовані версії: 1.5B-70B	1B, 4B, 12B, 27B	Не розголошується
Вимоги до пам'яті (RAM, VRAM), квантизація q4, довжина	1B – 1.5ГБ, 3B – 3ГБ, 8B – 6ГБ, 11B – 8ГБ, 70B – 48ГБ,	671B – 512ГБ, дистильовані: 1.5B – 2ГБ, 7B – 6ГБ, 8B –	1B – 1.5ГБ, 4B – 4ГБ, 12B – 10ГБ, 27B – 16ГБ	Не розголошуються

контексту – 4096 токенів	90B – 64ГБ, 405B – 256ГБ	6ГБ, 32B – 20ГБ 70B – 48ГБ		
Архітектура	Transformers, GQA, RoPE	Transformers, MoE, MLA, RoPE (YaRN)	Transformers, GQA, RoPE, GeGLU	Transformers
Відкритість	Відкрита	Відкрита	Відкрита	Закрита (доступ через API), безкоштовне використання при обмеженій кількості запитів
Можливість завантаження ваг	Так	Так	Так	Ні
Підтримка донавчання	Так	Так	Так	Обмежена (через Vertex AI)
Мождивість запуску на локальному комп’ютері	Так	Так	Так	Ні

Контекстне вікно	Llama 3 – 8192 токени Llama 3.1, 3.2, 3.3 - 131072 токени	131072 токени	131072 токени	1048576 токенів
Ключові особливості	Відкритість, навчання на великому наборі текстів, підтримка міркування (3.3)	Відкритість, Міркування, MoE, наявність дистильованих версій	Відкритість, міркування, мультимодальність структурований вивід	Мультимодальність, Міркування (Gemini 2.0 Thinking), структурований вивід
Українська мова	Обмежена підтримка	Обмежена підтримка	Навчена на 140 мовах, включаючи українську	Підтримується
Наукові дослідження	Дозволено	Дозволено	Дозволено	Дозволено (через API)
Структурований вивід	Обмежена підтримка (через запит)	Обмежена підтримка (через запит)	Так (Function Calling / JSON)	Так (Function Calling / JSON)

3.4Розробка методів пошуку кореферентних об'єктів на основі великих мовних моделей

3.4.1 Загальна архітектура застосунку пошуку кореферентних об'єктів на основі великих мовних моделей

В роботі для пошуку кореферентних об'єктів використано підхід Entity-Mention (розділ 1.4). Цей підхід описує знаходження кореферентних об'єктів через створення кластерів, в яких всі об'єкти посилаються на один і той самий реальний чи уявний об'єкт. В результаті застосування цього підходу, утворюється список кластерів, які включають в себе кореферентні об'єкти. Реалізовано два методи розв'язання задачі пошуку кореферентних об'єктів з використанням великих мовних моделей на основі Entity-Mention.

- Бінарна класифікація об'єктів на етапі генерації результатів мовною моделлю.
- Генерація кластеру об'єктів, кореферентних до обраного об'єкту як результат обробки тексту мовною моделлю. Для вирішення задачі пошуку кореферентних об'єктів використані моделі мов, представлені в розділі 3.3. Вирішення задачі пошуку кореферентних об'єктів для окремого тексту включає в себе наступні кроки:

1. Підготовка запиту до мовної моделі, що включає в себе чітко зформульоване завдання та текст з корпусу текстів. Для двох представлених варіантів пошуку кореферентних об'єктів запити до мовної моделі відрізняються: для варіанту з бінарною класифікацією до вхідного тексту додається два потенційно кореферентні об'єкти. Для методу отримання кластеру кореферентних об'єктів на етапі генерації, до вхідного запиту додається єдиний об'єкт та пронумерований список потенційно кореферентних об'єктів до обраного (детальніше в розділах 3.4.2 та 3.4.3).

2. Обробка вхідного запиту великою мовною моделлю, генерація відповіді.
3. Розпізнавання та інтерпретація відповіді мовної моделі алгоритмом. Цей етап відрізняється для двох представлених варіантів пошуку кореферентних об’єктів.
4. Після розпізнавання одиночної відповіді відбувається перехід до кроку 1, аж допоки весь текст не буде оброблено.
5. Результатом обробки тексту є отримання кластерів кореферентних об’єктів для всього тексту.

Під час роботи з великими мовними моделями особливу увагу варто приділяти підготовці запиту до моделі [80], що може значно впливати на якість згенерованої відповіді.

3.4.2 Метод пошуку кореферентних об’єктів з використанням бінарної класифікації на етапі генерації результатів мовною моделлю

Підготовка запиту до мовної моделі: на вхід мовній моделі подається завдання, зформульоване англійською мовою, де коротко описана задача пошуку кореферентних об’єктів та формат виводу, який нейронна мережа має використовувати (Таблиця 3.1).

Завдання сформульоване англійською мовою, так як на ній проводиться основна частина навчання великих моделей. Як наслідок, мовні моделі здатні краще інтерпретувати англомовні завдання.

Для виводу результату встановлено формат

$$boxed\{True|False\} \tag{3.1}$$

, оскільки деякі мовні моделі, зокрема моделі з міркуванням, такі як DeepSeek R1 та Gemini 2.0 Thinking передбачають генерацію міркувань перед остаточною відповіддю, тому відповідь потрібно відокремлювати від міркувань.

Після завдання, на вхід нейронній мережі подається текст T, що складається з послідовності слів та розділових знаків:

$$T = \{w_0, w_1, \dots, w_n\} \tag{3.2}$$

Початок слова показаний міткою «Text:» для спрощення розділення вхідної інформації.

Також, на вхід мовній моделі подаються цільові потенційно кореферентні об’єкти, відмічені мітками «First mention:» та «second mention:». Так як всередині одного тексту можуть повторюватися іменники, займенники та словосполучення, що вказують на різні об’єкти в залежності від контексту, до цільових кореферентних об’єктів додано нумерацію в дужках, в форматі [n], де n – порядковий номер слова чи знаку, починаючи від нуля при підрахунку від початку тексту.

Таблиця 3.2 – приклад запиту до великої мовної моделі для розв'язання задачі пошуку корефеорентних об'єктів з використанням методу на основі бінарної класифікації

Завдання	You are a task-specific expert that performs Coreference Resolution of Ukrainian Language texts. This task means you have to check if two chosen mentions from the text are coreferent or not (to check if they refer, correspond to the same real or imaginary object in a given context). You will receive a Ukrainian-language text. Additionally, you will receive a pair of mentions from this text, which you can exactly identify by their numeration. As a result, you should output [boxed{True}] or [boxed{False}], which marks the pair of received mentions as coreferent or not coreferent.

Текст	Text: На даний момент це може бути важко уявити, але багато людей вірять, що до 2100 року ми будемо жити на планеті Марс. Наша власна планета, Земля, стає дедалі більш перенаселеною та забрудненою. На щастя, ми можемо почати знову і побудувати кращий світ на Марсі…
Цільові об’єкти	First mention: року[24], second mention: Наша[26] власна[27] планета[28]

Після обробки тексту мовною моделлю, отримана відповідь інтерпретується - для boxed{True} пара об’єктів вважається кореферентною. При отриманні boxed{False} чи відповіді, в якій неможливо знайти результат в правильному форматі, пара вважається некореферентною.

При роботі з моделями мови звертати особливу увагу на забезпечення вирішення задачі за скінченний час, оскільки мовним моделям властиво генерувати відповіді, що можуть не відповідати жорстко заданому формату, а також генерувати нескінченні послідовності токенів. Саме тому використана обробка відповідей, наведена в попередньому абзаці, а в функції, що відповідає за запит до мовної моделі (client.chat.completions.create) вказано параметр max_tokens=32, що обмежує довжину згенерованої послідовності (для моделей з міркуванням встановлювалося значення max_tokens=2048).

3.4.3 Метод створення кластеру кореферентних об’єктів на етапі генерації результатів мовною моделлю

При підготовці запиту до мовної моделі для генерації кластера на вхід моделі подається завдання, аналогічно до пункту, 3.5.2, але зформульоване для задачі кластеризації потенційно кореферентних об'єктів (Таблиця 3.2).

Для виводу результату кластеризації встановлено формат:

$$boxed\{[w_n[n]\ w_{n+1}[n+1] \dots] \mid [w_k[k] \dots] \mid \dots\} \tag{3.3}$$

де w – слова чи розділові знаки, n – порядковий номер слова чи розділового знаку починаючи з 0, що зазначається в квадратних дужках одразу після слова чи знаку. Вираз $[w_n[n]\ w_{n+1}[n+1] \ldots]$ позначає один кореферентний об’єкт, що складається з декількох слів та знаків.

Після завдання, на вхід нейронній мережі подається текст T, так само позначений міткою «Text:», як у попередньому методі.

Також, на вхід мовній моделі подається обраний потенційно кореферентний об’єкт, що відмічений міткою «First mention:», та список потенційно кореферентних до нього об’єктів - «Potentially coreferent mentions:». До всіх потенційно кореферентних об'єктів додано нумерацію в дужках, в форматі [n], де n – порядковий номер слова чи знаку, починаючи від нуля при підрахунку від початку тексту.

Таблиця 3.3 – приклад запиту до великої мовної моделі для розв'язання задачі пошуку корефеорентних об'єктів з використанням методу на основі кластеризації

Завдання	You are a task-specific expert that performs Coreference Resolution of Ukrainian Language texts. This task means you have to find all mentions from the text that are coreferent to the chosen one (to check if they refer to the same real or imaginary object in a given context). You will receive a Ukrainian-language text. Additionally, you will receive the chosen first mention and a list of potentially coreferent mentions from this text, which you can exactly identify by their numeration. As a result, you should output a cluster of coreferent mentions to the chosen one, for example, if mentions [мій[11] тато[12]] and [він[354]] are coreferent, you should output: boxed{мій[11] тато[12]] | [він[354]}. This list should include first mentions in the first place + other mentions only from the list of potentially

	coreferent mentions.
Текст	Text: На даний момент це може бути важко уявити, але багато людей вірять, що до 2100 року ми будемо жити на планеті Марс. Наша власна планета, Земля, стає дедалі більш перенаселеною та забрудненою. На щастя, ми можемо почати знову і побудувати кращий світ на Марсі…
Цільові об’єкти	First mention: багато[10] людей[11] \n Potentially coreferent mentions: 2100[16] | 2100[16] року[17] | року[17] | ми[18] | планеті[22] | планеті[22] Марс[23] | Марс[23] | Наша[25] | Наша[25] власна[26] планета[27] | Наша[25] власна[26] планета[27] ,[28] Земля[29] | Земля[29] | ми[41] | Марсі[50]…

Відповідь, згенерована мовною моделлю інтерпретується та у відповідності до заданого формату отримується кластер кореферентних об'єктів. При отриманні відповіді, в якій неможливо знайти результат в правильному форматі, вхідний об’єкт, позначений як «First mention:» вважається окремим кластером.

Так само, як у розділі 3.5.2, необхідно звертати особливу увагу на забезпечення вирішення задачі за скінченний час. Тому використана обробка відповідей, яка повинна забезпечити виконання алгоритму за скінченний час, а в функції client.chat.completions.create вказано параметр max_tokens=2048, що обмежує довжину згенерованої послідовності (для моделей з міркуванням встановлювалося значення max_tokens=8192). Значення параметру max_tokens значно більші, ніж в попередньому розділі, оскільки на вихід велика мовна модель має повертати не результат одиничної класифікації, а цілий кластер об’єктів.

3.5 Донавчання великої мовної моделі Llama 3.2

Для покращення ефективності методу пошуку кореферентних об’єктів в україномовних текстах на основі великої мовної моделі прийнято рішення донавчити мовну модель. Для цього, обрано мовну модель Llama 3.2 [81], зокрема, квантизовану версію (q4) з 3 мільярдами параметрів. Вибір моделі пояснюється високою швидкодією даного варіанту моделі у порівнянні з більшими мовними моделями та можливістю запускати даний варіант моделі на більшому різновиді пристроїв, оскільки для її запуску достатньо 3 ГБ оперативної пам’яті.

Для донавчання моделі використано бібліотеки PyTorch, Transformers та Unsloth (детальніше в розділі 1.6) та алгоритм QLoRA [82], що уможливлює донавчання моделі на графічному процесорі з 8 ГБ відеопам’яті за рахунок корекції тільки частини ваг в процесі навчання та використання квантизації.

Використано підхід до донавчання на основі метод створення кластеру кореферентних об’єктів на етапі генерації результатів мовною моделлю. Для цього сформовано набір зразків для донавчання мовної моделі. Як вхідний запит до мовної моделі, використано запит з таблиці 3.3. В якості послідовності токенів, що мають бути передбачені нейронною мережею, використано очікуваний правильний вихід, що позначає кластер кореферентних об'єктів для заданого тексту та цільових об'єктів.

Корпус текстів містить 36600 текстів для навчання. В середньому, кожен текст містить по 10 кластерів кореферентних об'єктів. Таким чином, можна утворити 370000 зразків, що можуть бути використані для донавчання моделі мови. В ході роботи було виявлено проблеми з обробкою такої кількості зразків через нестачу оперативної пам'яті. Також, донавчання на такій кількості прикладів зайняло б близько 6 днів на системі з таблиці 4.2.

Було прийнято рішення виділити із кожного тексту по 2 зразки, що були використані для донавчання моделі. Таким чином, модель використовувала всі тексти для донавчання, а саме воно зайняло 28 годин на одному поколінні, в результаті моделлю було проаналізовано 73000 прикладів. Процес донавчання моделі показаний на рис. 3.2.

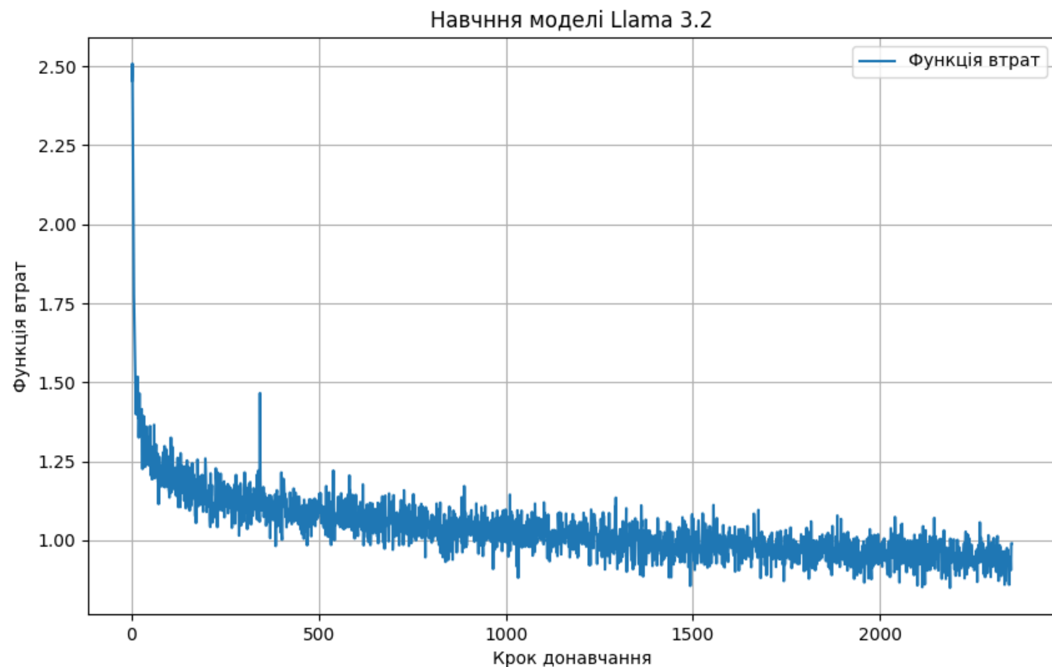

Рис. 3.2 – Графік залежності функції втрат від кроку навчання

Для донавчання моделі задано гіперпараметр rank =128, який визначає, яка частка параметрів використовується для донавчання. Для моделі Llama 3.1 3B, кількість параметрів що донавчалися – 194.5 мільйони для заданого значення rank. Таке значення параметру встановлено для отримання максимально можливої якості донавченої моделі без помітного збільшення часу донавчання.

Також, встановлено параметрів per_device_train_batch_size = 2 та gradient_accumulation_steps = 10, що обумовлює значення параметру batch size як добуток цих параметрів та дорівнює 20. Параметр batch size показує,

яку кількість зразків опрацьовує мовна модель, доки її ваги не будуть оновлені на основі результатів опрацювання зразків.

Активація параметру packing дозволила прискорити навчання моделі на 10% на пробному запуску без деградації якості отриманої моделі. Packing дозволяє моделі в деяких випадках об’єднувати запити для обробки, що особливо ефективно для навчанні на коротких текстах. Як результат застосування, кількість прикладів для донавчання зменшилася до 47000, а саме навчання зайняло 2350 кроків.

3.6 Висновки до розділу 3

Третій розділ дисертаційного дослідження присвячений розробці та дослідженню методів пошуку кореферентних об'єктів в україномовних текстах із застосуванням сучасних великих мовних моделей (LLM). Цей розділ є продовженням попередніх, та фокусується на застосуванні передових технологій обробки природної мови, заснованих на архітектурі Transformer для пошуку кореферентних об'єктів.

Представлено огляд архітектури Transformer, яка є основою для сучасних великих мовних моделей, пояснено ключові компоненти (кодер, декодер) та механізм уваги.

Описано початковий метод на основі моделі RoBERTa, адаптованої для української мови, що використовує підхід маскування для оцінки ймовірності кореферентності пари об'єктів.

Проведено аналіз новітніх великих мовних моделей, обраних для проведення дослідження методів пошуку кореферентних моделей (сімейства Llama 3, DeepSeek R1, Gemma 3, Gemini 2.0).

Згадано ключові архітектурні інновації, що підвищують їх ефективність та можливості великих мовних моделей (Grouped-Query

Attention, Rotary Position Embeddings, Mixture of Experts, Multi-head Latent Attention, квантизація).

Наведено порівняльну характеристику обраних моделей за рядом параметрів: розмір, вимоги до пам'яті, архітектурні особливості, відкритість, можливість донавчання та локального запуску, розмір контекстного вікна, підтримка української мови, здатність до структурованого виводу та міркування. Цей аналіз обґрунтовує вибір моделей для подальшої розробки методів.

Запропоновано загальну архітектуру застосунку для пошуку кореферентних об'єктів на основі LLM, що використовує підхід Entity-Mention.

Розроблено два методи взаємодії з LLM через зформовані запити (prompts):

Метод 1: Бінарна класифікація пар об'єктів: мовна модель отримує текст та два потенційно кореферентні об'єкти (з індексами) і повертає рішення (True/False) у специфічному форматі.

Метод 2: Генерація кластера об'єктів: LLM отримує текст, один цільовий об'єкт та список потенційних кандидатів, і генерує весь кластер кореферентних об'єктів у заданому структурованому форматі.

Окреслено практичні аспекти реалізації: формулювання завдань англійською мовою для кращої інтерпретації моделями, використання індексів для точної ідентифікації об'єктів, специфікація форматів виводу та обробка відповідей, включаючи обмеження довжини генерації (max_tokens) для забезпечення скінченності процесу.

Здійснено донавчання (fine-tuning) обраної відкритої моделі Llama 3.2 3B для задачі пошуку кореферентних об'єктів в україномовних текстах,

використовуючи метод створення кластерів на етапі генерації результатів мовною моделлю.

Обґрунтовано вибір моделі (баланс продуктивності та ресурсів),описано підготовку даних (формування зразків з корпусу), налаштування гіперпараметрів та процес навчання.

РОЗДІЛ 4. ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ МЕТОДІВ ОЦІНКИ КОРЕФЕРЕНТНОСТІ В УКРАЇНОМОВНИХ ТЕКСТАХ

4.1Дослідження прискорення роботи моделей пошуку кореферентних об’єктів з використанням графічних прискорювачів

У зв’язку з великими обчислювальними потужностями, необхідними для роботи великих мовних моделей, для їх практичного застосування доцільно використовувати графічні прискорювачі або інші спеціалізовані мікросхеми. Такий підхід дозволяє:

1. Досягти значно вищої швидкості обробки текстів великими мовними моделями.
2. Досягти вищої енергоефективності в обробці великих мовних моделей. Порівняння енергоефективності процесорів загального призначення, графічних процесорів та спеціалізованих мікросхем наведене в пункті 1.7 дисертаційного дослідження.

Таблиця 4.1 – Характеристики системи для випробування моделей пошуку кореферентних об’єктів

Тип	Модель	Специфікації
Процесор	AMD Ryzen 3900X	12 ядер 24 потоки Частота 4.2 ГГц 64МБ кешу L3 1.6 TFlops FP32
Оперативна пам’ять	DDR4-3600	32 ГБ 57.6 GB/s

Графічний прискорювач 1	AMD Radeon RX 6900 XT	5120 потокових процесорів Частота 2.6 GHz 16GB GDDR6 512 GB/s 128MB Infinity Cache 26 TFlops FP32 52 Tflops FP16 104 TOPS INT8 208 TOPS INT4
Графічний прискорювач 2	AMD Radeon RX 6800 XT	4608 потокових процесорів Частота 2.36 GHz 16GB GDDR6 512 GB/s 128MB Infinity Cache 21.75 TFlops FP32 43.5 Tflops FP16 87 TOPS INT8 174 TOPS INT4
Операційна система	Windows 11 Pro	Версія 23H2

Експериментальне дослідження прискорення роботи великих мовних моделей проводилося на апаратній конфігурації, зазначеній в таблиці 4.1. (використовувався лише графічний прискорювач 1). Результати порівняння середнього часу генерації одиночного токена для моделі Llama 3 8B наведені на рис. 4.1.

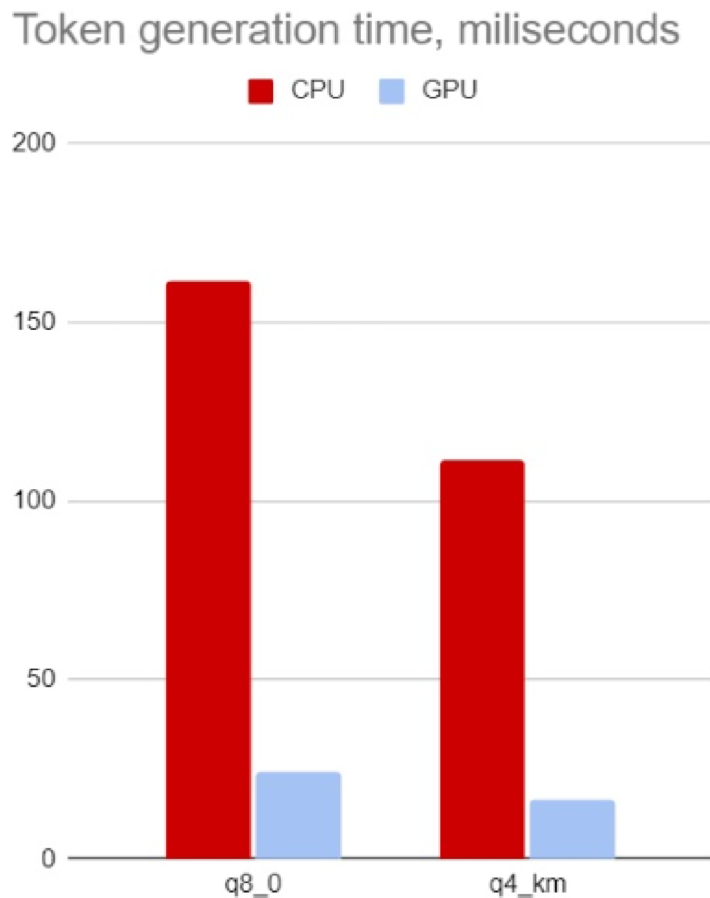

Рис. 4.1 – порівняння середнього часу генерації одиничного токена

В результаті використання графічного процесора, отримано приріст швидкодії в 6.7 разів при обробці запитів мовною моделлю з 8-бітною квантизацією та в 6.8 разів з 4-бітною. Прискорення від використання 4-бітної квантизації склало 45% на CPU та 47% на GPU.

На відмінну від інших архітектур нейронних мереж, які часто можуть повністю поміщатися в кеш-пам’яті графічного процесора чи CPU, швидкодія великих мовних моделей майже завжди обмежена швидкістю

доступу до динамічної пам’яті, оскільки великі мовні моделі містять в собі мільярди параметрів. Наявність високочастотної пам’яті з широкою шиною обміну даними зумовлює значний відрив графічних процесорів від CPU.

Сучасні архітектури мовних моделей підтримують розділення шарів для розподілу параметрів на декілька графічних прискорювачів (layer parallelism), що дозволяє поміщати всі ваги моделі в пам’ять графічних прискорювачів, що необхідно для повноцінного прискорення. Також, великі мовні моделі можуть використовувати паралелізм на рівні тензорів (tensor parallelism), який хоч і вимагає вищої швидкості обміну даними між GPU, але дозволяє реалізувати паралельну роботу декількох GPU, що підвищує швидкодію.

Для уможливлення проведення досліджень з великими мовними моделями, що не поміщаються в пам'ять одиничної GPU, застосовується паралелізм з розділенням шарів на 2 графічних прискорювача.

Донавчання великої мовної моделі проведене на конфігурації з таблиці 4.2. Навчання великих мовних моделей майже завжди проводиться на GPU чи спеціалізованих прискорювачах через значно вищу швидкість виконання навчання та адаптованості бібліотек для навчання до роботи в першу чергу на GPU.

Таблиця 4.2 – Характеристики системи, що використовувалася для донавчання великої мовної моделі Llama 3.2

Тип	Модель	Специфікації
Процесор	AMD Ryzen 5800H	8 ядер 16 потоків Частота 4 ГГц 16МБ кешу L3 1TFlops FP32

Оперативна пам’ять	DDR4-2666	32 ГБ 42.7 GB/s
Графічний прискорювач	Nvidia RTX 3070 Mobile	5120 ядер CUDA 160 тензорних ядер Частота 1,9 GHz 8GB GDDR6 448 GB/s 168 Tflops FP16
Операційна система	Windows 11 Pro	Версія 23H2

4.2Метрики оцінки ефективності алгоритмів

Для оцінки результатів, отриманих внаслідок використання методів пошуку кореферентних об’єктів використовуються метрики B-cubed [83] і MUC [84]. Ці метрики дозволяють порівнювати групи кластерів – із правильним упорядкуванням кореферентних об’єктів, та із передбачуваним упорядкуванням, чисельно відображаючи різницю між кластерами. Для кожної метрики обчислюється влучність - precision (відношення вірно обраних алгоритмом об’єктів до усіх обраних об’єктів), повнота - recall (відношення вірно обраних алгоритмом об’єктів до усіх об’єктів, що відносяться до даного кластеру), та F1 міра (середнє гармонійне повноти та влучності).

Метрика B-cubed використовується в широкому колі задач кластеризації. Ця метрика розглядає окремі елементи в списку передбачених кластерів, для яких розраховується інтегральний показник. Влучність для метрики B-cubed визначається як середнє арифметичне влучностей для кожного елемента:

$$P = \frac{1}{N}\sum_N \frac{Z_n}{T_n} \quad (4.1)$$

де n – номер обраного елемента, N – кількість елементів у списку, Z_n – кількість елементів, що належать до тої ж кореферентної групи, що й обраний елемент (включаючи обраний елемент) та входять в передбачуваний кластер, T_n – загальна кількість елементів в передбаченому кластері.

Повнота для метрики B-cubed визначається як середнє арифметичне повноти для кожного елемента:

$$R = \frac{1}{N}\sum_N \frac{Z_n}{M_n} \quad (4.2)$$

де n – номер обраного елемента, N – кількість елементів у списку, Z_n – кількість елементів, що належать до тої ж кореферентної групи, що й обраний елемент (включаючи обраний елемент) та входять в передбачуваний кластер, M_n – загальна кількість елементів в тій же групі, що й обраний елемент.

Метрика MUC спеціально розроблена для оцінки ефективності роботи алгоритмів, що розв'язують задач пошуку кореферентних об'єктів. В метриці MUC розглядається весь список кластерів, для якого розраховуються показники. Для MUC повнота визначається формулою:

$$R = \frac{\sum(|S_i| - |p(S_i)|)}{\sum |S_i|} \quad (4.3)$$

де $|S_i|$ –кількість елементів в істинному кореферентному кластері, $|p(S_i)|$- кількість підгруп, на які істинний кореферентний кластер ділять передбачувані кластери. Влучність визначається як:

$$P = \frac{\sum(|S_i'| - |p'(S_i')|)}{\sum|S_i'|} \quad (4.4)$$

де $|S_i'|$ –кількість елементів в передбачуваному кореферентному кластері, $|p'(S_i')|$ - кількість підгруп, на які передбачуваний кореферентний кластер ділять істинні кластери.

Для кожної метрики обчислюється F1 міра, що визначається формулою:

$$F_1 = \frac{2*R*P}{R+P} \quad (4.5)$$

де R – повнота, P – влучність.

4.3Дослідження використання дерев рішень, нейронних мереж та малих моделей природних мов для пошуку кореферентних об'єктів

Досліджено використання дерев рішень для пошуку кореферентних об'єктів [12, 13, 14]. Вони показують високі результати, найвищі для метрики B-cubed та наближаються до результатів моделі двозв'язної нейронної мережі з довгою та короткочасною пам'яттю (BiLSTM) [2] на метриці MUC, при цьому обганяють всі інші моделі на показнику влучності. Перевагою дерев рішень є інтерпретованість результатів, яка дозволяє аналізувати внутрішню логіку роботи алгоритму та найвища швидкодія у порівнянні з іншими алгоритмами через просте застосування сформульованих правил до вхідних текстів. Недоліком використання дерев рішень є необхідність екстракції додаткової інформації з текстів перед обробкою, що виконується бібліотекою UDpipe.

В роботі проведено дослідження ефективності використання моделі для пошуку кореферентних об'єктів на моделі природної мови RoBERTa [9, 10]. Отримані результати показують, що модель може використовуватися для пошуку кореферентних об'єктів та показує результати, близькі до згорткової нейронної мережі, хоча й відстає від інших моделей. Перевагою цієї моделі є відсутність необхідності навчання.

Досліджено використання моделі на основі згортково-рекурентних нейронних мереж з довгою та короткочасною пам’яттю [18]. Модель на основі такого типу нейронних мереж показала результати на рівні між згортковими нейронними мережами [3] та методом на основі BiLSTM [2].

Алгоритми, розглянуті в даному розділі, використовували корпус для пошуку кореферентних об’єктів №1, і тому результати їх роботи не можуть бути безпосередньо порівняними з результатами, представленими в наступному розділі, що використовують корпус №2, оскільки корпуси відрізняються середньою довжиною тексту, розміром та кількістю кластерів на один текст і складністю кореферентних зв’язків в текстах. Для прикладу наведені результати роботи методу кластеризації кореферентних об'єктів на етапі генерації результатів мовною моделлю на основі моделі Llama 3.2 (донавченої на корпусі №1). Результати, отримані в таблиці 4.3 значно вищі у порівнянні з результатами з таблиці 4.5, що відображає різницю між корпусами текстів.

Таблиця 4.3 – порівняння ефективності методів пошуку кореферентних об'єктів в україномовних текстах, досліджених на корпусі текстів №1 (результати, отримані в попередніх дослідженнях інших авторів позначені сірим кольором)

Метрика	***MUC***			***B-cubed***		
Модель	***Влучність***	***Повнота***	***F1 міра***	***Влучність***	***Повнота***	***F1 міра***
CNN[3]	*24.23*	*12.45*	*16.44*	*97.88*	*84.99*	*92.11*
BiLSTM [2] (multi-pass)	*56.36*	*39.68*	*45.88*	*93.13*	*90.43*	*91.76*
Дерево рішень	***73.46***	***29.08***	***41.67***	***98.15***	***88.14***	***92.87***
RoBERTa	***27.39***	***13.10***	***17.72***	***91.22***	***89.65***	***90.43***
ConvLSTM	***50.66***	***28.23***	***36.26***	***94.34***	***78.36***	***85.61***
Llama 3.2 3B fine-tuned	***92.20***	***74.76***	***82.57***	***93.38***	***77.20***	***84.52***

4.4Дослідження використання великих мовних моделей для пошуку кореферентних об’єктів.

Розв’язання задачі пошуку кореферентних об’єктів можна проводити, використовуючи великі мовні моделі. Для цього розроблено два методи для пошуку кореферентних об’єктів – на основі бінарної класифікації та кластеризації на етапі генерації результатів мовною моделлю.

Порівняння ефективності роботи мовних моделей з використанням методу бінарної класифікації приведено в таблиці 4.4 [19], в таблиці 4.5 приведено порівняння ефективності для методу кластеризації потенційно кореферентних об'єктів (результати вказано для квантизації q4, якщо не зазначено інакше). Великі моделі показують близькі результати при розв'язанні задачі пошуку кореферентних об'єктів обома методами. При цьому метод кластеризації на етапі генерації результатів мовною моделлю має велику перевагу – значно менша необхідна кількість запитів до мовної моделі для обробки одного тексту, що прискорює обробку текстів мовними моделями та уможливлює застосування мовних моделей Google Gemini, доступних безкоштовно, але з обмеженням на кількість запитів в хвилину.

Таблиця 4.4 – порівняння ефективності для методу пошуку кореферентних об’єктів з використанням бінарної класифікації на етапі генерації результатів мовною моделлю

Метрика	***MUC***			***B-cubed***		
Модель	***Влучність***	***Повнота***	***Модель***	***Влучність***	***Повнота***	***Модель***
Llama 3 8B zero-shot	***76.24***	***57.54***	***65.58***	***75.20***	***49.22***	***59.50***
Llama 3 8B q8 zero-shot	***75.10***	***67.53***	***71.11***	***62.43***	***58.09***	***60.18***
Llama 3 8B q8 multi-shot	***74.25***	***79.75***	***76.90***	***41.84***	***73.21***	***53.25***
Llama 3.2 3B	***73.93***	***38.44***	***50.58***	***85.94***	***34.68***	***49.42***
Llama 3.3 70B q3	***78.86***	***80.44***	***79.65***	***60.58***	***67.32***	***63.77***
Deepseek r1 – Qwen 7b	***51.81***	***54.00***	***52.88***	***38.89***	***41.59***	***40.19***
Gemma 3 27B	***79.60***	***79.77***	***79.68***	***65.07***	***67.29***	***66.17***
Gemma 3 4B	***69.63***	***66.22***	***67.88***	***56.89***	***54.38***	***55.61***

Таблиця 4.5 – порівняння ефективності для методу створення кластеру кореферентних об’єктів на етапі генерації результатів мовною моделлю

Метрика	***MUC***			***B-cubed***		
Модель	***Влучність***	***Повнота***	***Модель***	***Влучність***	***Повнота***	***Модель***
Llama 3.2 3B	***58.08***	***64.67***	***61.20***	***29.84***	***50.13***	***37.41***
Llama 3.3 70B q3	***72.87***	***60.89***	***66.34***	***74.51***	***45.81***	***56.74***
Deepseek r1 – Qwen 7b	***68.76***	***68.00***	***68.38***	***46.1***	***55.70***	***50.44***
Gemma 3 27B	***73.99***	***81.55***	***77.59***	***40.82***	***74.40***	***52.72***
Gemma 3 4B	***69.66***	***81.11***	***74.95***	***28.37***	***69.61***	***40.31***
Gemini 2.0 flash	***80.56***	***76.44***	***78.45***	***70.26***	***64.01***	***66.99***
Gemini 2.0 flash thinking	***84.85***	***87.11***	***85.96***	***71.52***	***81.54***	***76.20***
Llama 3.2 3B fine-tuned	***82.50***	***73.33***	***77.65***	***79.28***	***59.68***	***68.10***

Порівнюючи роботу моделей між собою, можна відзначити значний прогрес в розвитку відкритих мовних моделей та, зокрема, відзначити покращення в підтримці української мови в останніх мовних моделях. Так, модель Gemma 3 з 27 мільярдами параметрів показує себе на рівні моделі Llama 3.3 (при квантизації q3), що містить 70 мільярдів параметрів. Також, варто відмітити кращу продуктивність моделі Gemini 2.0 Flash Thinking, у порівнянні з моделлю зі схожою підтримкою природних мов Gemini 2.0 Flash, що свідчить про користь від використання ланцюжків міркувань для пошуку кореферентних об’єктів в текстах.

Метод кластеризації на етапі генерації результатів мовною моделлю було обрано за основу для донавчання моделі Llama 3.2. Отримана модель здатна швидко обробляти вхідні тексти, виділяючи кластери кореферентних об'єктів, а якість її роботи знаходиться на рівні закритої великої мовної моделі Gemini 2.0 flash. Завдяки компактності та швидкості роботи, така мовна модель може використовуватися для пошуку кореферентних об'єктів в пристроях з обмеженими обчислювальними ресурсами.

4.5 Висновки до розділу 4

Четвертий розділ дисертаційного дослідження присвячений експериментальному дослідженню та порівняльному аналізу ефективності розроблених методів пошуку кореферентних об'єктів в україномовних текстах.

З отриманих результатів, можна зробити наступні висновки:

1. Експериментальне дослідження використання графічних прискорювачів (GPU) для роботи з великими мовними моделями показали значний приріст швидкодії (в ~6.7-6.8 разів) порівняно з центральними процесорами (CPU). Це свідчить про важливість апаратного прискорення для практичного застосування великих мовних моделей та доцільність використання GPU-орієнтованих

підходів для покращення швидкодії та енергоефективності. Позитивний вплив квантизації на швидкодію (4-бітна квантизація швидша за 8-бітну на 45-47%) свідчить про обмеження швидкодії роботи моделі в першу чергу обмежену швидкість обміну обчислювального пристрою з пам'яттю, оскільки якби такого обмеження не було, швидкість роботи на CPU, що не підтримує роботи з 4-бітними числами, а, натомість, працює з ними як з 32-бітними не змінювалася б.

2. Порівняльний аналіз ефективності методів на основі дерев рішень, нейронних мереж (ConvLSTM) та моделі RoBERTa (на Корпусі №1) за метриками MUC та B-cubed дозволив кількісно оцінити їхню продуктивність. Для дерев рішень отримані високі результати в задачі пошуку кореферентних об’єктів, що в комбінації з їх іншими перевагами, а саме можливістю інтерпритувати результати та високою швидкодією робить їх одним із найкращих методів пошуку кореферентних об’єктів. Результати для ConvLSTM та RoBERTa вказують на їхню придатність для розв’язання задачі пошуку кореферентних об’єктів.
3. Дослідження ефективності двох розроблених методів взаємодії з великою мовною моделлю (бінарна класифікація та генерація кластерів на Корпусі №2) показало їхню співставну точність, однак метод генерації кластерів виявився значно ефективнішим за кількістю запитів, та, як наслідок, швидкодією, що свідчить про його перевагу для практичних застосувань.
4. Широке порівняння різних LLM (Llama 3, DeepSeek R1 distilled, Gemma 3, Gemini 2.0) виявило високу ефективність новітніх моделей, зокрема тих, що мають покращену підтримку української мови (Gemma 3) або спеціалізовані на міркуванні (Gemini 2.0 Thinking). Отримані результати свідчать про значний потенціал

великих мовних моделей для вирішення задачі пошуку кореферентних об’єктів.

5. Експериментальна перевірка донавченої моделі (Llama 3.2 3B fine-tuned) продемонструвала суттєве покращення якості порівняно з базовою моделлю, що дозволило досягнути якості кластеризації кореферентних об’єктів, співставного з пропрієтарними моделями. Це підтверджує доцільність та високу ефективність підходу з адаптацією (fine-tuning) відкритих LLM для специфічних завдань обробки української мови.

ВИСНОВКИ

На основі результатів, отриманих протягом дисертаційного дослідження, **можна зробити наступні висновки:**

1. *Розроблено метод пошуку кореферентних об'єктів в україномовних текстах на основі дерев рішень*, що використовує набір лінгвістичних ознак, отриманих за допомогою моделей векторних представлень слів (ELMo) та засобів морфологічного аналізу (UDpipe). Зведення задачі до бінарної класифікації пар потенційно кореферентних об'єктів дозволило досягти високої точності (зокрема, за метрикою B-cubed) та забезпечити інтерпретованість моделі шляхом візуалізації дерева. Додатково проведено формальну верифікацію властивостей розробленого методу з використанням розмічених транзиційних систем та автоматів Бюхі, що підтвердило його семантичну коректність та підвищило надійність.
2. *Запропоновано метод пошуку кореферентних об'єктів на основі згортково-рекурентної нейронної мережі з довгою та короткочасною пам'яттю (ConvLSTM).* Дана архітектура поєднує здатність згорткових шарів до виділення локальних ознак із векторних представлень слів та спроможність шарів LSTM моделювати довгострокові залежності в тексті, що є важливим для знаходження кореферентних зв'язків. Експериментально показано ефективність методу для задачі бінарної класифікації кореферентних пар в україномовних текстах, що позиціонує його як конкурентний підхід поряд з іншими нейромережевими архітектурами.
3. *Розроблено та досліджено два методи використання великих мовних моделей (LLM) для пошуку кореферентних об'єктів в україномовних текстах*: на основі бінарної класифікації пар об'єктів

за допомогою спеціально сформованого запиту (prompt) та на основі генерації повного кластеру кореферентних об'єктів безпосередньо моделлю у відповідь на запит. Встановлено, що метод генерації кластерів є значно ефективнішим за кількістю необхідних звернень до LLM для обробки одного тексту. Експериментально підтверджено високу ефективність сучасних LLM (зокрема, Gemma 3 та Gemini 2.0) для вирішення задачі кореференції в українській мові, причому моделі зі здатністю до міркування (Gemini 2.0 Thinking) демонструють найкращі результати.

4. *Здійснено успішне донавчання (fine-tuning) відкритої великої мовної моделі Llama 3.2* (3 мільярди параметрів) для специфічної задачі пошуку кореферентних об'єктів в україномовних текстах, використовуючи розроблений метод генерації кластерів та техніку QLoRA для ефективного навчання. Донавчена модель демонструє значне покращення якості розв'язання задачі порівняно з базовою моделлю "zero-shot" і досягає результатів, співставних з потужними пропрієтарними моделями (наприклад, Gemini 2.0 Flash). Це підтверджує високу ефективність та доцільність адаптації відкритих LLM для специфічних завдань обробки української мови.
5. *Досліджено ефективність використання високопродуктивних обчислень для прискорення роботи розроблених методів на основі великих мовних моделей*. Експериментально встановлено значний приріст швидкодії (у ~6.7-6.8 разів) при використанні графічних прискорювачів (GPU) порівняно з центральними процесорами (CPU) для етапу генерації відповідей (inference) LLM. Також підтверджено суттєві переваги квантизації (зниження бітності представлення ваг моделі, зокрема 4-бітної порівняно з 8-бітною) для зменшення вимог до пам'яті та подальшого прискорення обчислень (до ~47%). Отримані результати вказують на надзвичайно важливу роль апаратного прискорення та програмних технік покращення

швидкодії великих мовних моделей для їх практичного впровадження та використання у задачах обробки природної мови.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sukthanker, R., Poria, S., Cambria, E., & Thirunavukarasu, R. (2018). Anaphora and coreference resolution: A review. Information Fusion, 59, (pp. 139-162).
2. Telenyk, S., Pogorilyy, S., & Kramov, A. (2021). The complex method of coreferent clusters detection based on a BiLSTM neural network. Knowledge Based Systems, (pp. 205-210).
3. Погорілий, С. Д., & Крамов, А. А. (2019). Метод виявлення кореферентних пар в україномовному тексті з використанням згорткових нейронних мереж. Вісник університету "Україна". Серія : Інформатика, обчислювальна техніка та кібернетика, (2), (с. 259-268). http://nbuv.gov.ua/UJRN/Visunukr_inform_2019_2_26
4. Hobbs, J. R. (1978). Resolving pronoun references. Lingua, 44(4), (pp. 311–338).
5. McCarthy, J. F., & Lehnert, W. G. (1995). Using Decision Trees for Coreference Resolution. Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI ’95), (pp. 1060–1065).
6. Wiseman, S., Rush, A. M., & Shieber, S. M. (2016). Learning Global Features for Coreference Resolution. Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, (pp. 994–1004).
7. Lee, K., He, L., Lewis, M., & Zettlemoyer, L. (2017). End-to-end Neural Coreference Resolution. Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, (pp. 188-197).
8. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., & Polosukhin, I. (2017). Attention is all you need. In Proceedings of the 31st International Conference on Neural Information Processing Systems, (pp. 6000-6010).
9. Pogorilyy, S., & Biletskyi, P. (2022). Coreference resolution based on pretrained RoBERTa language model. In Proceedings of the 8th

International Conference on Control and Optimization with Industrial Applications, COIA 2022, (pp. 354–357). Baku, Azerbaijan. http://coia-conf.org/upload/editor/files/COIA2022_V1_abs.pdf

10. Погорілий, С. Д., & Білецький, П. В. (2021). Використання графічного процесора для прискорення пошуку кореферентних об'єктів з використанням моделі RoBERTa. Наукові праці Донецького національного технічного університету, Серія: Інформатика, кібернетика та обчислювальна техніка, 33(2), (pp. 4–9). https://doi.org/10.31474/1996-1588-2021-2-33-4-9.
11. Pogorilyy, S., & Biletskyi, P. (2018). Development and analysis of the tool for verification graphic isomorphism. Proceedings of the XVII International Young Scientists' Conference on Applied Physics, (pp. 108–109). https://indico.knu.ua/event/2/contributions/276/contribution.pdf
12. Погорілий, С., & Білецький, П. В. (2022). Алгоритм пошуку кореферентних об’єктів в україномовних текстах із використанням дерев рішень. Problems in programming, (3-4), (pp. 85–91). https://doi.org/10.15407/pp2022.03-04.085.
13. Pogorilyy, S., & Biletskyi, P. (2022). Coreference resolution algorithm for Ukrainian-language texts using decision trees. Proceeding of the UkrProg 2022 (pp. 81-90). https://ceur-ws.org/Vol-3501/s8.pdf
14. Pogorilyy, S., & Biletskyi, P. (2023). Analysis of Decision Trees for Coreference Resolution Task in Ukrainian Language. Proceedings of the Workshops at the X International Scientific Conference “Information Technology and Implementation” (IT&I-WS 2023) (CEUR Workshop Proceedings Vol. 3646, pp. 255–262). https://ceur-ws.org/Vol-3646/Short_7.pdf
15. Погорілий, С. Д., Слинько, М. С., & Білецький, П. В. (2024). Формальна верифікація властивостей моделі пошуку кореферентних об’єктів на основі дерев рішень. Problems in programming, (2-3), (pp. 319–328). https://doi.org/10.15407/pp2024.02-03.319.

16. Pogorilyy, S., Slynko, M., & Biletskyi, P. (2024). The verification of decision tree model for coreference resolution using marked transition systems, Petri nets and Büchi automata. In Proceedings of the 14th International Scientific and Practical Conference on Programming (UkrPROG 2024) (CEUR Workshop Proceedings Vol. 3806, pp. 361–371). https://ceur-ws.org/Vol-3806/S_26_Pogorilyy_Slynko_Biletskyi.pdf
17. Погорілий, С. Д., Крамов, А. А., & Білецький, П. В. (2019). Метод оцінки когерентності україномовних текстів з використанням згорткової нейронної мережі. Збірник Наукових Праць Військового Інституту Київського Національного Університету Імені Тараса Шевченка 65, (pp. 63–71). https://doi.org/10.17721/2519-481X/2019/65-08
18. Biletskyi, P., & Tkach Y. (2024). Coreference resolution in Ukrainian-language texts using convolutional long short-term memory neural networks. XXIV International young scientists conference on applied physics (pp. 129-130). https://icap.knu.ua/index.php/icap2024
19. Pogorilyy, S., & Biletskyi, P. (2024). Coreference Resolution in Ukrainian-language texts using Llama 3 large language model. In Proceedings of the 9th International Conference on Control and Optimization with Industrial Applications (COIA 2024) (pp. 538–542). Istanbul, Turkey. http://coia-conf.org/upload/editor/files/Proc_COIA2024.pdf
20. Бібліотека UDpipe. lindat.mff.cuni.cz. Retrieved 25 March 2025, from https://lindat.mff.cuni.cz/services/udpipe/.
21. Корпус текстів для навчання та тестування алгоритмів пошуку кореферентних об’єктів в українській мові. huggingface.co. Retrieved 25 March 2025, from https://huggingface.co/datasets/artemkramov/coreference-dataset-ua.
22. Clark, K., & Manning, C. D. (2016). Improving Coreference Resolution by Learning Entity-Level Distributed Representations. Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (pp. 643-653). https://doi.org/10.18653/v1/P16-1061.

23. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. Proceedings of the 26th International Conference on Neural Information Processing Systems, Vol. 2, (pp. 3111–3119).
24. WinoGrande Schema Challenge. winogrande.allenai.org. Retrieved 25 March 2025, from https://winogrande.allenai.org/.
25. Luo, L., Yang, C., Kharlamov, E., & Pan, S. (2024). Integrating Large Language Models and Knowledge Graphs for Next-level AGI. https://www.cs.emory.edu/~jyang71/files/klm-tutorial.pdf
26. Exploring the data landscapes of AGI: Knowledge graphs vs. relational databases. medium.com. Retrieved 25 March 2025, from https://medium.com/singularitynet/exploring-the-data-landscapes-of-agi-knowledge-graphs-vs-relational-databases-525ffe6d4221.
27. Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Vol. 1, (pp. 2227–2237).
28. Бібліотека Scikit-learn. scikit-learn.org. Retrieved 25 March 2025, from https://scikit-learn.org/.
29. Бібліотека Graphviz. graphviz.org. Retrieved 25 March 2025, from https://graphviz.org/.
30. Python JSON library. docs.python.org. Retrieved 25 March 2025, from https://docs.python.org/3/library/json.html.
31. HuggingFace Datasets library. huggingface.co. Retrieved 25 March 2025, from https://huggingface.co/docs/datasets/index.
32. NumPy library. numpy.org. Retrieved 25 March 2025, from https://numpy.org/.
33. Tensorflow library. tensorflow.org. Retrieved 25 March 2025, from https://www.tensorflow.org/.

34. PyTorch library. pytorch.org. Retrieved 25 March 2025, from https://pytorch.org/.
35. Бібліотека Unsloth. unsloth.ai. Retrieved 25 March 2025, from https://unsloth.ai/.
36. OpenAI Library. platform.openai.com. Retrieved 25 March 2025, from https://platform.openai.com/docs/libraries.
37. LM Studio application. lmstudio.ai. Retrieved 25 March 2025, from https://lmstudio.ai/.
38. Llama.cpp library. github.com. Retrieved 25 March 2025, from https://github.com/ggerganov/llama.cpp.
39. Бібліотека Google GenAI. cloud.google.com. Retrieved 25 March 2025, from https://cloud.google.com/vertex-ai/generative-ai/docs/sdks/overview.
40. Nvidia Ampere architecture. nvidia.com. Retrieved 25 March 2025, from https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.
41. Yudhishthira K., et al. (2025). A Comparison of the Cerebras Wafer-Scale Integration Technology with Nvidia GPU-based Systems for Artificial Intelligence. https://doi.org/10.48550/arXiv.2303.08774.
42. Intel FPGA for neural networks. intel.com. Retrieved 25 March 2025, from https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10/nx.html.
43. Jouppi, N. P., et al. (2017). In-Datacenter Performance Analysis of a Tensor Processing Unit. 44th International Symposium on Computer Architecture (ISCA). Toronto, Canada. https://doi.org/10.48550/arXiv.1704.04760
44. Analog AI IBM research. research.ibm.com. Retrieved 25 March 2025, from https://research.ibm.com/projects/analog-ai
45. Analog AI chip. research.ibm.com. Retrieved 25 March 2025, from https://research.ibm.com/blog/analog-ai-chip-low-power.
46. Cosemans, S., Verhoef, B. E., Doevenspeck, J., Papistas, I., Catthoor, F., Debacker, P., Mallik, A., & Verkest, D. (2019). Towards 10000TOPS/W

DNN Inference with Analog in-Memory Computing – A Circuit Blueprint, Device Options and Requirements. 2019 IEEE International Electron Devices Meeting (IEDM) (pp. 22.2.1-22.2.4). https://doi.org/10.1109/IEDM19573.2019.8993599.

47. Imec Develops Efficient Processor-in-Memory Technique for GloFo. eetimes.com. Retrieved 25 March 2025, from https://www.eetimes.com/imec-develops-efficient-processor-in-memory-technique-for-glofo/.
48. Cerebras Wafer-Scale Engine 3. hubspotusercontent-na1.net. Retrieved 25 March 2025, from https://8968533.fs1.hubspotusercontent-na1.net/hubfs/8968533/Cerebras%20Wafer%20Scale%20Cluster%20datasheet%20-%20final.pdf
49. Tera Operations Per Second (TOPS). www.intel.com. Retrieved 25 March 2025, from https://www.intel.com/content/dam/www/central-libraries/us/en/documents/real-performance-of-fpgas-tops-gpus-in-the-race-to-accelerate-ai-whitepaper.pdf.
50. Бойко, Ю. В., Кривий, С. Л., Погорілий, С. Д. та ін. (2016). Методи та новітні підходи до проектування, управління і застосування високопродуктивних IT-інфраструктур. ВПЦ «Київський університет». 447 с.
51. Kryvyi, S. L. et al. (2020). Method of semantic application verification in GPGPU technology. System Research & Information Technologies, (3), (pp. 7-22).
52. Krichen, M. et al. (2022). Are Formal Methods Applicable To Machine Learning And Artificial Intelligence? Proceedings of 2nd International Conference of Smart Systems and Emerging Technologies (SMARTTECH) (pp. 48-53).
53. Рекурентна нейронна мережа з довгою та короткочасною пам’яттю. medium.com. Retrieved 25 March 2025, from https://mohitv.medium.com/lstm-networks-75d44ac8280f.

54. Архітектура Transformers. jalammar.github.io. Retrieved 25 March 2025, from https://jalammar.github.io/illustrated-transformer/.
55. Brown, T. B. et al. (2020). Language Models are Few-Shot Learners. arXiv preprint arXiv:2005.14165. https://doi.org/10.48550/arXiv.2005.14165.
56. OpenAI, Achiam, J., Adler, S. et al. (2023). GPT-4 Technical Report. arXiv preprint arXiv:2303.08774. https://doi.org/10.48550/arXiv.2303.08774.
57. Liu, Y. et al. (2021). RoBERTa: A Robustly Optimized BERT Pretraining Approach. Proceedings of the 20th Chinese National Conference on Computational Linguistics (pp. 1218–1227). https://doi.org/10.48550/arXiv.1907.11692.
58. RoBERTa. github.com. Retrieved 25 March 2025, from https://github.com/youscan/language-models.
59. Набір даних OSCAR. huggingface.co. Retrieved 25 March 2025, from https://huggingface.co/datasets/oscar-corpus/OSCAR-2109.
60. Ainslie, J., et al. (2023). GQA: Training Generalized Query Models for Efficient Attention. arXiv preprint arXiv:2305.13245. https://arxiv.org/abs/2305.13245.
61. Meta Llama 3: The most capable openly available LLM to date. ai.meta.com. Retrieved 25 March 2025, from https://ai.meta.com/blog/meta-llama-3/.
62. Grattafiori, A., Almahairi, A., Xu, P., Scialom, T., Zettlemoyer, L., van der Maaten, L., et al. (2024). The Llama 3 Herd of Models. arXiv preprint arXiv:2407.21783. https://doi.org/10.48550/arXiv.2407.21783.
63. Gemma 3: State-of-the-art open models. blog.google. Retrieved 25 March 2025, from https://blog.google/technology/developers/gemma-3/.
64. Kamath, A., Ferret, J., Pathak, S., Vieillard, N. et al. (2025). Gemma 3 Technical Report. arXiv preprint arXiv:2503.19786. https://doi.org/10.48550/arXiv.2503.19786.

65. Su, J. et al. (2021). RoFormer: Enhanced Transformer with Rotary Position Embedding. arXiv preprint arXiv:2104.09864. https://doi.org/10.48550/arXiv.2104.09864.
66. Модель DeepSeek-R1. github.com. Retrieved 25 March 2025, from https://github.com/deepseek-ai/DeepSeek-R1.
67. DeepSeek-AI, Guo, D., Yang, D., Zhang, H. et al. (2025). DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv preprint arXiv:2501.12948. https://doi.org/10.48550/arXiv.2501.12948.
68. Peng, B., Quesnelle, J., Fan, H., & Shippole, E. (2023). YaRN: Efficient Context Window Extension of Large Language Models. arXiv preprint arXiv:2309.00071. https://doi.org/10.48550/arXiv.2309.00071.
69. Cai, W., Jiang, J., Wang, F., Tang, J., Kim, S., & Huang, J. (2024). A Survey on Mixture of Experts. arXiv preprint arXiv:2407.06204. https://doi.org/10.48550/arXiv.2407.06204.
70. Li, S., Ning, X., Wang, L., Liu, T. et al. (2024). Evaluating Quantized Large Language Models. arXiv preprint arXiv:2402.18158. https://doi.org/10.48550/arXiv.2402.18158.
71. Learning to reason with LLMs. openai.com. Retrieved 25 March 2025, from https://openai.com/index/learning-to-reason-with-llms/.
72. DeepSeek-AI, Liu, A., Feng, B., Xue, B., Wang, B. et al. (2024). DeepSeek-V3 Technical Report. arXiv preprint arXiv:2412.19437. https://doi.org/10.48550/arXiv.2412.19437.
73. DeepSeek-AI, Liu, A., Feng, B., Wang, B., Wang, B. et al. (2024). DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. arXiv preprint arXiv:2405.04434. https://doi.org/10.48550/arXiv.2405.04434.
74. Yang, A., Yang, B., Zhang, B., Hui, B., Zhen, B. et al. (2024). Qwen2.5 Technical Report. arXiv preprint arXiv:2412.15115. https://doi.org/10.48550/arXiv.2412.15115.

75. What Is Agentic AI?. blogs.nvidia.com. Retrieved 25 March 2025, from https://blogs.nvidia.com/blog/what-is-agentic-ai/.
76. Chatbot Arena. lmarena.ai. Retrieved 25 March 2025, from https://lmarena.ai/.
77. Smarter, faster, more helpful: Our latest AI updates. blog.google. Retrieved 25 March 2025, from https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/#gemini-2-0.
78. Gemini Flash Thinking. deepmind.google. Retrieved 25 March 2025, from https://deepmind.google/technologies/gemini/flash-thinking/.
79. OpenAI o3 mini. openai.com. Retrieved 25 March 2025, from https://openai.com/index/openai-o3-mini/.
80. Sahoo, P., Singh, A. K., Saha, S., Jain, V. et al. (2024). A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. arXiv preprint arXiv:2402.07927. https://doi.org/10.48550/arXiv.2402.07927.
81. Meta Llama 3.2: Powering the next wave of AI innovation on device and in the cloud. ai.meta.com. Retrieved 25 March 2025, from https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/.
82. Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: Efficient Finetuning of Quantized LLMs. arXiv preprint arXiv:2305.14314. https://doi.org/10.48550/arXiv.2305.14314.
83. Bagga, A., & Baldwin, B. (1998). Algorithms for Scoring Coreference Chains. The First International Conference on Language Resources and Evaluation Workshop on Linguistics Coreference (pp. 563-566).
84. Vilain, M., Burger, J., Aberdeen, J., Connolly, D., & Hirschman, L. (1995). A Model-Theoretic Coreference Scoring Scheme. Proceedings of the 6th Conference on Message Understanding (MUC), (pp. 45-52).

ДОДАТОК А. ЛІСТИНГ ПРОГРАМНОГО КОДУ РЕАЛІЗАЦІЇ МЕТРИКИ B-CUBED ТА MUC ДЛЯ ОЦІНКИ ЯКОСТІ ЗНАХОДЖЕННЯ КОРЕФЕРЕНТНИХ ОБ’ЄКТІВ

```
import math

# Клас для обчислення метрик оцінки ефективності B-cubed та
MUC для задачі пошуку кореферентних об’єктів.
class CoreferenceCalculator(object):

    # Ініціалізує стан калькулятора для акумуляції значень
метрик B-cubed та MUC.
    def __init__(self):
        self.objects_count = 0
        self.sum_b_cubed_precision = 0
        self.sum_b_cubed_recall = 0
        self.muc_recall_nominator = 0
        self.muc_recall_denominator = 0
        self.muc_precision_nominator = 0
        self.muc_precision_denominator = 0
        self.text_counter = 0

    # Обчислює метрики B-cubed та MUC, порівнюючи істинні та
передбачені кореферентні кластери.
    def calculate_coreference(self, correct_clusters,
predicted_clusters):
        print(correct_clusters)
        print(predicted_clusters)

        # self.print_curr_text()
        for i in range(len(correct_clusters)):  # Calculate
B-cubed metrics
            for j in range(len(correct_clusters[i])):
                for k in range(len(predicted_clusters)):
                    for z in
range(len(predicted_clusters[k])):
                        if correct_clusters[i][j] ==
predicted_clusters[k][z]:

#print(curr_correct_objects_list[i][j])
                            common_elements = 0
                            for m in
range(len(correct_clusters[i])):
```

```
                                for          n          in
range(len(predicted_clusters[k])):
                                    if
correct_clusters[i][m] == predicted_clusters[k][n]:
                                        common_elements += 1
                            len_correct                  =
len(correct_clusters[i])
                            len_predicted                =
len(predicted_clusters[k])
                            b_cubed_precision_curr       =
common_elements / len_predicted
                            b_cubed_recall_curr          =
common_elements / len_correct
                            self.sum_b_cubed_precision    +=
b_cubed_precision_curr
                            self.sum_b_cubed_recall       +=
b_cubed_recall_curr
                            self.objects_count += 1

            for i in range(len(correct_clusters)):  # Calculate
MUC metrics  (recall)
                len_correct = len(correct_clusters[i])
                if len_correct == 1:  # Якщо одинична група -
пропускаємо
                    continue
                clusters_number = 0
                number_of_elements_to_find = len_correct
                list_group_numbers = []
                for j in range(len(correct_clusters[i])):
                    for k in range(len(predicted_clusters)):
                        for                 z               in
range(len(predicted_clusters[k])):
                            if    correct_clusters[i][j]     ==
predicted_clusters[k][z]:
                                number_of_elements_to_find -= 1
                                if k not in list_group_numbers:
                                    clusters_number += 1
                                    list_group_numbers.append(k)
                            if number_of_elements_to_find == 0:
                                break
                        if number_of_elements_to_find == 0:
                            break
```

```
                    if number_of_elements_to_find == 0:
                        break
                self.muc_recall_nominator   +=   len_correct   -
clusters_number
                self.muc_recall_denominator += len_correct - 1

            for   i   in   range(len(predicted_clusters)):      #
Calculate MUC metrics  (precision)
                len_correct = len(predicted_clusters[i])
                if len_correct == 1:   # Якщо одинична група -
пропускаємо
                    continue
                clusters_number = 0
                number_of_elements_to_find = len_correct

                list_group_numbers = []
                for j in range(len(predicted_clusters[i])):
                    for k in range(len(correct_clusters)):
                        for                  z                 in
range(len(correct_clusters[k])):
                            if     correct_clusters[k][z]     ==
predicted_clusters[i][j]:
                                number_of_elements_to_find -= 1
                                if k not in list_group_numbers:
                                    clusters_number += 1
                                    list_group_numbers.append(k)
                            if number_of_elements_to_find == 0:
                                break
                        if number_of_elements_to_find == 0:
                            break
                    if number_of_elements_to_find == 0:
                        break
                self.muc_precision_nominator  +=  len_correct  -
clusters_number
                self.muc_precision_denominator += len_correct -
1
            self.text_counter += 1

            # Захист від ділення на нуль, якщо об'єктів не було
знайдено
            b_cubed_precision   =   self.sum_b_cubed_precision   /
self.objects_count if self.objects_count else 0
```

```
            b_cubed_recall    =    self.sum_b_cubed_recall    /
self.objects_count if self.objects_count else 0

            # Захист від ділення на нуль для F1-міри
            b_cubed_f1 = 0
            if (b_cubed_precision + b_cubed_recall) != 0:
                b_cubed_f1   =   (2   *   b_cubed_precision   *
b_cubed_recall) / (b_cubed_precision + b_cubed_recall)

            print("b_cubed_precision        =        "        +
str(b_cubed_precision))
            print("b_cubed_recall = " + str(b_cubed_recall))
            print("b_cubed_f1 = " + str(b_cubed_f1))

            muc_recall = 0
            muc_precision = 0
            muc_f1 = 0

            # Захист від ділення на нуль для MUC
            if self.muc_recall_denominator != 0:
                muc_recall    =    self.muc_recall_nominator    /
self.muc_recall_denominator
            if self.muc_precision_denominator != 0:
                muc_precision  =  self.muc_precision_nominator  /
self.muc_precision_denominator

            # Захист від ділення на нуль для F1-міри MUC
            if (muc_precision + muc_recall) != 0:
                muc_f1  =  (2  *  muc_precision  *  muc_recall)  /
(muc_precision + muc_recall)

            print("muc_precision = " + str(muc_precision))
            print("muc_recall = " + str(muc_recall))
            print("muc_f1      =      "      +      str(muc_f1))
```

ДОДАТОК Б. ЛІСТИНГ ПРОГРАМНОГО КОДУ СТВОРЕННЯ ДЕРЕВА РІШЕНЬ ДЛЯ ПОШУКУ КОРЕФЕРЕНТНИХ ОБ’ЄКТІВ В УКРАЇНОМОВНИХ ТЕКСТАХ

```
import os
import copy
import graphviz
import numpy as np
from sklearn import tree
from scipy import spatial
# import nltk
# from transformers import AutoTokenizer, AutoModelForMaskedLM, pipeline
# nltk.download('punkt')
# tokenizer = AutoTokenizer.from_pretrained("youscan/ukr-roberta-base")
# model = AutoModelForMaskedLM.from_pretrained("youscan/ukr-roberta-base")
# pipe = pipeline('fill-mask', model = model, tokenizer=tokenizer, device=0)

# структура даних корпусу текстів №1
# ('ID', b'int', 'NO', 'PRI', None, 'auto_increment') - Ідентифікатор слова
# ('RawText', b'text', 'NO', '', None, '') - Саме слово
# ('DocumentID', b'varchar(36)', 'NO', '', None, '') - Ідентифікатор тексту до якого входить слово
# ('WordOrder', b'smallint', 'NO', '', None, '') - Порядковий номер слова в межах тексту
# ('PartOfSpeech', b'varchar(20)', 'YES', '', None, '') - Частина мови
# ('Lemmatized', b'text', 'NO', '', None, '') - Початкова (лематизована) форма слова
# ('IsPlural', b'tinyint(1)', 'YES', '', None, '') - Чи множина
# ('IsProperName', b'tinyint(1)', 'YES', '', None, '') - Чи власна назва
# ('IsHeadWord', b'tinyint(1)', 'YES', '', None, '') - Чи є слово головним у словосполученні
# ('Gender', b'varchar(4)', 'YES', '', None, '') - Рід
# ('EntityID', b'varchar(36)', 'YES', '', None, '') - Ідентифікатор об’єкта до якого входить слово (окреме чи входить в словосполучення)
```

```
    # ('RawTagString', b'text', 'NO', '', None, '') - Не використовується
    # ('CoreferenceGroupID', b'varchar(36)', 'YES', '', None, '') - Ідентифікатор кореферентної групи, до якої входить слово
    # ('TimeCreated', b'timestamp', 'NO', '', b'CURRENT_TIMESTAMP', 'DEFAULT_GENERATED') - Не використовується
    # ('RemoteIPAddress', b'varchar(255)', 'YES', '', None, '') - Не використовується

    # Зчитує попередньо оброблені дані корпусу текстів та їх векторні представлення (ELMo) з файлів.
    def read_data(filename):
        filenameSave = os.path.join(os.getcwd(), filename)
        data = np.load(filenameSave + '.npz', allow_pickle=True)["arr_0"]
        return data

    # Реалізує метод пошуку кореферентних об'єктів в україномовних текстах на основі дерев рішень.
    class CoreferenceCalculator(object):
        # Ініціалізує обчислювач, завантажуючи дані корпусу та векторні представлення.
        def __init__(self):
            self.data = read_data("coreference")
            self.vectorized_text = read_data("coreference_vectors")
            self.curr_text = np.array(np.expand_dims(self.data[0], axis=0))
            self.word_index = 0
            self.text_id = ''
            self.text_counter = 0
            self.avrage_sum_number_of_comparisons = 0
            self.number_of_comparisons = 0
            self.curr_objects_list = []
            self.curr_correct_objects_list = []
            self.curr_vectorized_text = np.zeros((0, 1024))
            self.decision_tree = None
            self.curr_predicted_objects_list = []

        # Основний метод, що виконує підготовку даних, виділення лінгвістичних характеристик,
```

```
        # навчання дерева рішень та оцінку його ефективності на
основі метрик B-cubed та MUC.
        def calculate_coreference(self):
            print(self.vectorized_text.shape)
            print(self.data.shape)
            # X = np.expand_dims(np.zeros(10), axis=0)
            X = np.zeros((0, 12)) # Матриця ознак для навчання
дерева
            Y = np.zeros(0) # Вектор цільових значень (міток
кореферентності)
            while self.word_index < len(self.data):
                    self.prepare_current_text() # Готуємо дані
поточного тексту
                    self.prepare_curr_vectorized_text()         #
Готуємо вектори для поточного тексту
                    self.prepare_objects()      #      Виділяємо
потенційні об'єкти
                    # print(len(self.curr_objects_list))
                    print(self.curr_objects_list)
                    len_objects_list                           =
len(self.curr_objects_list)
                    self.prepare_correct_objects()   #   Готуємо
еталонні кластери
                    print(self.curr_correct_objects_list)
                    # self.print_curr_text()
                    x_temp    =    np.expand_dims(np.zeros(12),
axis=0) # Тимчасова матриця ознак для поточного тексту
                    y_temp  =  np.zeros(0)  #  Тимчасовий  вектор
міток для поточного тексту
                    # Ітерація по парах потенційно кореферентних
об'єктів для формування навчальної вибірки
                    for i in range(len_objects_list):
                        k = i + 1
                        while k < len_objects_list:
                            #    Обчислення    лінгвістичних    та
семантичних характеристик для пари об'єктів
                            cos_similatity                     =
self.cosine_similarity(i, k)
                            number_words_between_objects       =
self.words_between_objects(i, k)
                            number_objects_inbetween           =
self.count_objects_inbetween(i, k)
```

```
                            len_first_object                  =
self.len_object(i)
                            len_second_object                 =
self.len_object(k)
                            is_first_pronoun                  =
self.is_pronoun(i)
                            is_second_pronoun                 =
self.is_pronoun(k)
                            is_first_proper_name              =
self.is_proper_name(i)
                            is_second_proper_name             =
self.is_proper_name(k)
                            lematized_same                    =
self.is_lematized_same(i, k)
                            gender_same = self.is_gender_same(i,
k)
                            number_same = self.is_number_same(i,
k)
                            is_coherent                       =
self.is_coherent_real(i,   k)   #   Отримання   істинної   мітки
кореферентності
                            #  Формування  вектора  ознак  для
поточної пари
                            temp = np.expand_dims(
                                np.array((cos_similatity,
number_words_between_objects, number_objects_inbetween,
                                         len_first_object,
len_second_object, is_first_pronoun, is_second_pronoun,
                                         is_first_proper_name,
is_second_proper_name, lematized_same, gender_same,
                                         number_same)), axis=0)
                            # Додавання вектора ознак та мітки
до тимчасових масивів
                            x_temp  =  np.append(x_temp,  temp,
axis=0)
                            y_temp     =     np.append(y_temp,
is_coherent)
                            k += 1
                    if self.text_counter >= 1:
                        x_temp  =  np.delete(x_temp,  0,  0)  #
Видалення початкового нульового рядка
```

```
                    # Додавання ознак та міток поточного тексту
до загальних масивів X та Y
                    X = np.append(X, x_temp, axis=0)
                    if  self.text_counter  ==  1:  #  Спеціальна
обробка для першого тексту
                        X = np.copy(x_temp)
                    Y = np.append(Y, y_temp)
                    # Обмеження кількості текстів для навчання
                    if self.text_counter > 1500:
                        break
                    self.text_counter += 1
                    print(self.text_counter)
                    print(self.word_index)
                    self.avrage_sum_number_of_comparisons     +=
self.number_of_comparisons

            # Створення та навчання моделі дерева рішень
            #                  decision_tree                    =
tree.DecisionTreeClassifier(min_impurity_decrease=0.000005)#,
class_weight='balanced')
            #                  decision_tree                    =
tree.DecisionTreeClassifier(min_impurity_decrease=0.000003,
criterion='gini')#, max_depth=4)
            decision_tree                                       =
tree.DecisionTreeClassifier(min_impurity_decrease=0.000005,
criterion='gini',   max_depth=5)   #   Обмеження   глибини   та
мінімального зменшення неоднорідності
            #  decision_tree  =  tree.DecisionTreeClassifier()  #
Дерево без обмежень
            print(Y)
            decision_tree = decision_tree.fit(X, Y) # Навчання
дерева на всіх підготовлених даних
            print(decision_tree.predict(X))
            #                  decision_tree                    =
decision_tree.fit(X[0:X.shape[0]//2],    Y[0:X.shape[0]//2])    #
Навчання на першій половині даних

            # Назви ознак для візуалізації дерева
            feature_names          =          ['cos_similatity',
'number_words_between_objects', 'number_objects_inbetween',
                                          'len_first_object',
'len_second_object', 'is_first_pronoun', 'is_second_pronoun',
```

```
'is_first_proper_name',                    'is_second_proper_name',
'lematized_same', 'gender_same',
                                            'number_same']
             # Скорочені назви ознак
             feature_names   =['cosSim',   'nWBtw',   'nObjBtw',
'len1', 'len2', '1pron', '2pron', '1prp', '2prp', 'lemS',
                             'gendS', 'numS']
             class_names = ['NonCoref', 'Coref'] # Назви класів
             # Візуалізація дерева рішень (текстовий варіант)
             tree.plot_tree(decision_tree,
                            feature_names=feature_names,
                            class_names=class_names,
                            #filled=True   #   Заливка   вузлів
кольором
                            )
             # Візуалізація дерева рішень за допомогою Graphviz
(потребує встановленої бібліотеки та Graphviz)
            #                      dot_data                      =
tree.export_graphviz(decision_tree,feature_names=feature_names,
            #                             class_names=class_names,
out_file=None)
            # graph = graphviz.Source(dot_data)
            # graph.render("Coreference", view=True)

             # Оцінка точності дерева на навчальній та тестовій
(друга половина) вибірках
             print(X.shape)
             print(decision_tree.score(X[0:X.shape[0]//2],
Y[0:X.shape[0]//2]))
             print(decision_tree.score(X[X.shape[0]//2:-1],
Y[X.shape[0]//2:-1]))
             print(np.count_nonzero(Y))   #   Кількість   істинно
кореферентних пар
             print(np.count_nonzero(decision_tree.predict(X)))  #
Кількість пар, передбачених як кореферентні
             self.decision_tree  =  decision_tree  #  Збереження
навченого дерева

             # --- Етап тестування та оцінки метрик ---
             self.text_counter = 0
             self.word_index = 0
```

```
            self.word_index = 234919 # Початок індексу для
тестової вибірки

            # Ініціалізація змінних для акумуляції метрик B-
cubed та MUC
            objects_count = 0
            sum_b_cubed_precision = 0
            sum_b_cubed_recall = 0
            muc_recall_nominator = 0
            muc_recall_denominator = 0
            muc_precision_nominator = 0
            muc_precision_denominator = 0

            # Обробка текстів з тестової вибірки
            while self.word_index < len(self.data):
                self.prepare_current_text()
                self.prepare_curr_vectorized_text()
                self.prepare_objects()
                self.predict_coreference()    #    Передбачення
кластерів за допомогою навченого дерева
                print(self.text_counter)
                print(self.word_index)
                self.prepare_correct_objects()    #    Отримання
еталонних кластерів для поточного тексту
                print(self.curr_correct_objects_list)
                print(self.curr_predicted_objects_list)

                # Розрахунок метрики B-cubed для поточного
тексту
                for                    i                    in
range(len(self.curr_correct_objects_list)):
                    for                  j                  in
range(len(self.curr_correct_objects_list[i])):
                        for                k                in
range(len(self.curr_predicted_objects_list)):
                            for              z              in
range(len(self.curr_predicted_objects_list[k])):
                                if
self.curr_correct_objects_list[i][j]                        ==
self.curr_predicted_objects_list[k][z]:
                                    common_elements = 0
```

```
                                    for          m           in
range(len(self.curr_correct_objects_list[i])):
                                        for         n         in
range(len(self.curr_predicted_objects_list[k])):
                                            if
self.curr_correct_objects_list[i][m]                         ==
self.curr_predicted_objects_list[k][n]:
                                                common_elements
+= 1
                                    len_correct                =
len(self.curr_correct_objects_list[i])
                                    len_predicted              =
len(self.curr_predicted_objects_list[k])
                                    # Захист від ділення на нуль
                                    b_cubed_precision_curr     =
common_elements / len_predicted if len_predicted else 0
                                    b_cubed_recall_curr        =
common_elements / len_correct if len_correct else 0
                                    sum_b_cubed_precision     +=
b_cubed_precision_curr
                                    sum_b_cubed_recall        +=
b_cubed_recall_curr
                                    objects_count += 1

                # Розрахунок метрики MUC (повнота) для поточного
тексту
                for                    i                     in
range(len(self.curr_correct_objects_list)):
                    len_correct                                =
len(self.curr_correct_objects_list[i])
                    if len_correct == 1: # Одиничні кластери не
враховуються в MUC
                        continue
                    clusters_number = 0 # Кількість передбачених
кластерів, що містять елементи істинного кластера
                    number_of_elements_to_find = len_correct
                    list_group_numbers  =  []  #  Список  номерів
передбачених кластерів
                    for                  j                   in
range(len(self.curr_correct_objects_list[i])):
                        for                k                 in
range(len(self.curr_predicted_objects_list)):
```

```
                             for              z              in
range(len(self.curr_predicted_objects_list[k])):
                                 if
self.curr_correct_objects_list[i][j]                         ==
self.curr_predicted_objects_list[k][z]:
                                     number_of_elements_to_find -
= 1
                                     if      k      not      in
list_group_numbers:
                                         clusters_number += 1

list_group_numbers.append(k)
                                 if number_of_elements_to_find ==
0:
                                     break
                             if number_of_elements_to_find == 0:
                                 break
                         if number_of_elements_to_find == 0:
                             break
                     muc_recall_nominator   +=   len_correct   -
clusters_number # Чисельник повноти MUC
                     muc_recall_denominator += len_correct - 1 #
Знаменник повноти MUC

                 #  Розрахунок  метрики  MUC  (влучність)  для
поточного тексту
                 for                   i                   in
range(len(self.curr_predicted_objects_list)):
                     len_predicted_cluster                    =
len(self.curr_predicted_objects_list[i])
                     if  len_predicted_cluster  ==  1:  #  Одиничні
кластери не враховуються в MUC
                         continue
                     clusters_number  =  0  #  Кількість  істинних
кластерів, що містять елементи передбаченого кластера
                     number_of_elements_to_find               =
len_predicted_cluster
                     list_group_numbers  =  []  #  Список  номерів
істинних кластерів
                     for                 j                 in
range(len(self.curr_predicted_objects_list[i])):
```

```
                        for                 k                 in
range(len(self.curr_correct_objects_list)):
                            for                z              in
range(len(self.curr_correct_objects_list[k])):
                                if
self.curr_correct_objects_list[k][z]                          ==
self.curr_predicted_objects_list[i][j]:
                                    number_of_elements_to_find -
= 1
                                    if       k       not       in
list_group_numbers:
                                        clusters_number += 1

list_group_numbers.append(k)
                                if number_of_elements_to_find ==
0:
                                    break
                            if number_of_elements_to_find == 0:
                                break
                        if number_of_elements_to_find == 0:
                            break
                    muc_precision_nominator                   +=
len_predicted_cluster  -  clusters_number  #  Чисельник  влучності
MUC
                    muc_precision_denominator                 +=
len_predicted_cluster - 1 # Знаменник влучності MUC

                # Обмеження кількості текстів для тестування
                if self.text_counter > 1015:
                    break
                self.text_counter += 1

            # Розрахунок фінальних значень метрик B-cubed та MUC
            # Захист від ділення на нуль
            b_cubed_precision    =    sum_b_cubed_precision    /
objects_count if objects_count else 0
            b_cubed_recall = sum_b_cubed_recall / objects_count
if objects_count else 0
            b_cubed_f1    =    (2    *    b_cubed_precision    *
b_cubed_recall)   /   (b_cubed_precision   +   b_cubed_recall)   if
(b_cubed_precision + b_cubed_recall) else 0
```

```
            print("b_cubed_precision:          "           +
str(b_cubed_precision))
            print("b_cubed_recall: " + str(b_cubed_recall))
            print("b_cubed_f1: " + str(b_cubed_f1) + "\n")

            muc_recall                                       =
muc_recall_nominator/muc_recall_denominator                 if
muc_recall_denominator else 0
            muc_precision                                    =
muc_precision_nominator/muc_precision_denominator           if
muc_precision_denominator else 0
            muc_f1  =  (2  *  muc_precision  *  muc_recall)  /
(muc_precision  +  muc_recall)  if  (muc_precision  +  muc_recall)
else 0
            print("muc_precision: " + str(muc_precision))
            print("muc_recall: " + str(muc_recall))
            print("muc_f1: " + str(muc_f1) + "\n")

        # Виділяє та готує дані поточного тексту для аналізу з
повного корпусу.
        def prepare_current_text(self):
            self.curr_text                                   =
np.array(np.expand_dims(self.data[0], axis=0))
            while self.word_index < len(self.data):
                if               self.text_id                !=
self.data[self.word_index][2]:  # Перевірка зміни ідентифікатора
тексту
                    self.text_id = self.data[self.word_index][2]
                    self.curr_text  =  np.delete(self.curr_text,
0, 0) # Видалення початкового рядка
                    return
                #  Додавання  рядка  даних  до  масиву  поточного
тексту
                self.curr_text    =    np.append(self.curr_text,
np.expand_dims(self.data[self.word_index], axis=0), axis=0)
                self.word_index += 1

        # Ідентифікує потенційні об'єкти (згадки) в поточному
тексті на основі їхніх ідентифікаторів сутностей (EntityID).
        def prepare_objects(self):
            self.curr_objects_list = [] # Список об'єктів (кожен
об'єкт - список списків індексів слів)
```

```
            j = 0
            while j < self.curr_text.shape[0]:
                # Якщо слово має EntityID, вважаємо його (або
групу слів з тим же ID) об'єктом
                if self.curr_text[j][10] != None:
                    self.curr_objects_list.append([[j,   ]])   #
Створюємо новий об'єкт з одного слова
                    # Перевіряємо наступні слова на той самий
EntityID
                    for     k     in     range(j     +     1,
self.curr_text.shape[0]):
                        if      self.curr_text[j][10]       ==
self.curr_text[k][10]:

self.curr_objects_list[len(self.curr_objects_list)          -
1]][0].append(k) # Додаємо індекс слова до останнього об'єкта
                            j  +=  1  #  Пропускаємо  вже  додане
слово
                        else:
                            break # Зупиняємось, якщо EntityID
змінився
                j += 1

        #  Формує  еталонні  (істинні)  кластери  кореферентних
об'єктів  для  поточного  тексту  на  основі  даних  розмітки
(CoreferenceGroupID).
        def prepare_correct_objects(self):
            self.curr_correct_objects_list                    =
copy.deepcopy(self.curr_objects_list) # Копіюємо список об'єктів
            self.number_of_comparisons = 0
            len_objects_list                                  =
len(self.curr_correct_objects_list)
            j = 0
            # Ітерація по об'єктах для об'єднання в кластери
            while j < len_objects_list:
                k = j + 1
                while k < len_objects_list:
                    self.number_of_comparisons += 1
                    # Перевірка, чи об'єкти належать до однієї
кореферентної  групи  (за  CoreferenceGroupID  першого  слова
об'єкта)
```

```
                    if
self.curr_text[self.curr_correct_objects_list[j][0][0]][12]    !=
None and \

self.curr_text[self.curr_correct_objects_list[j][0][0]][12] == \

self.curr_text[self.curr_correct_objects_list[k][0][0]][12]:
                        # Додаємо всі слова k-го об'єкта до j-го
кластера
                        for          word_indices            in
self.curr_correct_objects_list[k]:

self.curr_correct_objects_list[j].append(word_indices)
                        # Видаляємо k-й об'єкт зі списку
                        del self.curr_correct_objects_list[k]
                        k  -=  1  #  Зменшуємо  індекс,  бо  список
скоротився
                        len_objects_list   -=   1   #   Зменшуємо
довжину списку
                    k += 1
                j += 1


        #   Виділяє   векторні   представлення   (ELMo)   для   слів
поточного тексту.
        def prepare_curr_vectorized_text(self):
            #  Вибираємо  відповідні  вектори  з  повного  масиву
векторів
            self.curr_vectorized_text                            =
self.vectorized_text[self.word_index                             -
self.curr_text.shape[0]:self.word_index]

        #  Використовує  навчене  дерево  рішень  для  передбачення
кореферентних  зв'язків  та  формування  кластерів  у  поточному
тексті.
        def predict_coreference(self):
            self.curr_predicted_objects_list                     =
copy.deepcopy(self.curr_objects_list) # Початково кожен об'єкт -
окремий кластер
            self.number_of_comparisons = 0
            len_objects_list                                     =
len(self.curr_predicted_objects_list)
```

```
            links_added = 1 # Прапорець, що показує, чи
відбулося злиття на ітерації
            # Цикл злиття кластерів, доки можливі злиття (multi-
pass clustering)
            while links_added > 0:
                links_added = 0
                i = 0
                # Ітерація по парах кластерів
                while i < len_objects_list:
                    k = i + 1
                    while k < len_objects_list:
                        self.number_of_comparisons += 1
                        pair_is_coreferent = False # Прапорець
кореферентності поточної пари кластерів
                        # Перевірка кореферентності між усіма
парами об'єктів з кластерів i та k
                        for cluster_i_object_idx in
range(len(self.curr_predicted_objects_list[i])):
                            for cluster_k_object_idx in
range(len(self.curr_predicted_objects_list[k])):
                                # Отримання індексів об'єктів у
початковому списку self.curr_objects_list
                                index1 = -1
                                index2 = -1
                                try: # Обробка можливої помилки,
якщо об'єкт не знайдено
                                    index1 =
self.curr_objects_list.index([self.curr_predicted_objects_list[i
][cluster_i_object_idx]])
                                    index2 =
self.curr_objects_list.index([self.curr_predicted_objects_list[k
][cluster_k_object_idx]])
                                except ValueError:
                                    print(f"Warning: Object not
found in original list during prediction.")
                                    continue # Пропускаємо пару,
якщо індекси не знайдено

                                # Обчислення ознак для пари
об'єктів
                                cos_similatity =
self.cosine_similarity(index1, index2)
```

```
                                 number_words_between_objects   =
self.words_between_objects(index1, index2)
                                 number_objects_inbetween       =
self.count_objects_inbetween(index1, index2)
                                 len_first_object               =
self.len_object(index1)
                                 len_second_object              =
self.len_object(index2)
                                 is_first_pronoun               =
self.is_pronoun(index1)
                                 is_second_pronoun              =
self.is_pronoun(index2)
                                 is_first_proper_name           =
self.is_proper_name(index1)
                                 is_second_proper_name          =
self.is_proper_name(k) # Помилка: має бути index2
                                 lematized_same                 =
self.is_lematized_same(index1, index2)
                                 gender_same                    =
self.is_gender_same(index1, index2)
                                 number_same                    =
self.is_number_same(index1, index2)
                                 # Формування вектора ознак
                                 features_vector                =
np.expand_dims(
                                     np.array((cos_similatity,
number_words_between_objects, number_objects_inbetween,
                                               len_first_object,
len_second_object, is_first_pronoun, is_second_pronoun,

is_first_proper_name,   is_second_proper_name,   lematized_same,
gender_same,
                                               number_same)),
axis=0)
                                 #   Передбачення   за   допомогою
дерева рішень
                                 prediction                     =
self.decision_tree.predict(features_vector)
                                 # Якщо хоча б одна пара об'єктів
з   кластерів   i   та   k   кореферентна,   вважаємо   кластери
кореферентними
                                 if prediction[0] == 1:
```

```
                                    pair_is_coreferent = True
                                    break    #    Виходимо    з
внутрішнього циклу по об'єктах кластера k
                            if pair_is_coreferent:
                                break # Виходимо з циклу по
об'єктах кластера i

                        # Якщо пара кластерів визначена як
кореферентна, зливаємо їх
                        if pair_is_coreferent:
                            # Додаємо всі об'єкти з кластера k
до кластера i
                            for        object_to_add        in
self.curr_predicted_objects_list[k]:

self.curr_predicted_objects_list[i].append(object_to_add)
                            # Видаляємо кластер k
                            del
self.curr_predicted_objects_list[k]
                            k -= 1 # Зменшуємо індекс
                            len_objects_list -= 1 # Зменшуємо
довжину списку
                            links_added += 1 # Позначаємо, що
відбулося злиття
                        k += 1
                    i += 1
                print(links_added,end=' ') # Виводимо кількість
злиттів на ітерації
            print()

        # Допоміжний метод для виведення поточного тексту та
його атрибутів на екран.
        def print_curr_text(self):
            for i in range(self.curr_text.shape[0]):
                for j in range(self.curr_text.shape[1]):
                    print('['    +    str(j)    +    ']'    +
str(self.curr_text[i][j]), end=' ')
                print()

        #   Обчислює   косинусну   схожість   між   усередненими
векторними представленнями (ELMo) двох об'єктів.
        def cosine_similarity(self, index1, index2):
```

```
            # Усереднення векторів для першого об'єкта (якщо він
складається з кількох слів)
            temp_vector1                                        =
np.zeros((len(self.curr_objects_list[index1][0]), 1024))
            for                    i                         in
range(len(self.curr_objects_list[index1][0])):
                word_idx = self.curr_objects_list[index1][0][i]
                if       0        <=        word_idx        <
len(self.curr_vectorized_text): # Перевірка індексу
                    temp_vector1[i]                            =
self.curr_vectorized_text[word_idx]
                else:
                    print(f"Warning: Index {word_idx} out of
bounds        for        vectorized        text        (len:
{len(self.curr_vectorized_text)})")
            vector1  =  np.average(temp_vector1,  axis=0)  if
temp_vector1.size > 0 else np.zeros(1024) # Усереднення або
нульовий вектор

            # Усереднення векторів для другого об'єкта
            temp_vector2                                        =
np.zeros((len(self.curr_objects_list[index2][0]), 1024))
            for                    i                         in
range(len(self.curr_objects_list[index2][0])):
                word_idx = self.curr_objects_list[index2][0][i]
                if       0        <=        word_idx        <
len(self.curr_vectorized_text): # Перевірка індексу
                   temp_vector2[i]                             =
self.curr_vectorized_text[word_idx]
                else:
                    print(f"Warning: Index {word_idx} out of
bounds        for        vectorized        text        (len:
{len(self.curr_vectorized_text)})")
            vector2  =  np.average(temp_vector2,  axis=0)  if
temp_vector2.size > 0 else np.zeros(1024) # Усереднення або
нульовий вектор

            # Обчислення косинусної схожості
            # Додаємо перевірку на нульові вектори, щоб уникнути
помилки ділення на нуль у spatial.distance.cosine
            if np.all(vector1 == 0) or np.all(vector2 == 0):
```

```
                return 0.0 # Якщо один з векторів нульовий, схожість 0
            similarity = 1 - spatial.distance.cosine(vector1, vector2)
            # Обробка можливого NaN значення, яке може виникнути при ідеально колінеарних векторах або нульових векторах
            return similarity if not np.isnan(similarity) else 1.0 # Якщо NaN, вважаємо схожість максимальною (або 0.0, залежно від логіки)

        # Обчислює кількість слів між останнім словом першого об'єкта та першим словом другого об'єкта.
        def words_between_objects(self, index1, index2):
            # Перевіряємо, чи списки індексів не порожні
            if not self.curr_objects_list[index1][0] or not self.curr_objects_list[index2][0]:
                return 0 # Або інше значення за замовчуванням
            last_element_first_group = self.curr_objects_list[index1][0][-1]
            first_element_second_group = self.curr_objects_list[index2][0][0]
            # Переконуємось, що об'єкти йдуть у правильному порядку
            if first_element_second_group <= last_element_first_group:
                return 0
            return first_element_second_group - last_element_first_group - 1 # Кількість слів *між* об'єктами

        # Підраховує кількість інших об'єктів, що знаходяться між двома заданими об'єктами.
        def count_objects_inbetween(self, index1, index2):
            if not self.curr_objects_list[index1][0] or not self.curr_objects_list[index2][0]:
                return 0
            last_element_first_group = self.curr_objects_list[index1][0][-1]
            first_element_second_group = self.curr_objects_list[index2][0][0]
            # Переконуємось, що об'єкти йдуть у правильному порядку
```

```
            if          first_element_second_group          <=
last_element_first_group:
                return 0
            objects_counter = 0
            # Перебираємо всі об'єкти, крім index1 та index2
            for i in range(len(self.curr_objects_list)):
                if  i  ==  index1  or  i  ==  index2  or  not
self.curr_objects_list[i][0]:  #  Пропускаємо  самі  об'єкти  та
порожні
                    continue
                # Перевіряємо, чи перше слово поточного об'єкта
знаходиться між заданими
                if      self.curr_objects_list[i][0][0]       >
last_element_first_group and \
                        self.curr_objects_list[i][0][0]       <
first_element_second_group:
                    objects_counter += 1
            return objects_counter

        # Повертає кількість слів у заданому об'єкті (згадці).
        def len_object(self, index):
            if not self.curr_objects_list[index][0]:
                return 0
            return len(self.curr_objects_list[index][0])

        #  Перевіряє,  чи  є  хоча  б  одне  слово  в  об'єкті
займенником (лінгвістична ознака).
        def is_pronoun(self, index):
            if not self.curr_objects_list[index][0]:
                return 0
            for                      i                      in
range(len(self.curr_objects_list[index][0])):
                word_idx = self.curr_objects_list[index][0][i]
                if  0  <=  word_idx  <  len(self.curr_text):  #
Перевірка індексу
                     if self.curr_text[word_idx][4] == "PRON":
                         return 1
                else:
                    print(f"Warning:  Index  {word_idx}  out  of
bounds for curr_text (len: {len(self.curr_text)})")
            return 0
```

```
        # Перевіряє, чи є хоча б одне слово в об'єкті власною назвою (лінгвістична ознака).
        def is_proper_name(self, index):
            if not self.curr_objects_list[index][0]:
                return 0
            for i in range(len(self.curr_objects_list[index][0])):
                word_idx = self.curr_objects_list[index][0][i]
                if 0 <= word_idx < len(self.curr_text): # Перевірка індексу
                    if self.curr_text[word_idx][7] == 1: # Використовуємо поле IsProperName (індекс 7)
                        return 1
                else:
                    print(f"Warning: Index {word_idx} out of bounds for curr_text (len: {len(self.curr_text)})")
            return 0

        # Перевіряє, чи збігаються лематизовані форми хоча б одного слова у двох об'єктах (лінгвістична ознака).
        def is_lematized_same(self, index1, index2):
            if not self.curr_objects_list[index1][0] or not self.curr_objects_list[index2][0]:
                return 0
            for i in range(len(self.curr_objects_list[index1][0])):
                word1_idx = self.curr_objects_list[index1][0][i]
                if not (0 <= word1_idx < len(self.curr_text)): continue # Перевірка індексу
                lemma1 = self.curr_text[word1_idx][5]
                if lemma1 is None: continue # Пропускаємо, якщо леми немає

                for j in range(len(self.curr_objects_list[index2][0])):
                    word2_idx = self.curr_objects_list[index2][0][j]
                    if not (0 <= word2_idx < len(self.curr_text)): continue # Перевірка індексу
                    lemma2 = self.curr_text[word2_idx][5]
                    if lemma2 is None: continue # Пропускаємо, якщо леми немає
```

```
                if lemma1 == lemma2:
                    return 1
            return 0

        # Перевіряє, чи збігається рід хоча б у однієї пари слів
з двох об'єктів (морфологічна характеристика).
        def is_gender_same(self, index1, index2):
            if  not  self.curr_objects_list[index1][0]  or  not
self.curr_objects_list[index2][0]:
                return 0
            for                      i                      in
range(len(self.curr_objects_list[index1][0])):
                word1_idx = self.curr_objects_list[index1][0][i]
                if not  (0  <=  word1_idx  <  len(self.curr_text)):
continue
                gender1 = self.curr_text[word1_idx][9]
                #  Ігноруємо  невизначений  рід  або  відсутність
інформації
                if  gender1  is  None  or  gender1  ==  "None"  or
gender1 == "": continue

                for                      j                      in
range(len(self.curr_objects_list[index2][0])):
                    word2_idx                                     =
self.curr_objects_list[index2][0][j]
                    if     not     (0     <=     word2_idx     <
len(self.curr_text)): continue
                    gender2 = self.curr_text[word2_idx][9]
                    if gender2 is None or gender2 == "None" or
gender2 == "": continue

                    if gender1 == gender2:
                        return 1
            return 0

        # Перевіряє, чи збігається число хоча б у однієї пари
слів з двох об'єктів (морфологічна характеристика).
        def is_number_same(self, index1, index2):
            if  not  self.curr_objects_list[index1][0]  or  not
self.curr_objects_list[index2][0]:
                return 0
```

```
            for                         i                       in
range(len(self.curr_objects_list[index1][0])):
                word1_idx                                        =
self.curr_objects_list[index1][0][i]
                if not (0 <= word1_idx < len(self.curr_text)):
continue
                number1 = self.curr_text[word1_idx][6] # Поле
IsPlural (індекс 6)
                if number1 is None: continue

                for                     j                       in
range(len(self.curr_objects_list[index2][0])):
                    word2_idx                                    =
self.curr_objects_list[index2][0][j]
                    if     not     (0     <=     word2_idx     <
len(self.curr_text)): continue
                    number2 = self.curr_text[word2_idx][6]
                    if number2 is None: continue

                    if number1 == number2:
                        return 1
            return 0

        # Перевіряє, чи належать перші слова двох об'єктів до
однієї істинної кореферентної групи згідно з розміткою корпусу.
        def is_coherent_real(self, index1, index2):
            # Перевіряємо, чи існують індекси та чи вони в межах
масиву тексту
            if  not  self.curr_objects_list[index1][0]  or  not
self.curr_objects_list[index2][0]:
                return 0
            word1_idx = self.curr_objects_list[index1][0][0]
            word2_idx = self.curr_objects_list[index2][0][0]

            if not (0 <= word1_idx < len(self.curr_text) and 0
<= word2_idx < len(self.curr_text)):
                return 0

            group_id1    =    self.curr_text[word1_idx][12]    #
CoreferenceGroupID (індекс 12)
            group_id2 = self.curr_text[word2_idx][12]
```

```
            # Перевіряємо, чи обидва об'єкти мають ID групи і чи
ці ID збігаються
            if group_id1 != None and group_id1 == group_id2:
                return 1
            return 0

        def TF_IDF(self):
            pass

    # Точка входу програми.
    if __name__ == '__main__':
        coreference_calculator = CoreferenceCalculator()
        coreference_calculator.calculate_coreference()
```

ДОДАТОК В. ЛІСТИНГ ПРОГРАМНОГО КОДУ РЕАЛІЗАЦІЇ МЕТОДУ ПОШУКУ КОРЕФЕРЕНТНИХ ОБ'ЄКТІВ З ВИКОРИСТАННЯМ БІНАРНОЇ КЛАСИФІКАЦІЇ НА ЕТАПІ ГЕНЕРАЦІЇ РЕЗУЛЬТАТІВ МОВНОЮ МОДЕЛЛЮ

```
import io
import json
# Імпорт класу для обчислення метрик кореферентності (передбачається, що він існує в окремому файлі)
from CoreferenceClusteringLLM import CoreferenceCalculator
# Цей застосунок виконує пошук корферентних об'єктів з використанням великої мовної моделі,
# запущеної локально (наприклад, в LM Studio або подібному середовищі).
# Він реалізує метод пошуку кореферентних об’єктів з використанням бінарної класифікації
# пар потенційно кореферентних об'єктів на етапі генерації результатів мовною моделлю,
# з подальшим об'єднанням об'єктів в кластери.

from openai import OpenAI

# Встановлення з'єднання з локальним сервером LLM
# client = OpenAI(base_url="http://localhost:1234/v1", api_key="lm-studio")
client = OpenAI(base_url="http://192.168.3.8:1234/v1", api_key="lm-studio") # Альтернативна адреса сервера



# Додає нумерацію до слів у списку, повертаючи рядок формату "слово[індекс]".
# Використовується для точної ідентифікації згадок (об'єктів) у запиті до LLM.
def numerate_words(input_list, start):
    output = ''
    for i, word in enumerate(input_list):
        output += f'{word}[{i + start}] ' # Додаємо слово та його індекс у квадратних дужках
    return output.strip() # Повертаємо рядок без зайвих пробілів на кінцях
```

```
# Перетворює список слів на один текстовий рядок.
def list_to_text(input_list):
    output = ''
    for i, word in enumerate(input_list):
        output += f'{word} ' # Додаємо слово та пробіл
    return output.strip() # Повертаємо рядок без зайвих пробілів на кінцях

# Основний блок обробки корпусу текстів
with io.open('coreference-dataset-ua//dev.jsonl', encoding='utf-8', mode='r') as file: # Відкриття файлу з корпусом даних
    coreference_calculator = CoreferenceCalculator() # Ініціалізація обчислювача метрик
    doc_counter = 0
    dataset = [] # Список для збереження даних для потенційного донавчання
    for line in file: # Ітерація по документах (рядках) у файлі
        print(doc_counter)
        doc_counter += 1
        if doc_counter > 10: # Обмеження кількості оброблюваних документів для тестування
            break
        doc_dict = json.loads(line) # Завантаження даних документа з JSON
        print(doc_dict['doc_key']) # Ідентифікатор документа
        print(doc_dict['clusters']) # Істинні (еталонні) кластери кореферентних об'єктів
        list_correct_clusters = doc_dict['clusters']

        # Підготовка списку всіх згадок (об'єктів) з істинних кластерів
        grouped_list = []
        for cluster in list_correct_clusters:
            for sublist in cluster: # Кожен sublist - це [початковий_індекс, кінцевий_індекс] згадки
                grouped_list.append(sublist)
        # Сортування згадок за їхнім початковим індексом у тексті
        grouped_list.sort()
```

```
            print(doc_dict['tokens']) # Список токенів (слів та
знаків пунктуації) документа
            input_list = doc_dict['tokens']
            predicted_clusters= [] # Список для збереження
передбачених кластерів

            # Формування початкового системного запиту (prompt)
для LLM
            messages = []
            request = ("You are a task-specific expert that
performs Coreference Resolution of Ukrainian Language "
                        " texts. This task means you have to
check if two chosen mentions from the text are coreferent "
                        "or not (to check if they refer,
correspond to the same real or imaginary object in a given
context). "
                        "You will receive a Ukrainian-language
text with words and punctuation signs numerated"
                        "starting from 0. Additionally, you will
receive a pair of mentions from this text, "
                        "which you can exactly identify by their
numeration. As a result, you should output "
                        "\[\\boxed{True}\] or \[\\boxed{False}\]
, which marks the pair of received mentions as " # Формат
очікуваної відповіді
                        "coreferent or not coreferent. "
                        # Додаткові інструкції (закоментовані)
                        # "After you will recieve a pair of
mention you should write only an answer without explanations."
                        # "First, translate the whole text into
english, and then you will recieve "
                        # "the pairs of potentially coreferent
mentions."
                        + "\nText: " +
str(list_to_text(input_list))) # Додавання тексту документа до
запиту
            messages.append({"role": "system", "content":
request}) # Встановлення ролі "system" для інструкції

            # Основний цикл кластеризації: поки є необроблені
згадки
            while len(grouped_list) > 0:
```

```
                 cluster_numbers = [0, ] # Індекси згадок у
grouped_list, які будуть включені до поточного кластера
(починаємо з першої)
                 print("New itteration")

                 list_counter = 1 # Індекс для перебору решти
згадок у grouped_list
                 # Порівняння першої згадки (grouped_list[0]) з
усіма наступними
                 for list_mention in grouped_list[1:]:
                     question = messages.copy() # Копіювання
системного запиту
                     # Формування першої та другої згадок з
нумерацією слів
                     first_mention =
numerate_words(input_list[grouped_list[0][0]:grouped_list[0][1]
+ 1], grouped_list[0][0])
                     second_mention =
numerate_words(input_list[list_mention[0]:list_mention[1] + 1],
list_mention[0])
                     # Формування запиту користувача для бінарної
класифікації
                     request_user = "First mention: " +
first_mention + " , second mention: " + second_mention #+
"\n\n### Response:\n" # Формулювання запитання про пару згадок
                     question.append({"role": "user", "content":
request_user}) # Додавання запиту користувача
                     print(question)

                     # (Для донавчання) Визначення правильної
відповіді для поточного запиту
                     correct_result = "False"
                     for cluster in list_correct_clusters:
                         if grouped_list[0] in cluster:
                             if list_mention in cluster:
                                 correct_result = "True"
                             break
                     sample_dict = {
                         "instruction" : messages[0]['content'],
# Збереження системної інструкції
                         "input" : request_user, # Збереження
запиту користувача
```

```
                    "output" : "\[\\boxed{" + correct_result
+ "}\]" # Збереження очікуваного виводу у потрібному форматі
                }
                dataset.append(sample_dict)

                # Надсилання запиту до великої мовної моделі
                completion = client.chat.completions.create(
                    #     Вибір     моделі     (закоментовані
альтернативи)
                    model="llama-3.2-3b-instruct",
                    # model="gemma-3-4b-it",
                    #                    model="llama-3.2-3b-
trained_4datasets",
                    # model="deepseek-r1-distill-qwen-7b",
                    # model="gemma-3-12b-it",
                    #   model="deepseek-r1-distill-qwen-32b-
uncensored",
                    # model="gemma-3-27b-it",
                    # model="meta-llama-3-8b-instruct",
                    # model="gemma-3-27b-it@q8_0",
                    #         model="llama-3.3-70b-instruct-
abliterated",
                    messages=question,  #  Передача  історії
діалогу (системна інструкція + запит)
                    temperature=0,  #  Температура  =  0  для
детермінованого виводу
                    max_tokens=32,  # Обмеження максимальної
довжини відповіді
                    stream=False,   #   Не   використовувати
потокову передачу
                )

                # Обробка відповіді від LLM
                result                                     =
completion.choices[0].message.content  #  Отримання  згенерованої
відповіді

                #    Інтерпретація    результату    бінарної
класифікації
                if  "boxed{True}"  in  result:  #  Перевірка
наявності маркера True
                    result_bool = True
```

```
                    else:
                        result_bool = False # В іншому випадку
вважаємо False
                    print(result_bool)

                    #   Якщо   згадки   визнані   кореферентними,
додаємо індекс другої згадки до списку для кластера
                    if result_bool:
                        cluster_numbers.append(list_counter)
                    list_counter += 1 # Перехід до наступної
згадки для порівняння

                # Формування передбаченого кластера на основі
зібраних індексів
                predicted_clusters.append([])
                # Додаємо згадки до кластера у правильному
порядку та видаляємо їх з grouped_list
                for i in reversed(range(len(cluster_numbers))):
# Ітерація у зворотному порядку, щоб уникнути проблем з
індексами при видаленні
                    predicted_clusters[-1].insert(0,
grouped_list[cluster_numbers[i]]) # Додаємо згадку на початок
кластера
                    del grouped_list[cluster_numbers[i]] #
Видаляємо оброблену згадку зі списку
                print(predicted_clusters) # Виводимо поточний
стан передбачених кластерів

            # Обчислення та виведення метрик B-cubed та MUC для
поточного документа

coreference_calculator.calculate_coreference(doc_dict['clusters'
], predicted_clusters)

        # out_file = open("train_coref_LLM_binary.json", "w",
encoding ='utf8')
        # json.dump(dataset, out_file, indent=4, ensure_ascii =
False)
        # out_file.close()
```

ДОДАТОК Г. ЛІСТИНГ ПРОГРАМНОГО КОДУ РЕАЛІЗАЦІЇ МЕТОДУ СТВОРЕННЯ КЛАСТЕРУ КОРЕФЕРЕНТНИХ ОБ'ЄКТІВ НА ЕТАПІ ГЕНЕРАЦІЇ РЕЗУЛЬТАТІВ МОВНОЮ МОДЕЛЛЮ

```
import io
import re
import json
# Імпорт класу для обчислення метрик кореферентності
from CoreferenceClusteringLLM import CoreferenceCalculator
# Цей застосунок виконує пошук кореферентних об'єктів з використанням великої мовної моделі (LLM), запущеної локально.
# Він реалізує метод створення кластеру кореферентних об’єктів на етапі генерації результатів мовною моделлю.

from openai import OpenAI

# Встановлення з'єднання з локальним сервером LLM
# client = OpenAI(base_url="http://localhost:1234/v1", api_key="lm-studio")
client = OpenAI(base_url="http://192.168.3.8:1234/v1", api_key="lm-studio") # Альтернативна адреса сервера


# Додає нумерацію до слів у списку для формування рядка згадки (mention) у запиті до ВММ.
# Повертає рядок формату "слово[індекс] слово[індекс+1] ...".
def numerate_words(input_list, start):
    output = ''
    for i, word in enumerate(input_list):
        output += f'{word}[{i + start}] ' # Додаємо слово та його глобальний індекс у тексті
    return output.strip()

# Перетворює список рядків на один текст із заданим рядком-роздільником.
# Використовується для форматування основного тексту та списку потенційних згадок для запиту до ВММ.
def list_to_text(input_list, seperator):
    output = ''
    for i, word in enumerate(input_list):
```

```
            output += f'{word}'+ str(seperator) # Додаємо
елемент списку та роздільник
        # Видаляємо останній роздільник, якщо він був доданий
        if output and seperator:
             output = output[:-len(str(seperator))]
        return output.strip()

    # Розбирає (парсить) рядкову відповідь ВММ, що містить
кластер згадок у форматі boxed{...|...}.
    # Вилучає початковий та кінцевий індекси для кожної згадки.
    # Повертає список списків: [[початок1, кінець1], [початок2,
кінець2], ...].
    def parse_cluster_string(cluster_string):
        """Parses the cluster string and returns a list of
lists."""
        # Видалення форматування "boxed{...}"
        cluster_string = cluster_string.replace("boxed{",
"").replace("}", "")
        # Розділення на окремі згадки за символом "|"
        mentions = cluster_string.split("|")
        result = []
        for mention in mentions:
            mention = mention.strip() # Видалення зайвих
пробілів
            # Пошук усіх індексів у квадратних дужках [число]
            indices = re.findall(r'\[(\d+)\]', mention)
            if not indices: # Якщо індексів не знайдено,
пропускаємо цю частину рядка
                continue
            # Вилучення першого та останнього індексів як чисел
            start_number = int(indices[0])
            last_number = int(indices[-1])
            result.append([start_number, last_number])
        return result

    # Валідує згенерований ВММ та розібраний кластер згадок.
    # Залишає лише ті згадки, які дійсно існують у списку всіх
можливих згадок (grouped_list),
    # та видаляє дублікати. Повертає відсортований список
валідних згадок.
    def check_predicted_clusters(parsed_clusters, grouped_list):
        valid_clusters = []
```

```
        seen_clusters = set() # Для відстеження унікальних
згадок
        for cluster in parsed_clusters:
            # Перевірка, чи існує така згадка у початковому
списку і чи вона ще не була додана
            if cluster in grouped_list and tuple(cluster) not in
seen_clusters:
                valid_clusters.append(cluster)
                seen_clusters.add(tuple(cluster))
        valid_clusters.sort() # Сортування валідних згадок за
початковим індексом
        return valid_clusters

    # Видаляє згадки, що містяться у валідованому кластері
(parsed_clusters),
    # з основного списку ще не оброблених згадок (grouped_list).
    def delete_mentions(parsed_clusters, grouped_list):
        """Deletes mentions existing in parsed_clusters from
grouped_list."""
        for cluster in parsed_clusters:
            try:
                grouped_list.remove(cluster) # Видалення згадки
зі списку
            except ValueError:
                pass # Ігноруємо, якщо згадки вже немає (на
випадок помилок)

    # Основний блок обробки корпусу текстів
    with io.open('coreference-dataset-ua//dev.jsonl',
encoding='utf-8', mode='r') as file: # Відкриття файлу з
корпусом даних
        coreference_calculator = CoreferenceCalculator() #
Ініціалізація обчислювача метрик
        doc_counter = 0
        dataset = [] # Список для збереження даних для
потенційного донавчання
        for line in file: # Ітерація по документах (рядках) у
файлі
            print(doc_counter)
            doc_counter += 1
            if doc_counter > 10: # Обмеження кількості
оброблюваних документів для тестування
```

```
                break
            doc_dict = json.loads(line) # Завантаження даних
документа з JSON
            list_correct_clusters = doc_dict['clusters'] #
Істинні (еталонні) кластери

            # Підготовка списку всіх згадок (об'єктів) з
істинних кластерів
            grouped_list = []
            for cluster in list_correct_clusters:
                for sublist in cluster: # Кожен sublist - це
[початковий_індекс, кінцевий_індекс] згадки
                    grouped_list.append(sublist)
            # Сортування всіх згадок за їхнім початковим
індексом у тексті
            grouped_list.sort()

            input_list = doc_dict['tokens'] # Список токенів
документа
            predicted_clusters= [] # Список для збереження
передбачених кластерів для поточного документа

            # Формування системного запиту (інструкції) для ВММ
            messages = []
            request_system = ("You are a task-specific expert
that performs Coreference Resolution of Ukrainian Language
texts. "
            "This task means you have to find all mentions from
the text that are coreferent to the chosen one "
            "(to check if they refer to the same real or
imaginary object in a given context). "
            "You will receive a Ukrainian-language text.
Additionally, you will receive "
            "the chosen first mention and a list of potentially
coreferent mentions from this text, "
            "which you can exactly identify by their numeration.
As a result, you should output a cluster "
            "of coreferent mentions to the chosen one, for
example, if mentions [мій[11] тато[12]] and [він[354]] "
            "are coreferent, you should output: boxed{мій[11]
тато[12]] | [він[354]}. " # Приклад формату виводу
```

```
            "This list should include first mentions in the first place + "
            "other mentions only from the list of potentially coreferent mentions." # Важливе уточнення щодо вмісту кластера
                        # Додаткові інструкції (закоментовані)
                        # "First, translate the whole text into english, and then proceed with the task. "
                        # "Carefully consider everything, before writing a reply."
                        + "\n\nText: ") + str(list_to_text(input_list, " ")) # Додавання тексту документа
            messages.append({"role": "system", "content": request_system}) # Встановлення ролі "system"
            print(request_system)

            # Основний цикл обробки згадок у документі: поки є необроблені згадки
            while len(grouped_list) > 0:
                question = messages.copy() # Копіювання системного запиту для поточного запиту користувача
                # Формування списку потенційно кореферентних згадок (усі, крім першої)
                potentially_coreferent_list = []
                for mention_indices in grouped_list[1:]:
                    # Форматування кожної потенційної згадки з нумерацією слів

potentially_coreferent_list.append(numerate_words(input_list[mention_indices[0]:mention_indices[1] + 1], mention_indices[0]) )
                # Об'єднання потенційних згадок у рядок з роздільником " | "
                potentially_coreferent_string = list_to_text(potentially_coreferent_list, " | ")
                # Формування першої (якірної) згадки для поточного запиту
                first_mention_string = numerate_words(input_list[grouped_list[0][0]:grouped_list[0][1] + 1], grouped_list[0][0])
                # Формування запиту користувача з якірною згадкою та списком потенційних
```

```
                request_user   =     "\nFirst   mention:   "   +
first_mention_string   + "\nPotentially coreferent mentions: " +
potentially_coreferent_string
                question.append({"role":   "user",   "content":
request_user}) # Додавання запиту користувача
                print(question)

                # Надсилання запиту до великої мовної моделі
                completion = client.chat.completions.create(
                    # Вибір моделі (закоментовані альтернативи)
                    # model="gemma-3-4b-it",
                    # model="meta-llama-3-8b-instruct",
                    # model="llama3.1-8b_trained",
                    # model="llama-3.2-3b-trained_4datasets",
                    # model="llama-3.2-3b-instruct",
                    #    model="lmstudio-community/llama-3.2-3b-
overtrained/llama3.2_overtrained11.gguf",
                    # model="llama-3.2-1b-instruct",
                    # model="llama-3.2-1b-overtrained",
                    # model="deepseek-r1-distill-qwen-7b",
                    model="llama-3.3-70b-instruct",
                    # model="gemma-3-12b-it",
                    #       model="deepseek-r1-distill-qwen-32b-
uncensored",
                    # model="gemma-3-27b-it",
                    messages=question,   #   Передача   історії
діалогу
                    temperature=0.0,  #  Температура  =  0  для
детермінованого виводу
                    max_tokens=8192,   #   Максимальна   довжина
генерованого кластера (з запасом)
                    stream=True,   #   Використання   потокової
передачі для спостереження за генерацією
                )

                #  Отримання  та  виведення  відповіді  ВММ  у
потоковому режимі
                llm_output = ""
                for chunk in completion:
                    if  chunk.choices[0].delta.content  is  not
None:
```

```
                        llm_output                           +=
chunk.choices[0].delta.content
                        print(chunk.choices[0].delta.content,
end="", flush=True) # Друк кожного фрагмента відповіді
                print()   #   Перехід   на   новий   рядок   після
завершення генерації

                # Розбір (парсинг) отриманої відповіді ВММ
                parsed_clusters_llm                            =
parse_cluster_string(llm_output)

                # Визначення індексів якірної згадки
                anchor_mention_indices   =   [grouped_list[0][0],
grouped_list[0][1]]
                #   Гарантоване   додавання   якірної   згадки   до
розібраного кластера, якщо її там немає
                if      anchor_mention_indices      not      in
parsed_clusters_llm:
                    parsed_clusters_llm.insert(0,
anchor_mention_indices) # Додаємо на початок

                #   Валідація   розібраного   кластера   (видалення
неіснуючих та дублікатів)
                validated_cluster                              =
check_predicted_clusters(parsed_clusters_llm, grouped_list)

                #   Видалення   валідованих   згадок   з   основного
списку необроблених
                delete_mentions(validated_cluster, grouped_list)

                # Додавання фінального валідованого кластера до
списку передбачених кластерів документа
                predicted_clusters.append(validated_cluster)

            # Виведення фінального списку передбачених кластерів
для документа
            print(predicted_clusters)
            # Обчислення та виведення метрик B-cubed та MUC для
поточного документа

coreference_calculator.calculate_coreference(doc_dict['clusters'
], predicted_clusters)
```

ДОДАТОК Д. ЛІСТИНГ ПРОГРАМНОГО КОДУ РЕАЛІЗАЦІЇ ДОНАВЧАННЯ НЕЙРОННОЇ МЕРЕЖІ LLAMA 3.2

```
import torch # Імпортуємо PyTorch для роботи з нейронними мережами та тензорами
print(torch.cuda.get_device_name(0)) # Виводимо назву графічного прискорювача (GPU), якщо він доступний
from trl import SFTTrainer # Імпортуємо SFTTrainer з бібліотеки TRL для донавчання (supervised fine-tuning) моделей
from transformers import TrainingArguments # Імпортуємо TrainingArguments для налаштування параметрів процесу навчання
from datasets import load_dataset # Імпортуємо load_dataset для зручного завантаження наборів даних
max_seq_length = 2048 # Задаємо максимальну довжину послідовності токенів для моделі
# Імпорт бібліотеки Unsloth для прискореного завантаження та донавчання великих мовних моделей
import unsloth
from unsloth import FastLanguageModel # Клас для швидкого завантаження моделі
from unsloth import is_bfloat16_supported # Функція для перевірки підтримки формату bfloat16

# Завантаження квантизованої великої мовної моделі та токенізатора за допомогою Unsloth.
# Використовується 4-бітна квантизація для зменшення вимог до пам'яті.
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "unsloth/Llama-3.2-3B-Instruct-bnb-4bit", # Назва попередньо навченої моделі (3 мільярди параметрів)
    # model_name = "unsloth/Llama-3.2-1B-Instruct-bnb-4bit", # Альтернативна модель (1 мільярд параметрів)
    # model_name = "unsloth/gemma-3-4b-it-unsloth-bnb-4bit", # Альтернативна модель Gemma 3
    max_seq_length = max_seq_length, # Встановлення максимальної довжини послідовності
    dtype = None, # Тип даних (визначається автоматично)
    load_in_4bit = True, # Завантаження у 4-бітному форматі
)
# Підготовка моделі до генерації відповідей (inference)
```

```
    # FastLanguageModel.for_inference(model) # Розкоментувати, якщо потрібно використовувати модель для генерації одразу після завантаження

    # Налаштування адаптерів LoRA (Low-Rank Adaptation) для ефективного донавчання моделі (техніка QLoRA).
    # Дозволяє донавчати лише невелику кількість додаткових параметрів.
    model = FastLanguageModel.get_peft_model(
        model,
        r = 128, # Ранг матриць LoRA (розмір додаткових параметрів)
        target_modules = ["q_proj", "k_proj", "v_proj", "o_proj",
                          "gate_proj", "up_proj", "down_proj",], # Модулі моделі, до яких застосовуються адаптери LoRA
        lora_alpha = 256, # Коефіцієнт масштабування LoRA
        lora_dropout = 0, # Dropout для LoRA (0 для автопідбору)
        bias = "none",    # Використання зміщень (bias) у LoRA ( "none" для автопідбору)
        #    Використання    удосконаленої    версії    gradient checkpointing від Unsloth для економії пам'яті
        use_gradient_checkpointing = "unsloth", # True або "unsloth" для довгих послідовностей
        random_state = 3407, # Задаємо стан для відтворюваності
        use_rslora = False,
        loftq_config = None,
    )

    # --- Підготовка даних для донавчання ---

    # Додає нумерацію до слів у списку для формування рядка згадки (mention) у запиті до ВММ.
    # Повертає рядок формату "слово[індекс] слово[індекс+1] ...".
    def numerate_words(input_list, start):
        output = ''
        for i, word in enumerate(input_list):
            output += f'{word}[{i + start}] '
        return output.strip()
```

```
    # Перетворює список рядків на один текст із заданим рядком-
роздільником.
    def list_to_text(input_list, seperator):
        output = ''
        for i, word in enumerate(input_list):
            output += f'{word}'+ str(seperator)
        # Видаляємо останній роздільник, якщо він був доданий
        if output and seperator:
             output = output[:-len(str(seperator))]
        return output.strip()

    EOS_TOKEN = tokenizer.eos_token # Отримуємо токен кінця
послідовності


    # Форматує приклади з набору даних у формат запитів
(prompts) та очікуваних відповідей
    # для донавчання ВММ за методом генерації кластеру
кореферентних об'єктів.
    def formatting_prompts_func(examples):
        inputs = [] # Список для збереження вхідних частин
запитів (instruction + input)
        outputs = [] # Список для збереження очікуваних вихідних
кластерів
        # formatted_texts = [] # Список для збереження повних
текстів (input + output) - закоментовано, бо використовується
chat template
        # Ітерація по кожному прикладу (документу) в батчі даних
        for doc_key, clusters, sentences, tokens in
zip(examples["doc_key"], examples["clusters"],
examples["sentences"], examples["tokens"]):
            # Створення повного тексту документа зі списку
речень/токенів
            full_text_list = [word for sentence in sentences for
word in sentence]
            full_text = list_to_text(full_text_list, " ")

            # Підготовка списку всіх можливих згадок у документі
            grouped_list = []
            for cluster in clusters:
                for sublist in cluster:
                    grouped_list.append(sublist)
```

```
            grouped_list.sort()   #   Сортування   згадок   за
початковим індексом

            #   Формування   рядка   зі   всіма   потенційно
кореферентними згадками
            potentially_coreferent_list = []
            for   mention_indices   in   grouped_list:   #
Використовуємо всі згадки як потенційні

potentially_coreferent_list.append(numerate_words(tokens[mention
_indices[0]:mention_indices[1] + 1], mention_indices[0]) )
            potentially_coreferent_string                   =
list_to_text(potentially_coreferent_list, " | ")

            # Генерація навчальних прикладів: для кожної якірної
згадки з істинного кластера
            for cluster in clusters:
                # Вибираємо першу згадку кластера як якірну
(anchor mention)
                anchor_mention_string                       =
numerate_words(tokens[cluster[0][0]:cluster[0][1]    +    1],
cluster[0][0])

                #  Формування  правильної  відповіді  (істинний
кластер для якірної згадки)
                coreferent_list_answer = []
                for mention_indices in cluster: # Беремо всі
згадки з поточного істинного кластера

coreferent_list_answer.append(numerate_words(tokens[mention_indi
ces[0]:mention_indices[1] + 1], mention_indices[0]) )
                correct_answer_string                       =
"boxed{"+list_to_text(coreferent_list_answer,  "  |  ")+"}"  #
Форматування відповіді

                #  Формування  системної  інструкції  та  запиту
користувача
                instruction = ("You are a task-specific expert
that performs Coreference Resolution of Ukrainian Language "
                             " texts. This task means you have to
find  all  mentions  from  the  text  that  are  coreferent  to  the
chosen "
```

```
                              "one (to check if they refer to the
same real or imaginary object in a given context). "
                              "You will receive a Ukrainian-
language text. Additionally, you will receive the chosen first
mention and a list of "
                              "potentially coreferent mentions
from this text, "
                              "which you can exactly identify by
their numeration. As a result, you should output "
                              " a cluster of coreferent mentions
to the chosen one, for example"
                              "if mentions [мій[11] тато[12]] and
[він[354]] is coreferent, you should output: boxed{мій[11]
тато[12]] | [він[354]}.This list should include first mention at
first plase + other mentions only from the list of potentially
coreferent mentions"
                              + "\\n\\nText: ") + full_text #
Додаємо текст документа до інструкції
                 user_input = "\\nFirst mention: " +
anchor_mention_string + "\\nPotentially coreferent mentions: " +
potentially_coreferent_string # Формуємо вхідні дані користувача

                 # Зберігаємо підготовлені вхідні та вихідні дані
                 inputs.append(instruction + user_input) #
Поєднуємо інструкцію та вхідні дані
                 outputs.append(correct_answer_string)
                 # formatted_texts.append(instruction +
user_input + correct_answer_string) # Повний текст для навчання
(якщо не використовується chat template)

         # Повертаємо словник з відформатованими даними
         # Потрібно повернути "input" та "output" для
використання з format_chat_template
         return {"input": inputs, "output": outputs}

     # Форматує дані у вигляді чату (user/assistant) за допомогою
шаблону токенізатора.
     # Це стандартний підхід для донавчання моделей, орієнтованих
на діалог/інструкції.
     def format_chat_template(row):
         # Створюємо список словників, що представляють діалог
```

```
        row_json = [{"role": "user", "content": row["input"]}, # Запит користувача (інструкція + вхідні дані)
                    {"role": "assistant", "content": row["output"]}] # Очікувана відповідь асистента
        # Застосовуємо шаблон чату токенізатора для створення єдиного текстового рядка
        row["text"] = tokenizer.apply_chat_template(row_json, tokenize=False)
        return row

    # Налаштування токенізатора для padding (доповнення послідовностей до однакової довжини)
    tokenizer.padding_side = 'right' # Додавати padding справа
    tokenizer.pad_token = tokenizer.eos_token # Використовувати токен кінця послідовності як токен padding

    # Завантаження набору даних для пошуку кореферентних об'єктів
    from datasets import Dataset
    # dataset = load_dataset("artemkramov/coreference-dataset-ua", split = "train") # Завантаження з Hugging Face Hub
    dataset = load_dataset("json", data_files="coreference-dataset-ua/train.jsonl", split = "train") # Завантаження з локального JSONL файлу
    print(dataset)

    # Застосування першої функції форматування для створення запитів та відповідей
    # Обмежуємо кількість прикладів для пришвидшення обробки/тестування
    formatted_prompt_dict = formatting_prompts_func(dataset[0:73000]) # Змінено на використання словника
    # Створення нового об'єкту Dataset зі словника
    dataset_formatted = Dataset.from_dict(formatted_prompt_dict)

    # Застосування другої функції форматування для створення фінального текстового формату для навчання
    dataset_final = dataset_formatted.map(format_chat_template, num_proc=4,)
    print(dataset_final)
```

```
# Виведення прикладу фінального відформатованого тексту
print(dataset_final['text'][0])

from trl import SFTTrainer
from transformers import TrainingArguments
from unsloth import is_bfloat16_supported

# Ініціалізація тренера SFTTrainer
trainer = SFTTrainer(
    model = model, # Модель для донавчання (з LoRA адаптерами)
    tokenizer = tokenizer, # Токенізатор
    train_dataset = dataset_final, # Підготовлений набір даних
    dataset_text_field = "text", # Назва колонки з фінальним текстом
    max_seq_length = max_seq_length, # Максимальна довжина послідовності
    dataset_num_proc = 2, # Кількість процесів для обробки датасету
    packing = True, # Упаковка коротких послідовностей для прискорення навчання
    args = TrainingArguments( # Налаштування параметрів навчання
        per_device_train_batch_size = 2, # Розмір пакету (batch size) на один GPU
        gradient_accumulation_steps = 10, # Кількість кроків для акумуляції градієнтів (ефективний batch size = 2 * 10 = 20)
        warmup_steps = 5,
        num_train_epochs = 1, # Кількість епох навчання (один повний прохід по датасету)
        # max_steps = 1000, # Альтернатива: максимальна кількість кроків навчання
        learning_rate = 2e-4, # Швидкість навчання
        fp16 = not is_bfloat16_supported(),
        bf16 = is_bfloat16_supported(),
        logging_steps = 1, # Як часто виводити лог навчання
        optim = "adamw_8bit", # Оптимізатор AdamW з 8-бітними станами для економії пам'яті
        weight_decay = 0.01, # Коефіцієнт L2-регуляризації
        lr_scheduler_type = "linear", # Тип планувальника швидкості навчання (лінійне зменшення)
```

```
            seed = 3407, # Зерно для відтворюваності
            output_dir = "outputs", # Директорія для збереження
результатів навчання
            report_to = "none",
        ),
    )

    # Виведення інформації про використання пам'яті GPU перед
початком навчання
    gpu_stats = torch.cuda.get_device_properties(0)
    start_gpu_memory = round(torch.cuda.memory_reserved(0) /
1024 / 1024 / 1024, 3) # Використовуємо memory_reserved
    max_memory = round(gpu_stats.total_memory / 1024 / 1024 /
1024, 3)
    print(f"GPU = {gpu_stats.name}. Max memory = {max_memory}
GB.")
    print(f"{start_gpu_memory} GB of memory reserved.")

    # Запуск процесу донавчання моделі
    trainer_stats = trainer.train()

    # Виведення статистики після завершення навчання
    # Використовуємо max_memory_reserved для отримання пікового
значення
    used_memory = round(torch.cuda.max_memory_reserved(0) / 1024
/ 1024 / 1024, 3)
    used_memory_for_lora = round(used_memory - start_gpu_memory,
3) # Приблизна оцінка пам'яті, використаної для навчання
    used_percentage = round(used_memory / max_memory * 100, 3)
    lora_percentage = round(used_memory_for_lora / max_memory *
100, 3)
    print(f"{trainer_stats.metrics['train_runtime']} seconds
used for training.")
    print(f"{round(trainer_stats.metrics['train_runtime']/60,
2)} minutes used for training.")
    print(f"Peak reserved memory = {used_memory} GB.")
    print(f"Peak reserved memory for training =
{used_memory_for_lora} GB.")
    print(f"Peak reserved memory % of max memory =
{used_percentage} %.")
    print(f"Peak reserved memory for training % of max memory =
{lora_percentage} %.")
```

```
# Збереження донавченої моделі (LoRA адаптерів)
model.save_pretrained("llama-3.2-3b-coref-lora")         # Зберігаємо лише адаптери LoRA
tokenizer.save_pretrained("llama-3.2-3b-coref-lora")       # Зберігаємо токенізатор разом з адаптерами
#  model.save_pretrained_merged("llama-3.2-3b-coref-merged", tokenizer, save_method = "merged_16bit",) # Альтернатива: злиття адаптерів з базовою моделлю
```

ДОДАТОК Е. СЕРТИФІКАТ ПРО УЧАСТЬ В МІЖНАРОДНІЙ НАУКОВІЙ КОНФЕРЕНЦІЇ COIA-2022 (БАКУ, АЗЕРБАЙДЖАН, 24-26 СЕРПНЯ 2022)

COIA 2022

The 8th International Conference on Control and Optimization with Industrial Applications

Baku, Azerbaijan, 24-26 August, 2022

CERTIFICATE

This is to certify that

Prof. PAVLO BILETSKYI

participated and presented in hybrid international conference COIA-2022 an online talk entitled

COREFERENCE RESOLUTION BASED ON PRETRAINED ROBERTA LANGUAGE MODEL

Fikret Aliev

Chief of the Organizing Committee

ДОДАТОК Є. СЕРТИФІКАТ ПРО УЧАСТЬ В МІЖНАРОДНІЙ НАУКОВІЙ КОНФЕРЕНЦІЇ COIA-2024 (СТАМБУЛ, ТУРЕЧЧИНА, 27-29 СЕРПНЯ 2024)

«ЗАТВЕРДЖУЮ»
Декан факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка

_________ доктор фізико-математичних наук КАШПУР О.Ф.
«___»___________2025 р.

АКТ

впровадження в навчальний процес результатів дисертаційної роботи Білецького П. В. «Створення методів пошуку кореферентних об'єктів в україномовних текстах на основі дерев рішень, нейронних мереж та великих мовних моделей», поданої на здобуття наукового ступеня доктора філософії за спеціальністю 123 – Комп'ютерна інженерія

Цей акт засвідчує, що наступні результати дисертаційної роботи Білецького Павла Володимировича:

1. методи створення алгоритмів обробки природних мов на основі дерев рішень, нейронних мереж та великих мовних моделей (розділи 2 та 3 роботи);
2. порівняння ефективності застосування великих мовних моделей для української мови (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.