Міністерство освіти і науки України
Західноукраїнський національний університет
Факультет комп’ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління

Товпишко Данило Ростиславович

Застосунок обробки природної мови на основі легких трансформерних моделей для класифікації текстів / Application for Natural Language Processing based on Lightweight Transformer Models for Text Classification

спеціальність: 122 - Комп'ютерні науки
освітньо-професійна програма - Комп’ютерні науки

Кваліфікаційна робота

Виконав: студент групи КН–41
Д. Р. Товпишко

_____

Науковий керівник:
к.т.н., доцент М. З. Домбровський

_____

Випускну кваліфікаційну роботу
допущено до захисту:
«____»_____ 2026 р.

завідувач кафедри
_____ Н. В. Дзюбановська

Тернопіль – 2026

Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління
Освітній ступінь «бакалавр»
спеціальність: 122 – Комп'ютерні науки
освітньо-професійна програма – Комп'ютерні науки

«ЗАТВЕРДЖУЮ»
В.о. завідувача кафедри
_____ Н.В. Дзюбановська
«____» _____20__р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
ТОВПИШКУ Данилу Ростиславовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи:
Застосунок обробки природної мови на основі легких трансформерних моделей для класифікації текстів / Application for Natural Language Processing based on Lightweight Transformer Models for Text Classification

керівник роботи к.т.н., доцент М. З. Домбровський
затверджені наказом по університету від 01 грудня 2025 року № 894.
2. Строк подання студентом закінченої кваліфікаційної роботи 25 травня 2026 р.
3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література, офіційна документація до програмних бібліотек.
4. Основні питання, які потрібно розробити
проаналізувати предметну область та еволюцію методів класифікації текстів;
- виконати огляд архітектур малих мовних моделей та специфіки обробки україномовних текстів;
- сформулювати постановку задачі дослідження;
- спроєктувати об'єктно-орієнтовану архітектуру програмного застосунку;
- розробити математичне та алгоритмічне забезпечення системи;
- реалізувати програмний засіб мовою Python;
- провести експериментальне дослідження швидкодії формування логічного висновку моделі;
- розробити клієнтський інтерфейс користувача та провести тестування програмного засобу.
5. Перелік графічного матеріалу у роботі
- Перелік графічного матеріалу у роботі
- Діаграма варіантів використання системи класифікації текстів
- Діаграма класів програмного застосунку
- Схема технологічного стеку програмної системи
- Графічний інтерфейс результатів роботи системи

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 01 грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Огляд літературних джерел відповідно до предметної області та складання плану роботи	до 01.01.2026 р.	
2	Написання 1 розділу кваліфікаційної роботи	до 01.02.2026 р.	
3	Написання 2 розділу кваліфікаційної роботи	до 15.03.2026 р.	
4	Написання 3 розділу кваліфікаційної роботи	до 01.05.2026 р.	
5	Представлення попереднього варіанту кваліфікаційної роботи, перевірка та внесення змін керівником	до 15.05.2026 р.	
6	Опрацювання зауважень та представлення завершеного варіанту кваліфікаційної роботи. Підготовка супроводжуючих документів	до 25.05.2026 р.	
7	Перевірка кваліфікаційної роботи на оригінальність тексту	до 30.05.2026 р.	
8	Оформлення кваліфікаційної роботи та отримання допуску до захисту	до 10.06.2026 р.	
9	Подання кваліфікаційної роботи до захисту на засіданні атестаційної комісії	до 10.06.2026 р.	

Студент _____ Д. Р. Товпишко

підпис

Керівник роботи _____ к.т.н., доцент М. З. Домбровський

підпис

АНОТАЦІЯ

Кваліфікаційна робота на тему «Застосунок обробки природної мови на основі легких трансформерних моделей для класифікації текстів» на здобуття освітнього ступеня «Бакалавр» зі спеціальності 122 «Комп'ютерні науки» освітньої програми «Комп'ютерні науки» написана обсягом 57 сторінок і містить 7 рисунків, 1 таблиця, 2 додатки та 31 використаних джерел.

Об'єктом дослідження є процес автоматичної класифікації природномовних текстів. Предметом дослідження є методи, алгоритми оптимізації та програмні засоби класифікації текстів на основі архітектур легких трансформерних моделей.

Метою даної кваліфікаційної роботи є розробка та тестування програмного застосунку для класифікації україномовних текстів з використанням оптимізованих трансформерних моделей для забезпечення високої швидкодії на пристроях з обмеженими ресурсами.

У роботі проведено аналіз архітектур трансформерів, методів дистиляції знань та квантування. Реалізовано програмний конвеєр на базі моделі BERT, оптимізований за допомогою середовища ONNX Runtime. Для взаємодії з користувачем розроблено веб-інтерфейс на основі бібліотеки Gradio.

Результати експериментального дослідження підтвердили, що застосування графової оптимізації дозволило прискорити процес формування логічного висновку у 4,69 рази, знизивши середню затримку на запит до 44,98 мс без втрати точності класифікації.

Ключові слова: ОБРОБКА ПРИРОДНОЇ МОВИ, ТРАНСФОРМЕРНІ МОДЕЛІ, BERT, ONNX RUNTIME, ОПТИМІЗАЦІЯ ЛОГІЧНОГО ВИСНОВКУ, КЛАСИФІКАЦІЯ ТЕКСТІВ, НЕЙРОННІ МЕРЕЖІ.

ANNOTATION

The qualification thesis on the topic ‘Application for Natural Language Processing based on Lightweight Transformer Models for Text Classification’ for obtaining the Bachelor’s degree in specialty 122 “Computer Science” of the educational program “Computer Science” consists of 57 pages and contains 7 figures, 1 table, 2 appendices and 31 references.

The object of the research is the process of automatic classification of natural language texts. The subject of the research is the methods, optimisation algorithms and software tools for text classification based on lightweight transformer model architectures.

The aim of this qualification thesis is to develop and test a software application for classifying Ukrainian-language texts using optimised Transformer models to ensure high performance on devices with limited resources.

The thesis analyses transformer architectures, knowledge distillation methods and quantisation. A software pipeline based on the BERT model, optimised using the ONNX Runtime environment, has been implemented. A web interface based on the Gradio library has been developed for user interaction.

The results of the experimental study confirmed that the application of graph optimisation accelerated the process of forming a logical conclusion by a factor of 4.69, reducing the average latency per query to 44.98 ms without any loss of classification accuracy.

Keywords: NATURAL LANGUAGE PROCESSING, TRANSFORMER MODELS, BERT, ONNX RUNTIME, LOGICAL INFERENCE OPTIMISATION, TEXT CLASSIFICATION, NEURAL NETWORKS.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API (Application Programming Interface) – інтерфейс прикладного програмування.

BERT (Bidirectional Encoder Representations from Transformers) – двоспрямована кодувальна модель на основі архітектури Transformer.

CNN (Convolutional Neural Network) – згорткова нейронна мережа.

CPU (Central Processing Unit) – центральний процесор.

F1-score (F1-measure) – F1-міра, гармонійне середнє між точністю (precision) та повнотою (recall).

GPU (Graphics Processing Unit) – графічний процесор.

JSON (JavaScript Object Notation) – текстовий формат обміну даними.

LLM (Large Language Model) – велика мовна модель.

ML (Machine Learning) – машинне навчання.

NLP (Natural Language Processing) – обробка природної мови.

ONNX (Open Neural Network Exchange) – відкритий формат для представлення моделей машинного навчання.

REST (Representational State Transfer) – архітектурний стиль взаємодії компонентів розподіленого застосунку.

RNN (Recurrent Neural Network) – рекурентна нейронна мережа.

SLM (Small Language Model) – мала мовна модель.

SVM (Support Vector Machine) – метод опорних векторів.

TF-IDF (Term Frequency-Inverse Document Frequency) – статистична міра ваги слова у тексті.

UI (User Interface) – інтерфейс користувача.

.

ВСТУП

Актуальність теми. Сучасний етап розвитку інформаційних технологій характеризується стрімким впровадженням систем штучного інтелекту та обробки природної мови (NLP) у різноманітні сфери людської діяльності. Поява великих мовних моделей (LLM) на базі архітектури Transformer стала каталізатором розвитку галузі. Проте використання масивних моделей супроводжується значним збільшенням вимог до обчислювальних ресурсів та енергоспоживання, що створює перешкоди для їх розгортання в середовищах з обмеженими ресурсами. У зв'язку з цим, актуальним є використання малих мовних моделей (Small Language Models), які здатні забезпечити оптимальний баланс між точністю прогнозування та ефективністю. Вирішення цих завдань у контексті обробки україномовних текстів є особливо важливим для створення швидких та автономних інформаційних систем, таких як моніторинг новин чи виявлення токсичного контенту.

Метою кваліфікаційної роботи є розробка та тестування застосунку обробки природної мови для класифікації україномовних текстів з використанням легких трансформерних моделей для забезпечення швидкодії на пристроях з обмеженими обчислювальними ресурсами.

Для досягнення поставленої мети було сформульовано та вирішено такі завдання:

1. Проаналізувати еволюцію методів обробки природної мови, архітектурні особливості легких трансформерів (DistilBERT, ALBERT) та специфіку роботи з україномовними наборами даних.

2. Дослідити математичний апарат механізмів внутрішньої уваги та алгоритмічні основи компресії нейронних мереж (дистиляція знань, квантування).

3. Спроєктувати мікросервісну архітектуру застосунку, здійснити вибір технологічного стека та інтегрувати середовище оптимізованого висновування ONNX Runtime.

4. Програмно реалізувати застосунок, розробити інтерфейс користувача та провести експериментальну оцінку якості класифікації та продуктивності системи.

Об’єкт дослідження – процеси збору, представлення, обробки, зберігання, передачі та доступу до інформації в комп’ютерних системах.

Предмет дослідження – методи, алгоритми оптимізації та програмні засоби класифікації текстів на основі архітектур легких трансформерних моделей у процесах класифікації текстів природної мови.

Методи дослідження. Для розв’язання поставлених завдань у роботі використано: методи системного та порівняльного аналізу (для вибору архітектур моделей та технологічного стека); математичне моделювання (для опису механізмів уваги та функцій втрат); об'єктно-орієнтоване проєктування (для розробки архітектури програмного застосунку); емпіричні методи (експериментальне тестування для розрахунку метрик ефективності та продуктивності моделі).

Практичне значення одержаних результатів. Розроблений програмний застосунок на базі фреймворків FastAPI [1] та Gradio [2] з використанням ONNX Runtime може бути впроваджений як автономний модуль у системи модерації контенту, аналізу відгуків клієнтів або фільтрації спаму. Використання оптимізованих моделей дозволяє розгортати систему на стандартних центральних процесорах (CPU) без необхідності залучення дорогих графічних прискорювачів (GPU), зберігаючи при цьому високу точність класифікації.

Результати кваліфікаційної роботи апробовані та опубліковані в матеріалах Міжнародної мультидисциплінарної інтернет-конференції «Світ наукових досліджень», м. Ополе, Польща (Додаток А).

1 АНАЛІТИЧНИЙ ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ТА МЕТОДІВ КЛАСИФІКАЦІЇ ТЕКСТІВ

1.1 Еволюція методів класифікації текстів

Класифікація текстів є однією з фундаментальних задач обробки природної мови (Natural Language Processing, NLP), яка полягає у віднесенні текстового документа до однієї або кількох заздалегідь визначених категорій. Завдяки стрімкому зростанню обсягів неструктурованих текстових даних в інтернеті, потреба в автоматизації їх сортування, фільтрації та аналізу стала критичною. Історичний розвиток методів класифікації текстів можна умовно поділити на кілька ключових етапів: від простих систем на основі правил до сучасних архітектур на базі великих та малих трансформерних моделей.

Першим етапом розвитку систем класифікації були підходи, засновані на правилах та евристиках. У таких системах експерти предметної області вручну формували набори лексичних і синтаксичних правил (наприклад, регулярні вирази), за якими текст відносився до певної категорії. Незважаючи на високу точність у вузькоспеціалізованих предметних областях, цей підхід мав суттєві недоліки: неможливість масштабування, високу вартість розробки та підтримки, а також низьку адаптивність до нових даних чи змін у мовних конструкціях.

Наступним кроком стала епоха традиційного машинного навчання. У цей період задачі NLP почали розв'язуватися за допомогою статистичних алгоритмів. Оскільки алгоритми машинного навчання не здатні безпосередньо обробляти сирий текст, виникла потреба у математичному представленні слів. Найпопулярнішим методом векторизації стала модель «мішка слів» (BoW), яка ігнорує граматику та порядок слів, залишаючи лише їхню частотність. Для покращення якості класифікації BoW часто комбінувався з метрикою TF-IDF, яка дозволяла знизити вагу загальновживаних слів та підкреслити специфічні для конкретного документа терміни:

$$TFIDF(t, d, D) = TF(t, d) \cdot IDF(t, D) \quad (1.1)$$

де *t* це частота терміну у документі *d*;

IDF(t, D) – обернена частота документів, що містять цей термін у всій збірці текстів *D*.

На основі таких векторних представлень успішно застосовувалися класичні алгоритми: Наївний баєсівський класифікатор (Naive Bayes), метод опорних векторів (SVM), логістична регресія та випадкові ліси (Random Forest). Хоча ці методи продемонстрували значний стрибок у продуктивності порівняно з експертними системами, вони страждали від «прокляття розмірності» (curse of dimensionality) та повної втрати семантичного і контекстуального зв'язку між словами у реченні.

Подолати проблему семантичного контексту вдалося з появою методів глибинного навчання та щільних векторних представлень слів (Word Embeddings). Моделі Word2Vec [3] та GloVe [4] дозволили відображати слова у щільний векторний простір меншої розмірності (зазвичай 100-300 вимірів), де відстань між векторами відображала семантичну близькість слів. На базі цих вбудовувань почали активно застосовуватися рекурентні нейронні мережі (RNN) та їхні модифікації: мережі довгої короткострокової пам'яті (LSTM) і вентильні рекурентні блоки (GRU). На відміну від традиційних моделей, RNN здатні обробляти текст як послідовність, зберігаючи інформацію про попередні слова завдяки внутрішньому стану. Паралельно для задачі класифікації текстів також адаптувалися згорткові нейронні мережі (CNN) [5], які виявилися ефективними для виділення локальних *n*-грамних ознак та швидкої класифікації. Проте, архітектури RNN/LSTM мають обмеження щодо паралелізації обчислень та стикаються з проблемою забування контексту на дуже довгих послідовностях.

Справжньою революцією в NLP стала публікація у 2017 році статті «Attention is All You Need» [6], яка представила архітектуру Transformer. Відмовившись від рекурентності на користь механізму самоуваги, трансформери дозволили моделі одночасно аналізувати всі слова у вхідній послідовності та встановлювати залежності між ними незалежно від їхньої позиції в тексті. Це вирішило проблему

довгострокових залежностей та відкрило шлях до масової паралелізації обчислень на графічних процесорах (GPU).

На базі архітектури Transformer з'явилися гігантські попередньо натреновані мовні моделі, такі як BERT [7] (Bidirectional Encoder Representations from Transformers), RoBERTa, XLNet та інші. BERT змінив парадигму класифікації: замість навчання моделі з нуля на обмеженому наборі даних, використовується концепція Transfer Learning. Модель спочатку проходить етап неконтрольованого навчання (Pre-training) на терабайтах текстових даних для розуміння структури мови, а потім донавчається на відносно невеликій розміченій вибірці для конкретної задачі класифікації. Цей підхід забезпечив безпрецедентну точність.

Однак, незважаючи на передові результати сучасного рівня, класичні моделі BERT та їхні аналоги мають сотні мільйонів або навіть мільярди параметрів. Наприклад, базова версія BERT (BERT-Base) містить близько 110 млн параметрів і вимагає понад 400 МБ пам'яті для зберігання ваг. Це робить їхнє використання у реальних виробничих умовах експлуатації, мобільних пристроях, IoT-системах або на серверах з обмеженими обчислювальними ресурсами вкрай складним, повільним та економічно невигідним.

Відповіддю на ці виклики стала еволюція у напрямку компресії моделей та створення легких трансформерних архітектур або малих мовних моделей (SLM). Завдяки застосуванню методів дистиляції знань, квантування та прунінгу, дослідникам вдалося створити такі моделі, як DistilBERT [8], ALBERT, MobileBERT та TinyBERT [11]. Вони зберігають до 95-97% точності своїх «великих» аналогів, при цьому зменшуючи розмір та час логічного висновку у кілька разів. Використання саме таких легковагових архітектур для задачі класифікації є найбільш раціональним компромісом між обчислювальною ефективністю та семантичною точністю, що і є основним предметом дослідження у даній кваліфікаційній роботі.

1.2 Аналіз існуючих рішень та технологічних викликів обробки україномовних текстів

Розвиток архітектур на базі Transformer призвів до появи моделей з експоненціально зростаючою кількістю параметрів, що забезпечує високу якість обробки природної мови, проте обмежує їх використання на пристроях із низькою обчислювальною потужністю. У відповідь на цей виклик сформувався напрям розробки малих мовних моделей, архітектурні рішення яких спрямовані на максимізацію ефективності при мінімальному використанні пам'яті та енергоресурсів.

Основним завданням при проєктуванні архітектур SLM є подолання квадратичної складності механізму самоуваги відносно довжини вхідної послідовності та зменшення розміру матриць ваг. На архітектурному рівні це досягається декількома основними підходами: факторизацією параметрів, спільним використанням шарів та введенням спеціальних «вузьких місць».

Однією з перших успішних спроб архітектурної оптимізації стала модель ALBERT (A Lite BERT). У цій моделі реалізовано два ключових рішення: декомпозиція матриць вбудовувань та спільне використання параметрів між шарами. Замість того, щоб проєктувати вектори токенів безпосередньо у простір прихованих шарів великої розмірності H, спочатку виконується проєкція у простір меншої розмірності E. Це дозволяє значно зменшити кількість параметрів на початковому етапі обробки. Крім того, у ALBERT всі шари трансформера використовують однакові ваги, що радикально зменшує обсяг пам'яті, необхідний для зберігання моделі, хоча обчислювальна складність під час виконання залишається незмінною.

Для мобільних пристроїв та вбудованих систем було розроблено архітектуру MobileBERT. Вона базується на концепції «учитель-учень» та використовує інноваційну структуру шарів з інвертованими вузькими місцями. У MobileBERT кожний шар містить додаткові блоки лінійної проекції, які стискають дані перед подачею на механізм уваги та розширюють їх після обробки. Це дозволяє моделі

імітувати поведінку великої мережі BERT-Large, маючи при цьому значно меншу кількість параметрів та в рази вищу швидкодію обчислень на мобільних процесорах.

Окрему групу SLM складають моделі, розроблені шляхом дистиляції знань на рівні прихованих станів, такі як TinyBERT. На відміну від стандартної дистиляції, де «учень» вчиться лише на вихідних ймовірностях «вчителя», архітектура TinyBERT передбачає передачу знань безпосередньо з матриць уваги та результатів кожного проміжного шару. Це дозволяє створити надзвичайно компактні моделі (наприклад, 4 або 6 шарів замість 12), які зберігають здатність до складного контекстуального аналізу.

Сучасним трендом у розробці SLM є створення моделей «з нуля» на високоякісних синтетичних наборах даних, що дозволяє досягти вражаючих результатів навіть при невеликій кількості параметрів. Прикладом такої архітектури є серія моделей Phi від Microsoft Research. Модель Phi-1.5, маючи лише 1.3 мільярда параметрів, демонструє здатність до логічного мислення та розуміння мови на рівні моделей, що в 5-10 разів більші за неї [12]. Це досягається за рахунок ретельного відбору навчальних даних, які за структурою нагадують підручники, та оптимізації структури багатоголової уваги.

Іншим перспективним напрямком є модель TinyLlama, яка реалізує архітектуру Llama в компактному варіанті на 1.1 мільярда параметрів. Вона використовує сучасні техніки, такі як Rotary Positional Embeddings (RoPE) та активаційну функцію SwiGLU, що дозволяє моделі ефективніше працювати з контекстом порівняно з класичними BERT-подібними архітектурами [13]. Завдяки відкритому коду та сумісності з інструментами квантування, такі моделі стають ідеальним фундаментом для створення застосунків класифікації текстів, що працюють локально на пристрої користувача.

Таким чином, аналіз архітектур SLM показує, що ефективність класифікації текстів залежить не лише від кількості параметрів, а й від стратегії їх організації. Використання механізмів дистиляції та структурних оптимізацій дозволяє

інтегрувати потужні засоби обробки природної мови у легковагові програмні рішення, що відповідає меті даної роботи.

Ефективність застосування легких трансформерних моделей для класифікації текстів суттєво залежить від лінгвістичних особливостей мови, на якій здійснюється обробка [15]. На відміну від англійської мови, яка є аналітичною за своєю структурою, українська мова належить до синтетичних мов із розвиненою системою словозміни та багатою морфологією. Це створює низку специфічних викликів для систем автоматичної обробки природної мови (NLP), які необхідно враховувати при розробці інтелектуальних застосунків.

Однією з головних особливостей української мови є високий ступінь флективності. Наявність семи відмінків, категорій роду, числа та виду призводить до того, що одне й те саме слово може мати десятки різних словоформ. Для класичних моделей класифікації це спричиняє проблему розрідженості даних: різні форми однієї лексеми сприймаються алгоритмом як окремі, непов'язані одиниці, що збільшує розмір словника та вимагає більших обсягів навчальних вибірок. Хоча трансформерні моделі використовують субтокени, що частково вирішує проблему морфологічної складності, специфічні префікси та суфікси української мови часто вимагають спеціалізованих словників токенізації, натренованих саме на збірках українських текстів.

Іншим важливим аспектом є відносно вільний порядок слів у реченні. В українській мові синтаксична роль слова визначається переважно його морфологічними ознаками (закінченнями), а не позицією в реченні, як в англійській. Це означає, що механізм уваги у трансформерних моделях повинен ефективно виявляти довгострокові залежності між словами, які можуть бути розділені іншими членами речення. Для легких моделей з обмеженою кількістю шарів це стає критичним завданням, оскільки вони мають меншу «глибину» для захоплення таких складних зв'язків.

Крім лінгвістичних труднощів, розробка NLP-систем для української мови стикається з проблемою обмеженості цифрових ресурсів (low-resource language). Попри значний прогрес останніх років, кількість розмічених наборів даних для

задач класифікації (аналіз тональності, категоризація новин, детекція токсичного контенту) залишається значно меншою порівняно з англійською чи іншими поширеними європейськими мовами.

У ході дослідження було проаналізовано існуючі набори даних для української мови, які можуть бути використані для навчання та тестування систем класифікації. До основних джерел належать:

- збірка текстів UA-GEC – набір даних, що містить анотовані граматичні та семантичні помилки, але також може бути адаптований для задач лінгвістичного аналізу [16];
- набори даних для аналізу тональності, сформовані на основі відгуків користувачів інтернет-магазинів (наприклад, Rozetka) або соціальних мереж. Такі датасети часто містять специфічний лексикон, суржик та емоційно забарвлені конструкції, що робить їх складними для класифікації;
- УкрВікіКорпус – велика збірка текстів з української Вікіпедії, який зазвичай використовується для попереднього навчання моделей, але також містить мітки категорій (статті за темами), що дозволяє використовувати його для тематичної класифікації;
- спеціалізовані новинні датасети, що містять класифіковані заголовки та повні тексти новин з українських медіа-ресурсів, розподілені за рубриками (політика, спорт, технології тощо);

Важливим етапом у подоланні «мовної нерівності» стала поява попередньо натренованих моделей спеціально для української мови. До найбільш ефективних належать UKR-RoBERTa [18] та модифікації BERT, натреновані на великій збірці українських текстів обсягом у десятки гігабайт. Використання таких моделей як бази для подальшої дистиляції або квантування дозволяє створювати легкі застосунки, які «розуміють» нюанси української граматики та семантики.

Для реалізації практичної частини роботи особлива увага приділяється методам передобробки (preprocessing) україномовних текстів. Досліджується доцільність застосування лематизації та видалення стоп-слів у контексті трансформерних моделей. На відміну від класичних статистичних методів, де

лематизація була обов'язковою, трансформери часто демонструють кращі результати на «сирих» текстах, зберігаючи морфологічні нюанси, що можуть бути важливими для визначення контексту. Проте, для легких моделей (SLM) мінімізація вхідного словника через розумну фільтрацію може надати суттєвий приріст у швидкості без втрати якості.

Таким чином, специфіка української мови вимагає виваженого підходу до вибору архітектури та підготовки даних. Висока флективність та вільний порядок слів ставлять підвищені вимоги до здатності моделі виділяти ознаки, що робить перевірку легких трансформерів на україномовних датасетах актуальним науково-технічним завданням.

1.3 Задачі обробки природної мови на основі легких трансформерних моделей

Перехід від аналітичного огляду існуючих архітектур та лінгвістичних особливостей обробки природної мови до безпосереднього проектування системи вимагає строгої математичної та інженерної формалізації. Це дозволяє визначити чіткі критерії ефективності розроблюваного застосунку та окреслити межі дослідження.

У рамках даної роботи розглянуто задачу багатокласової класифікації україномовних текстових повідомлень за п'ятибальною шкалою тональності з одночасною програмно-апаратною оптимізацією обчислювального графа нейронної мережі для виконання на центральному процесорі.

Математично задачу класифікації текстів можна сформулювати наступним чином. Нехай задано вхідний простір неструктурованих текстових послідовностей $T = \{t_1, t_2, \dots, t_N\}$, де кожне повідомлення представлене у кодуванні UTF-8 і складається з певної кількості субтокенів, отриманих після етапу токенізації. Також визначено дискретну множину цільових класів тональності:

$$C = \{c_1, c_2, \dots, c_K\} \tag{1.2}$$

де $K = 5$ – кількість категорій, що відповідають оцінкам від 1 до 5 зірок.

Кінцевою метою інтелектуального аналізу є побудова стійкої моделі відображення F, яка зіставляє довільну текстову послідовність із найбільш релевантним класом тональності: $F: T \to C$

Процес класифікації кожного тексту зводиться до того, що нейронна мережа розраховує ймовірність його належності до кожної з можливих категорій, після чого автоматично обирає клас із найвищим показником.

Особливістю цього дослідження є врахування жорстких вимог до швидкості роботи системи та економії апаратних ресурсів. Оскільки застосунок планується розгортати на звичайних центральних процесорах (CPU) без залучення графічних прискорювачів, ключовим завданням є максимальне прискорення математичних обчислень.

Якщо позначити час обробки одного запиту базовою моделлю у середовищі PyTorch як τ_{base}, то суть оптимізації полягає у переведенні її структури у формат ONNX. Це дозволить побудувати ефективніший конвеєр із часом виконання τ_{opt}, який задовольняє наступну систему умов:

$$\begin{cases} \tau_{\text{opt}} \to \min \\ \tau_{\text{opt}} \leq \beta \cdot \tau_{\text{base}} \\ Accuracy_{\text{opt}} \approx Accuracy_{\text{base}} \end{cases} \quad (1.3)$$

де β– цільовий коефіцієнт зниження часових витрат;

$Accuracy_{\text{base}}$ та $Accuracy_{\text{opt}}$ – точність класифікації базової та оптимізованої моделей відповідно на верифікаційній збірці даних.

Для досягнення поставленої мети та виконання системи обмежень (1.3) у роботі необхідно послідовно вирішити такі завдання:

- дослідити внутрішню структуру та особливості архітектури багатомовних моделей трансформерів, здатних ефективно обробляти специфічний синтаксис української мови;

- спроєктувати модульну об'єктно-орієнтовану архітектуру програмного застосунку, яка відокремлює етапи підготовки лінгвістичних даних від ядра обчислень;
- реалізувати алгоритми очищення тексту та субсимвольної токенізації для усунення ефекту розрідженості ознак;
- інтегрувати середовище ONNX Runtime, задіявши низькорівневі оптимізації обчислювального графа на рівні злиття операторів;
- побудувати графічний інтерфейс користувача для інтерактивної взаємодії із системою та провести порівняльний бенчмаркінг характеристик роботи конвеєрів.

2 АЛГОРИТМІЧНЕ, МАТЕМАТИЧНЕ ТА ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

2.1 Малі мовні моделі трансформерів з механізмом внутрішньої уваги

В основі розроблюваного застосунку лежать архітектури малих мовних моделей (Small Language Models), концептуально заснована на трансформерних моделях, проте містять низку математичних та алгоритмічних оптимізацій рівня State-of-the-Art (SOTA). Фундаментальним апаратом, що забезпечує здатність таких моделей до аналізу контексту, є модифікований механізм внутрішньої уваги.

Процес обробки тексту розпочинається з перетворення вхідної послідовності токенів у матрицю векторних вбудовувань $X \in R^{n \times d}$, де n – довжина послідовності, а d – розмірність прихованого простору. Для обчислення уваги матриця X проектується у матрицю запитів Q, ключів K та значень V. Суть механізму масштабованого скалярного добутку уваги зводиться до обчислення міри подібності між запитами та ключами, що описується наступним рівнянням [6]:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.1}$$

У класичних архітектурах застосовувався механізм багатоголової уваги (Multi-Head Attention, MHA). Проте, для мінімізації споживання пам’яті під час обчислювального процесу в сучасних легких трансформерах (наприклад, архітектурах Llama) використовується згрупована увага (GQA). У цьому підході кількість голів для ключів (K) та значень (V) значно менша за кількість голів для запитів (Q). Декілька голів запитів спільно використовують одну й ту саму голову ключів та значень, що радикально зменшує розмір KV-кешу під час генерації або класифікації, зберігаючи при цьому точність, наближену до MHA.

Оскільки механізм самоуваги обробляє всі токени паралельно, йому необхідна інформація про порядок слів. Замість застарілого абсолютного синусоїдального кодування, у сучасних моделях імплементовано ротаційне

позиційне кодування (RoPE). Цей метод кодує абсолютну позицію токена за допомогою матриці обертання, що дозволяє механізму уваги природним чином враховувати відносну відстань між словами. Математично, для токена на позиції m, вектори q та k розбиваються на 2D-пари, до яких застосовується обертання на кут $m\theta_i$ у комплексній площині [19]:

$$(q_{m,2i}q_{m,2i+1}) = \begin{pmatrix}\cos(m\theta_i) & -\sin(m\theta_i)\\ \sin(m\theta_i) & \cos(m\theta_i)\end{pmatrix}(x_{m,2i}x_{m,2i+1}) \qquad (2.2)$$

де $\theta_i = 10000^{-2i/d}$;

$i \in [0, d/2-1]$.

Такий підхід є критично важливим для мов із вільним порядком слів, таких як українська.

Наступним кроком оптимізації архітектури є заміна класичного шару нормалізації на обчислювально легшу нормалізацію середньоквадратичного значення (RMSNorm). RMSNorm ігнорує операцію центрування середнього значення, спираючись лише на дисперсію, що прискорює обчислення без втрати стабільності градієнтів [20]:

$$\mathrm{RMSNorm}(\mathrm{x}) = \frac{\mathrm{x}}{\sqrt{\frac{1}{\mathrm{d}}\sum_{\mathrm{i}=1}^{\mathrm{d}}\mathrm{x}_\mathrm{i}^2 + \epsilon}} \odot \gamma \qquad (2.3)$$

де ϵ – мала константа для стабільності;

γ – параметр масштабування, що навчається;

$\odot$ – поелементне множення.

Мережа прямого поширення (Position-wise Feed-Forward Network, FFN) у кожному блоці трансформера також зазнала змін. Сучасним стандартом є використання активаційної функції SwiGLU (Swish Gated Linear Unit) замість

класичної ReLU. SwiGLU використовує механізм шлюзування на основі функції *Swish*, що забезпечує більш плавні градієнти та кращу виразну здатність моделі [21]:

$$SwiGLU(x) = \left(xW_1 \odot \sigma(\beta xW_1)\right) \odot (xW_2)W_3 \quad (2.4)$$

де W_1, W_2, W_3 – матриці лінійних проекцій;

σ – сигмоїдна функція.

Нарешті, для практичної реалізації швидкої класифікації на апаратному рівні застосовується алгоритм FlashAttention. Це IO-обізнаний (Input/Output-aware) алгоритм, який оптимізує доступ до пам'яті графічного (GPU) або центрального (CPU) процесора. FlashAttention розбиває матриці *Q*, *K*, *V* на блоки (tiling) і виконує обчислення softmax локально у швидкій статичній пам'яті (SRAM), уникаючи ресурсомісткого запису проміжних матриць розміром $n \times n$ у повільну глобальну пам'ять (HBM) [22]. Це перетворює квадратичну складність доступу до пам'яті $O(n^2)$ на лінійну $O(n)$, що робить легкі трансформери повноцінно придатними для роботи на пристроях з обмеженими ресурсами.

Для проведення обчислювальних експериментів у рамках даної роботи було задіяно попередньо натреновану трансформерну архітектуру bert-base-multilingual-uncased-sentiment. Оскільки цей інструмент є безпосереднім математичним ядром розроблюваного застосунку, необхідно детально формалізувати його внутрішню топологію та конфігурацію параметрів, що визначають підсумкове навантаження на підсистему пам'яті центрального процесора.

За своєю архітектурною топологією модель являє собою багатошаровий двоспрямований кодувальник трансформера. Конфігурація системи включає 12 послідовних прихованих шарів (Transformer blocks). Кожен шар містить у собі індивідуальний блок багатоголової внутрішньої уваги, що складається з 12 незалежних голів уваги. Розмірність прихованого векторного простору становить $H = 768$, а розмірність внутрішнього повнозв'язного шару прямого поширення

дорівнює $D_{ff} = 3072$. Загальний обсяг ваг, що підлягають обчисленню під час прямого поширення, становить приблизно 168 мільйонів параметрів.

Процес обробки україномовного тексту всередині цієї архітектури підпорядкований суворим перетворенням. Спочатку вхідна послідовність символів надходить до спеціалізованого багатомовного токенізатора, який використовує алгоритм WordPiece. Словник моделі розширено до 105 879 унікальних субтокенів, що дозволяє мінімізувати появу невідомих символів при роботі з синтетичними мовними системами.

Конфігурація uncased визначає, що на етапі передобробки токенізатор примусово згладжує регістр тексту, переводячи всі літери у нижній регістр. Це дозволило розробникам архітектури зберегти високу щільність векторного простору без роздування розміру фінальних матриць вбудовувань. Узагальнену структурну схему конвеєра обробки даних та послідовної архітектури шарів кодувальника Трансформера разом із фінальним класифікаційним рівнем наведено на рисунку 2.1.

Важливим фактором вибору даної моделі є її інформаційний тренувальний базис, який забезпечує високу здатність до узагальнення контексту. Модель пройшла два послідовні етапи навчання:

- етап неконтрольованого базового навчання. модель була натренована корпорацією Google на масивному корпусі текстів світової Вікіпедії, що охоплює 104 мовні домени, включаючи український сегмент. Основними задачами навчання були масковане мовне моделювання, де мережа вгадувала 15% випадково прихованих слів у контексті, та прогнозування наступного речення. Це заклало фундамент розуміння синтаксису та взаємозв'язків між словами у флективних мовних системах з вільним порядком слів;
- на етапі цільового донавчання для налаштування моделі під задачу аналізу тональності розробники використали спеціалізований багатомовний датасет відгуків користувачів. Модель навчилася відображати внутрішнє семантичне представлення класифікаційного токена [CLS] через фінальний

лінійний шар у дискретний простір п'яти цільових класів, які відповідають оцінкам рейтингу від 1 до 5 зірок.

-

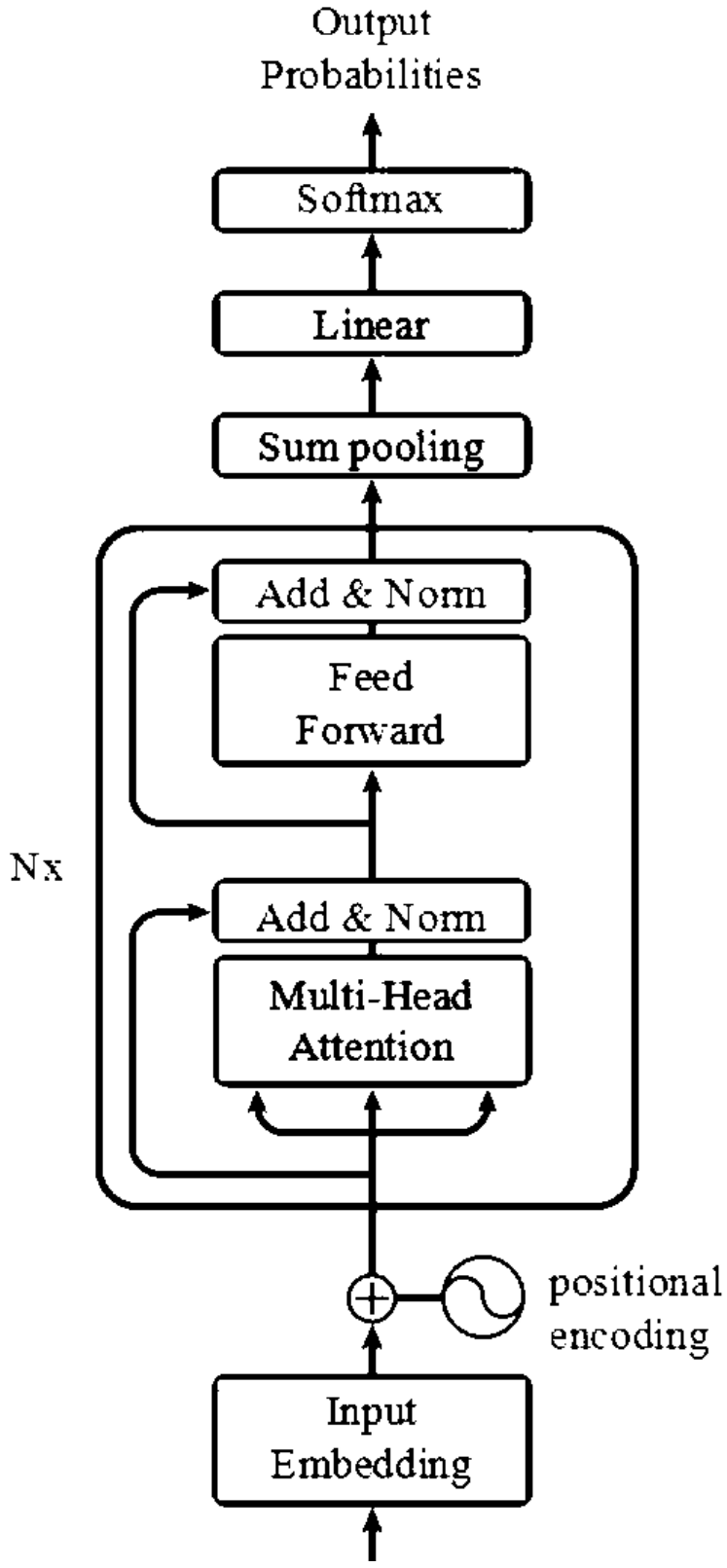

Рисунок 2.1 – Структурна схема архітектури класифікатора на основі блоків кодувальника Трансформера

2.2 Методи підвищення обчислювальної ефективності та графової оптимізації в середовищі ONNX

Використання сучасних трансформерних архітектур для задач обробки природної мови супроводжується значними вимогами до об'єму оперативної пам'яті та обчислювальних ресурсів процесора. Для адаптації великих мовних моделей до умов обмежених апаратних можливостей застосовуються спеціалізовані алгоритми компресії нейронних мереж. До основних математичних та алгоритмічних методів, що дозволяють радикально зменшити розмір моделі (створити Small Language Model) при мінімальній втраті якості класифікації, належать дистиляція знань, квантування параметрів, прунінг та графова оптимізація.

Дистиляція знань – це метод машинного навчання, заснований на парадигмі передачі знань від складної, попередньо натренованої моделі-вчителя до більш компактної моделі-учня [23]. Під час стандартної задачі класифікації текстів модель оптимізує свої ваги, орієнтуючись лише на істинні мітки класів. Проте розподіл ймовірностей, який генерує модель-вчитель, містить важливу приховану інформацію про семантичну спорідненість класів. Для згладжування цього розподілу в алгоритм активації softmax вводиться гіперпараметр температури *T*:

$$\mathrm{p_i} = \frac{\exp(\mathrm{z_i}/\mathrm{T})}{\sum_\mathrm{j} \exp(\mathrm{z_j}/\mathrm{T})} \tag{2.5}$$

де p_i – ймовірність приналежності вхідного тексту до класу *i*;

z_i – вихідний логіт (неактивоване значення) моделі для класу *i*;

T – гіперпараметр температури (при *T=1* функція працює як стандартний softmax, при *T>1* розподіл стає більш рівномірним).

Базова функція втрат L_{KD} обчислюється як зважена сума перехресної ентропії відносно істинних міток та дивергенції Кульбака-Лейблера між передбаченнями вчителя і учня:

$$L_{KD} = \alpha L_{CE}(y, \hat{y}) + (1 - \alpha)T^2 \cdot D_{KL}(p^{teacher} || p^{student}) \quad (2.6)$$

де α – коефіцієнт балансування втрат.

Однак для глибоких трансформерних моделей дистиляції лише вихідного шару недостатньо. Щоб учень краще розумів лінгвістичні закономірності, застосовується дистиляція прихованих станів та матриць уваги. Навчання прихованих шарів описується функцією середньоквадратичної помилки (MSE):

$$L_{hidden} = MSE(H^{teacher}, H^{student} W_h) \quad (2.7)$$

де $H^{teacher}$ та $H^{student}$ – матриці прихованих станів вчителя та учня відповідно;

W_h – матриця лінійної проекції, яка вирівнює розмірності векторів учня і вчителя, якщо учень має меншу кількість прихованих нейронів.

Дистиляція матриць уваги дозволяє учневі копіювати те, на яких саме словах фокусується вчитель при обробці українського тексту, що є критичним для мов із вільним порядком слів:

$$L_{attn} = \frac{1}{h} \sum_{i=1}^{h} MSE\left(A_i^{teacher}, A_i^{student}\right) \quad (2.8)$$

де A_i – матриця ваг *i*-ї голови механізму уваги;

h – кількість голів.

Квантування – це алгоритмічний процес відображення неперервних дійсних значень (наприклад, 32-бітних чисел із плаваючою комою, FP32) у дискретний набір значень меншої розрядності, зокрема 8-бітні цілі числа (INT8). Це дозволяє зменшити обсяг пам'яті, який займає модель, у 4 рази, та прискорити виконання

матричних операцій на центральному процесорі за рахунок використання апаратних інструкцій (наприклад, AVX2 або VNNI) [24].

Основним методом, що застосовується для оптимізації моделей NLP, є лінійне квантування. Воно поділяється на асиметричне та симетричне. В асиметричному квантуванні перетворення здійснюється за допомогою масштабного коефіцієнта *S* (Scale) та нульової точки *Z* (Zero-point), яка зміщує діапазон:

$$x_q = clip\left(round\left(\frac{x_f}{S}\right) + Z,\ Q_{min},\ Q_{max}\right) \tag{2.9}$$

де x_q – квантоване ціле значення;

x_f – початкове дійсне число (вага або активація);

$Q_{min},\ Q_{max}$ – межі цілочисельного діапазону (наприклад, [0, 255] для UINT8).

Більш ефективним з точки зору обчислювальної складності є симетричне квантування, де *Z = 0*, а діапазон центрується навколо нуля ($[-127,127]$ для INT8). Масштабний коефіцієнт *S* у цьому випадку обчислюється як:

$$S = \frac{\max(|x_f|)}{127} \tag{2.10}$$

У контексті розроблюваного застосунку найбільш доцільним є використання динамічного квантування. При цьому ваги нейронної мережі перетворюються у формат INT8 заздалегідь, а квантування матриць активацій (вхідних текстів) відбувається динамічно під час логічного висновку. Це дозволяє уникнути значної деградації точності моделі без необхідності проведення етапу калібрування на окремому наборі даних.

Прунінг – це метод зменшення складності нейронної мережі шляхом видалення параметрів, які мають найменший вплив на функцію втрат.

Неструктурований прунінг за величиною обнуляє окремі елементи матриць ваг, абсолютне значення яких є меншим за заданий поріг λ.

Незважаючи на те, що такий підхід утворює розріджені матриці і зменшує інформаційну ємність моделі, він не забезпечує пропорційного прискорення на звичайних процесорах через особливості доступу до пам'яті. Тому для легких трансформерів розглядається структурний прунінг. Цей підхід передбачає видалення цілих блоків нейронної мережі: конкретних голів уваги у механізмі Multi-Head Attention або цілих рядків у матрицях повнозв'язних шарів. Це фізично зменшує розмірність матричних множень (d_{model} та d_{ff}), що напряму прискорює час відгуку системи.

Окрім математичних змін самої моделі, алгоритмічною основою швидкодії є оптимізація обчислювального графа в середовищі ONNX Runtime. Коли трансформерна модель експортується у формат ONNX, вона представляється у вигляді спрямованого ациклічного графа (DAG), де вузли відповідають математичним операціям (MatMul, Add, Softmax), а ребра – тензорам даних.

ONNX Runtime виконує злиття вузлів (Node Fusion). Наприклад, операції множення матриць (MatMul) та додавання зсуву (Add) автоматично об'єднуються в єдину операцію (GEMM - General Matrix Multiply). Аналогічно, операції LayerNorm або RMSNorm, які математично складаються з кількох кроків (знаходження середнього, дисперсії, ділення), зливаються в один оптимізований виклик на рівні ядра (C++).

Комплексне застосування дистиляції для зменшення кількості шарів, квантування для використання 8-бітної арифметики та графової оптимізації ONNX для мінімізації накладних витрат формує замкнений алгоритмічний цикл. Це дозволяє розгорнути застосунок класифікації україномовних текстів у середовищі з жорстко обмеженими обчислювальними ресурсами, зберігаючи при цьому семантичну точність базових великих мовних моделей.

2.3 Інформаційне забезпечення та етапи обробки даних

Інформаційне забезпечення системи базується на використанні неструктурованих текстових даних, що надходять у форматі Unicode (кодування UTF-8). Оскільки об’єктом дослідження є україномовні тексти, система повинна коректно опрацьовувати специфічні символи кирилиці та враховувати морфологічні особливості мови. Основним джерелом даних для формування логічного висновку є вхідні речення або фрагменти тексту, які потребують перетворення у цифрові структури (тензори) для подальшої обробки нейронною мережею.

Узагальнена блок-схема конвеєра обробки наведена на рисунку 2.2. Процес обробки інформації у розробленому застосунку складається з декількох критичних етапів: лінгвістична передобробка:

- на цьому етапі виконується очищення тексту від нерелевантних символів, нормалізація пробілів та приведення всіх символів до нижнього регістру, що дозволяє зменшити розмір словника;
- субсимвольна токенізація та мапування: текст розбивається на субтокени алгоритмом BPE, після чого кожен токен зіставляється з його числовим ідентифікатором зі словника моделі. На цьому ж етапі виконується перевірка довжини послідовності з подальшим обрізанням або доповненням (padding) до заданого ліміту;
- генерація структур для ONNX Runtime: створюються вхідні тензори, що містять ідентифікатори токенів (input_ids) та бінарну маску уваги (attention_mask), яка вказує, які елементи є значущими;
- реалізація обчислювального процесу: підготовлені тензори передаються до обчислювального графа ONNX, де виконується пряме поширення через оптимізовану нейронну мережу;
- фінальна обробка результатів: до отриманих сирих логітів застосовується математична функція Softmax, що перетворює їх у ймовірнісний розподіл для виведення фінальної мітки категорії.

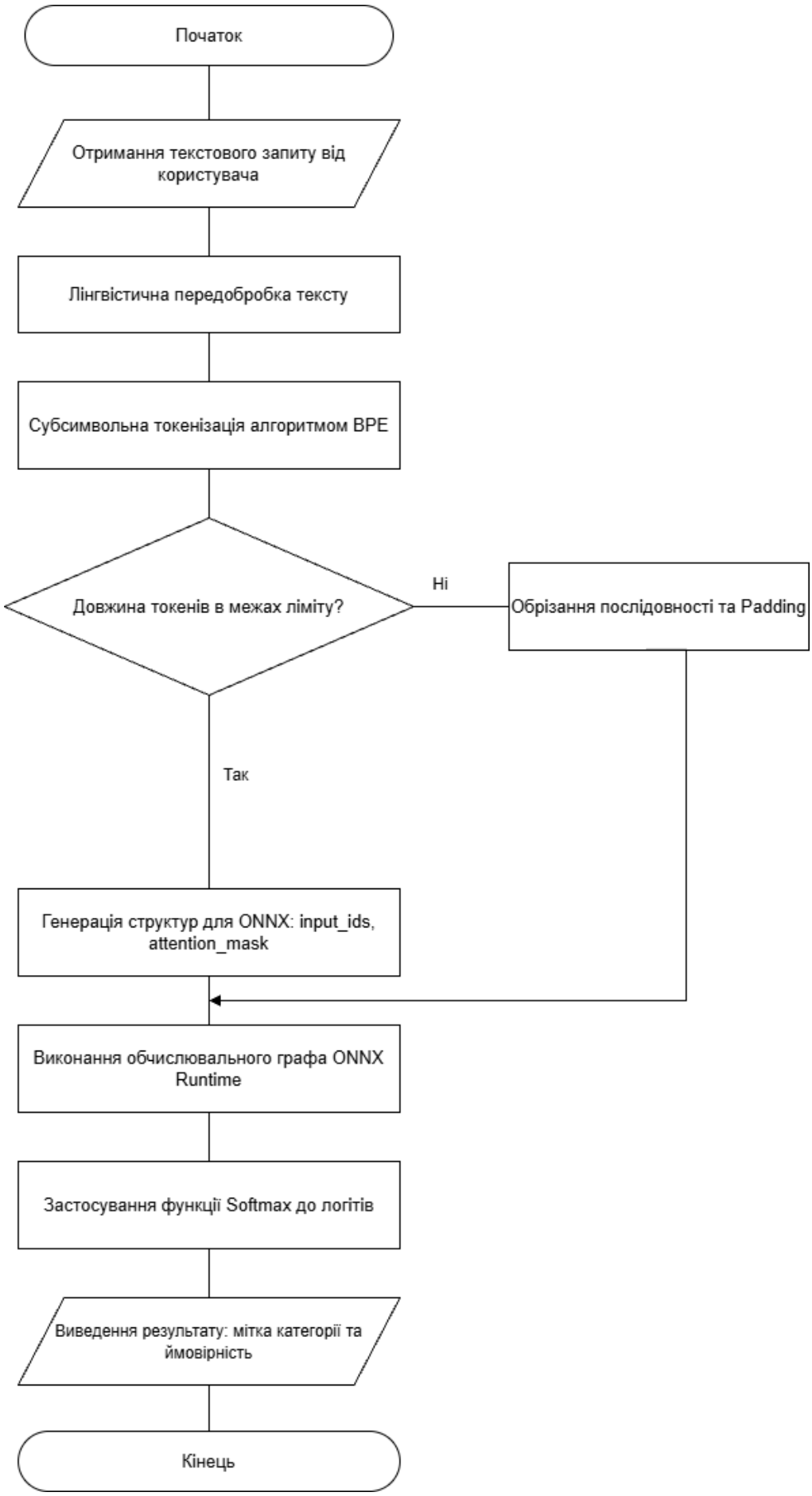

Рисунок 2.2 – Схема алгоритму функціонування обчислювального конвеєра

Такий підхід до інформаційного забезпечення гарантує цілісність передачі семантичних зв’язків від вхідного тексту до математичного графа обчислень. Використання статичних структур даних на етапі підготовки дозволяє мінімізувати накладні витрати на динамічне керування пам’яттю, що є критично важливим для забезпечення високої швидкодії системи на центральному процесорі.

2.4 Об’єктно-орієнтоване проєктування архітектури застосунку

Перехід від теоретичних алгоритмів компресії нейронних мереж до програмної реалізації вимагає чіткого проєктування структури системи. Для забезпечення модульності, масштабованості та зручності подальшої підтримки застосунку використано методологію об'єктно-орієнтованого проєктування (ООП). Моделювання архітектури та інформаційних потоків виконано за допомогою уніфікованої мови моделювання (UML).

Діаграма варіантів використання дозволяє описати функціональне призначення системи з погляду зовнішніх акторів. У розроблюваному застосунку виокремлено двох основних акторів: «Користувач» (кінцевий споживач програмного продукту) та «Середовище виконання» (зовнішній оптимізатор ONNX Runtime, який інкапсулює низькорівневі операції з процесором). Головними прецедентами для актора «Користувач» є:

- введення тексту для аналізу;
- ініціалізація процесу класифікації;
- перегляд результатів (категорія тональності та ймовірнісна впевненість моделі);
- очищення форми введення або пакетне завантаження даних (опціонально);
- «Ініціалізація процесу класифікації» включає (відношення <<include>>) ряд прихованих системних підпроцесів: завантаження квантованої моделі в оперативну пам'ять, нормалізацію тексту та токенізацію. Актор «Середовище виконання» взаємодіє виключно з прецедентом виконання

обчислювального графа, отримуючи тензори та повертаючи логіти класу. Функціональні вимоги до системи та межі її взаємодії із зовнішніми акторами наведено на діаграмі варіантів використання (рисунок 2.3).

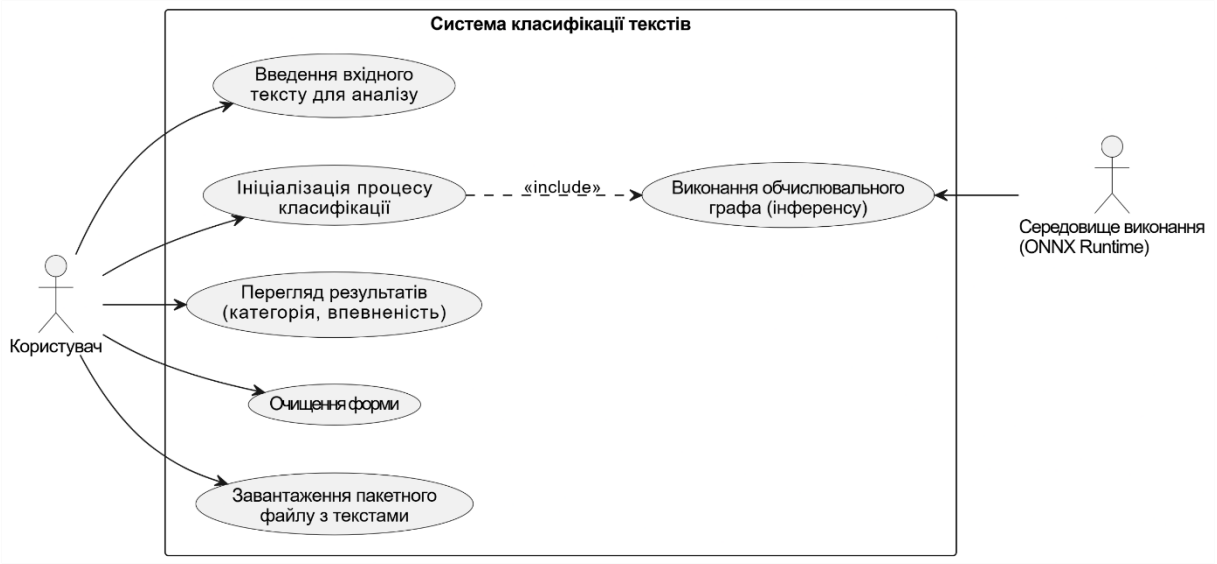

Рисунок 2.3 – Діаграма варіантів використання системи

Основою архітектури програмного застосунку є діаграма класів, яка відображає об'єктну декомпозицію системи. Архітектура побудована за шаблоном Pipeline (Конвеєр) із застосуванням патерну Singleton (Одинак) для управління життєвим циклом важкої моделі машинного навчання.

Ключовими класами розробленої системи є (рисунок 2.4):

- TextPreprocessor. Клас відповідає за очищення сирих даних. Містить публічний метод clean_text(), який використовує регулярні вирази для видалення нерелевантних символів та уніфікації регістру.
- TokenizerModule. Клас, що інкапсулює логіку розбиття тексту на субтокени алгоритмом BPE. Містить атрибут vocab_size та метод encode(), який приймає рядок і повертає числові масиви input_ids та attention_mask.
- ONNXInferenceEngine. Центральний клас обчислювального ядра. Реалізує патерн Singleton, щоб уникнути повторного завантаження файлу моделі в пам'ять при кожному запиті. Зберігає екземпляр сесії ONNX (InferenceSession) та

містить метод predict(), який приймає тензори від токенізатора і виконує пряме поширення через нейронну мережу.

- PostProcessor. Допоміжний клас, який отримує сирі логіти від обчислювального ядра, застосовує до них математичну функцію softmax (метод apply_softmax()) та перетворює числові індекси у зрозумілі текстові мітки категорій (метод get_labels()).

- AppController. Клас-контролер, який пов'язує графічний інтерфейс або API з внутрішньою логікою, послідовно викликаючи методи препроцесора, токенізатора та рушія виконання.

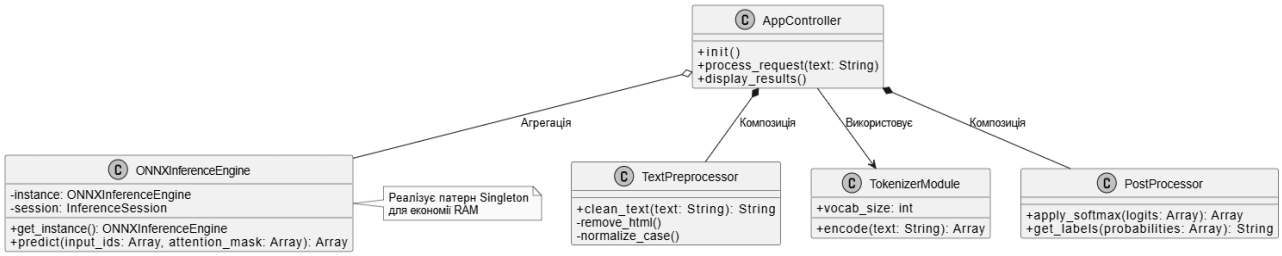

Рисунок 2.4 – Діаграма класів програмного застосунку

Відношення між класами побудовані на основі композиції та агрегації. Клас AppController жорстко контролює життєвий цикл TextPreprocessor та PostProcessor (композиція), оскільки їхнє існування поза контекстом конкретного запиту не має сенсу. Водночас ONNXInferenceEngine пов'язаний із контролером відношенням агрегації, оскільки обчислювальний рушій ініціалізується один раз під час старту застосунку і може обслуговувати різні потоки або контролери паралельно. Таке об'єктно-орієнтоване розбиття дозволяє ізолювати етап лінгвістичної обробки від етапу математичного висновування (обчислення графа). Це означає, що в майбутньому зміна мови системи (заміна токенізатора) або оновлення архітектури самої нейронної мережі (заміна ONNX файлу) не вимагатиме переписування всього програмного коду, що цілком відповідає принципам SOLID.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ЗАСТОСУНКУ ОБРОБКИ ПРИРОДНОЇ МОВИ

3.1 Архітектурне рішення і технологічний стек застосунку

Для розробки та впровадження системи класифікації текстів на основі архітектури Transformer було обрано стек технологій, що забезпечує баланс між швидкістю розробки, швидкодією функціонування моделі та гнучкістю налаштувань. Кожен компонент стеку було обрано з урахуванням специфіки задачі оптимізації NLP-моделей.

Для розробки застосунку обрано мову Python (версія 3.10+). Такий вибір зумовлений її домінуванням в екосистемі машинного навчання та наявністю зрілих бібліотек для роботи з тензорними обчисленнями. Для забезпечення відтворюваності результатів та ізоляції залежностей застосовувалося віртуальне середовище (venv), а для коректного виведення кириличних символів у консольних журналах — кодування UTF-8.

Для доступу до попередньо натренованих моделей використано інструментарій екосистеми Hugging Face, зокрема бібліотеку transformers [26]. Вона надає стандартизований інтерфейс для завантаження токенізаторів та ваг моделей, що дозволяє легко інтегрувати архітектури типу BERT у власні конвеєри обробки. Оскільки задача вимагає аналізу української мови, як базову було обрано багатомовну модель bert-base-multilingual-uncased-sentiment.

Ключовою ланкою для оптимізації та інтеграції у проєкті стала бібліотека optimum. Вона виконує роль моста між високорівневими абстракціями Hugging Face та спеціалізованими рушіями виконання. Застосування модуля optimum.onnxruntime дозволило автоматизувати процес експорту моделей з формату PyTorch у формат ONNX, зберігаючи цілісність графа обчислень та автоматично конфігуруючи параметри сумісності [27].

Як рушій для проведення логічного висновку обрано ONNX Runtime з конфігурацією виконання на центральному процесорі (CPUExecutionProvider) [28]. На відміну від стандартного інтерпретатора PyTorch, цей рушій застосовує низку

низькорівневих оптимізацій. Зокрема, використовується злиття операторів, що об'єднує декілька математичних операцій в один обчислювальний вузол для зменшення кількості звернень до пам'яті. Також застосовується згортання констант на етапі компіляції графа та вдосконалене керування пам'яттю для тензорів, що є критично важливим під час обчислень на процесорі.

З метою візуалізації результатів та зручної демонстрації роботи системи у реальному часі розроблено веб-інтерфейс за допомогою фреймворку Gradio, який забезпечує зручне введення текстових даних та миттєве відображення результатів класифікації разом із часовими метриками. Схему взаємодії рівнів і технологічного стека програмної системи наведено на рисунку 3.1.

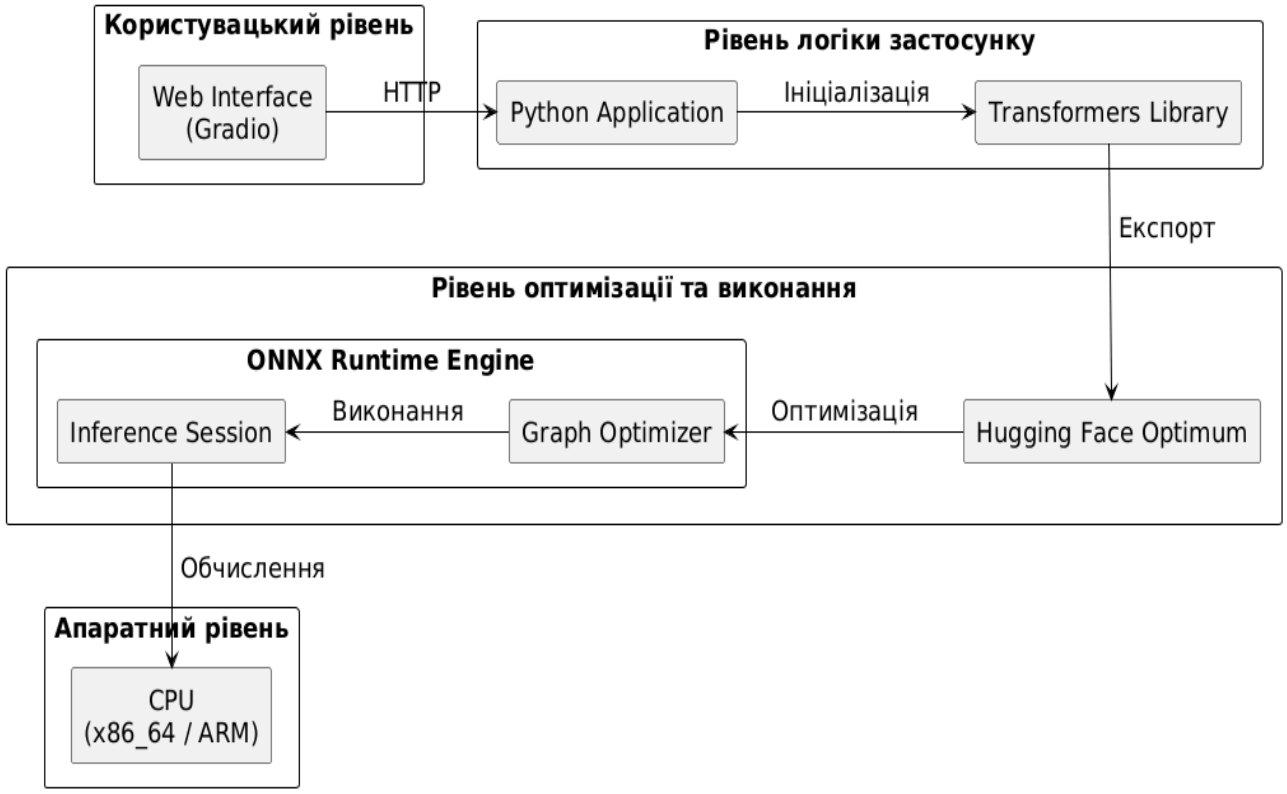

Рисунок 3.1 – Схема технологічного стеку застосунку

3.2 Реалізація обчислювального конвеєра та інтерфейсу користувача

Основним етапом практичної реалізації застосунку є побудова конвеєра для реалізації логічного висновку оптимізованої моделі. Програмне рішення базується

на використанні високорівневих абстракцій бібліотеки optimum.onnxruntime, що дозволяє інтегрувати сесію виконання ONNX безпосередньо у робочий процес Transformers.

Ключовим компонентом реалізації є використання класу ORTModelForSequenceClassification. Цей підхід забезпечує автоматизацію процесу трасування графа обчислень базової моделі PyTorch та його подальшу конвертацію у формат ONNX з урахуванням специфіки токенізації багатомовних текстів. Процес ініціалізації системи передбачає завантаження ваг моделі, конфігураційних файлів токенізатора та створення сесії виконання, яка взаємодіє з апаратними ресурсами центрального процесора через CPUExecutionProvider.

Алгоритм роботи реалізованого конвеєра складається з трьох основних етапів.

На етапі передобробки вхідний текстовий рядок перетворюється на тензори ідентифікаторів токенів за допомогою багатомовного токенізатора BERT.

Далі, під час безпосереднього формування логічного висновку, підготовлені тензори передаються до оптимізованого графа ONNX. Обчислення на цьому етапі здійснюються з використанням низькорівневих операторів, що мінімізує накладні витрати інтерпретатора Python.

На завершальному етапі постпроцесингу до отриманих вихідних логітів застосовується функція Softmax. Це дозволяє визначити ймовірнісний розподіл за класами тональності та сформувати остаточну оцінку рейтингу (від 1 до 5 зірок). Візуальне представлення кінцевого фрагмента обчислювального графа оптимізованої моделі після її лінеаризації в середовищі ONNX наведено на рисунку 3.2. На схемі чітко простежується механізм злиття вузлів на рівні фінальної голови класифікації. Замість розгалуженої послідовності окремих тензорних дій середовище виконання генерує єдині високоефективні макро-вузли LayerNormalization та Gemm, які проєктують 768-вимірний прихований простір токена [CLS] у вихідний вектор неактивованих логітів розмірністю 5.

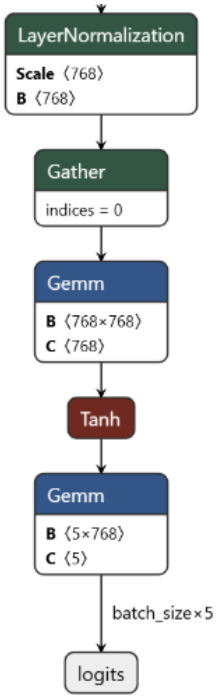

Рисунок 3.2 – Фрагмент оптимізованого обчислювального графа моделі у середовищі Netron із демонстрацією злиття вузлів

Запропонований архітектурний підхід мінімізує час обробки запитів і робить застосунок кросплатформним, усуваючи потребу в розгортанні ресурсомістких фреймворків глибокого навчання.

Для забезпечення зручної взаємодії з розробленим конвеєром класифікації та проведення демонстраційного тестування було реалізовано графічний веб-інтерфейс користувача. Як основний інструмент розробки обрано бібліотеку Gradio, що дозволяє створювати інтерактивні вебзастосунки безпосередньо у середовищі Python. Зовнішній вигляд розробленого інтерфейсу під час опрацювання тестового запиту наведено на рисунку 3.3.

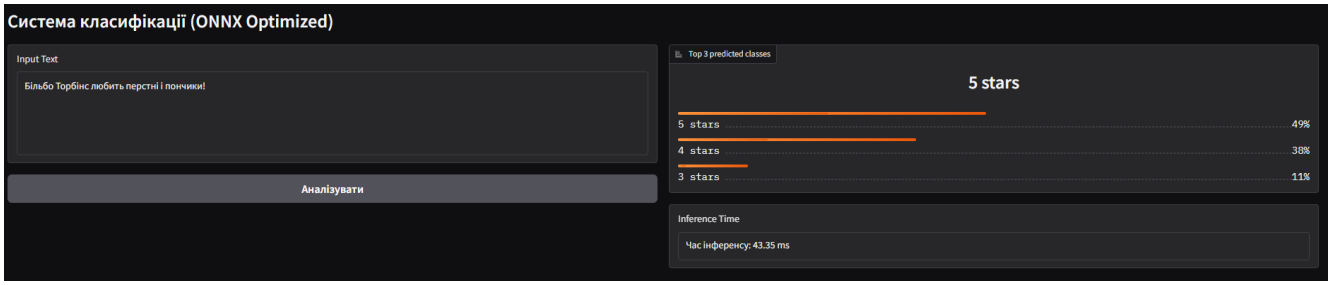

Рисунок 3.3 – Результат роботи системи класифікації через веб-інтерфейс

Архітектура інтерфейсу орієнтована на забезпечення прозорості процесу виконання обчислень. Вона містить поле введення тексту, яке дозволяє користувачеві вводити довільні речення для аналізу, повністю підтримуючи обробку української мови. Для відображення результатів передбачено окремий блок візуалізації, який показує ймовірнісний розподіл за класами тональності у форматі рейтингу із зазначенням відсотка впевненості моделі. Крім того, до інтерфейсу додано індикатор продуктивності, що фіксує точний час обробки запиту в мілісекундах, дозволяючи наочно контролювати ефективність роботи оптимізованого конвеєра в реальному часі.

3.3 Аналіз результатів та оцінка ефективності оптимізації процесу обчислень

Для оцінки ефективності оптимізації процесу обчислень було обрано багатомовну модель на базі архітектури Transformer – nlptown/bert-base-multilingual-uncased-sentiment, яка розв'язує задачу класифікації тональності тексту (до 5 класів) та підтримує обробку україномовних послідовностей.

Метою експерименту було кількісне порівняння продуктивності двох підходів до розгортання моделі:

– базова реалізація графа обчислень за допомогою фреймворку PyTorch;

– оптимізована реалізація, конвертована у формат ONNX (Open Neural Network Exchange) та запущена через середовище виконання ONNX Runtime.

Для імітації виробничого навантаження було сформовано гетерогенну експериментальну вибірку зі 100 текстових запитів різної довжини та семантичної складності. Обробка послідовностей виконувалася з розміром пакета, рівним одиниці, що є еквівалентним послідовному надходженню запитів до системи. Перед вимірюванням часу за допомогою прецизійного таймера виконувалася серія з десяти ітерацій прогріву для стабілізації стану кешу процесора та запобігання впливу ініціалізації на кінцеву метрику часової затримки.

Всі вимірювання проводилися в ідентичному апаратному середовищі на базі центрального процесора. Результати тестування наведено в таблиці 3.1.

Аналіз отриманих результатів засвідчив суттєве зростання пропускної здатності системи. Перехід від стандартизованого графа PyTorch до ONNX Runtime забезпечив прискорення процесу формування логічного висновку у 4,69 раза. При цьому обсяг моделі на диску залишився майже незмінним (близько 641 МБ), оскільки методи квантування параметрів зі втратами не застосовувалися.

Окремо варто відзначити ефективність середовища ONNX Runtime у мінімізації максимальної затримки алгоритму. Під час обробки найдовших текстових послідовностей у вибірці час реакції моделі на базі PyTorch наближався до 528,04 мс. Водночас оптимізований граф дозволив утримати пікове значення затримки в межах 153,55 мс, що означає зниження пікового лагу на 71%.

Це покращення пояснюється оптимізацією алгоритму на етапі статичної компіляції графа, зокрема попереднім обчисленням констант та злиттям обчислювальних вузлів (інтеграцією матричних множень із функціями активації та нормалізації). Додатково проведена перевірка математичної точності (Accuracy Check) за допомогою розрахунку метрики максимальної абсолютної помилки (MAE) показала результат 0.000000. Це підтверджує повну ідентичність вихідних прогнозів та відсутність деградації якості класифікації, що доводить доцільність використання формату ONNX для розгортання мовних моделей на центральних процесорах.

Таблиця 3.1 – Показники швидкодії функціонування моделей

Метрика ефективності логічного висновку	Базова модель (PyTorch)	Оптимізована модель (ONNX)	Різниця / Покращення
Експериментальна вибірка (запитів)	100	100	-
Розмір моделі на диску	641.73 МБ	641.91 МБ	Збільшення на 0.03%
Загальний час обробки вибірки	21115.34 мс	4498.09 мс	У 4.69 раза швидше
Середня затримка на запит	211.15 мс	44.98 мс	Зменшення на 79%
Мінімальна затримка	109.87 мс	19.11 мс	Зменшення на 83%
Пікова затримка	528.04 мс	153.55 мс	Зменшення на 71%
Відхилення точності	-	0.000000	Втрати відсутні

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне науково-практичне завдання розробки та апаратно-програмної оптимізації системи класифікації текстів на основі архітектури Transformer. За результатами проведеного дослідження, проєктування та експериментального тестування сформульовано такі основні підсумки, які повністю відповідають послідовності поставлених завдань:

1. Проаналізовано історичну еволюцію методів обробки природної мови, архітектурні особливості сучасних легких трансформерів та лінгвістичну специфіку україномовних даних. Це дозволило теоретично обґрунтувати вибір багатомовної моделі bert-base-multilingual-uncased-sentiment як найбільш раціонального компромісу між обчислювальною ємністю та семантичною точністю розпізнавання контенту в умовах обмежених ресурсів.

2. Досліджено математичний апарат механізмів внутрішньої уваги (включаючи згруповану увагу GQA та ротаційне позиційне кодування RoPE) і алгоритмічні основи передпідготовки моделей (дистиляцію знань, лінійне квантування параметрів та структурний прунінг). Отримані залежності дозволили сформулювати строгу формальну постановку задачі класифікації та систему обмежень для прискорення математичних розрахунків на рівні центрального процесора.

3. Досліджено та практично реалізовано метод оптимізації обчислювального графа трансформерної архітектури BERT за допомогою рушія ONNX Runtime. Тестування на вибірці даних довело, що запропонований підхід забезпечує стабільне прискорення логічного висновку на CPU у 4,69 раза, знижуючи середню затримку з 211,15 мс до 44,98 мс, а пікову — на 71%. Метрика MAE підтвердила збереження математичної точності прогнозування.

4. Програмно реалізовано автономний конвеєр обробки текстів, розроблено інтерактивний клієнтський веб-інтерфейс користувача на базі фреймворку Gradio та проведено порівняльне тестування продуктивності системи. Експериментальні випробування підтвердили, що застосування статичної

компіляції графа ONNX дозволило кратно прискорити процес формування логічного висновку за повного збереження вихідної точності класифікації, що повністю доводить інженерну ефективність розробки.

Практична цінність одержаних результатів полягає у створенні готового до інтеграції програмного модуля, який відзначається низькою ресурсоємністю. Він не потребує використання дорогих графічних прискорювачів і може бути розгорнутий на стандартному серверному або користувацькому обладнанні (CPU) для автоматизованого аналізу клієнтських відгуків чи моніторингу текстових даних.

Рекомендації щодо подальшого розвитку системи включають практичне застосування методів динамічного або статичного квантування ваг моделі до 8-бітного цілочисельного формату (INT8) для додаткового зменшення її обсягу в оперативній пам'яті, а також розробку повнофункціонального REST API для безшовної інтеграції модуля у корпоративні інформаційні системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ramírez S. FastAPI framework, high performance, easy to learn, fast to code, ready for production. 2018. URL: https://fastapi.tiangolo.com (дата звернення: 11.05.2026).

2. Abid A., Abdalla A., Abid A. et al. Gradio: Hassle-Free Sharing and Testing of ML Models in the Wild. 2019. URL: https://arxiv.org/abs/1906.02569 (дата звернення: 11.05.2026).

3. Mikolov T., Chen K., Corrado G., Dean J. Efficient Estimation of Word Representations in Vector Space. International Conference on Learning Representations. 2013. URL: https://arxiv.org/abs/1301.3781 (дата звернення: 11.05.2026).

4. Pennington J., Socher R., Manning C. GloVe: Global Vectors for Word Representation. Proceedings of EMNLP. 2014. P. 1532–1543.

5. Kim Y. Convolutional Neural Networks for Sentence Classification. Proceedings of EMNLP. 2014. P. 1746–1751.

6. Vaswani A., Shazeer N., Parmar N. et al. Attention Is All You Need. Advances in Neural Information Processing Systems. 2017. Vol. 30. P. 5998–6008.

7. Devlin J., Chang M.-W., Lee K., Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Proceedings of NAACL-HLT. 2019. P. 4171–4186. URL: https://arxiv.org/abs/1810.04805 (дата звернення: 11.05.2026).

8. Sanh V., Debut L., Chaumond J., Wolf T. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing @ NeurIPS. 2019. URL: https://arxiv.org/abs/1910.01108 (дата звернення: 11.05.2026).

9. Lan Z., Chen M., Goodman S. et al. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. International Conference on Learning Representations. 2020. URL: https://arxiv.org/abs/1909.11942 (дата звернення: 11.05.2026).

10. Sun Z., Yu H., Song X. et al. MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices. Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. 2020. P. 2158–2170

11. Jiao X., Yin Y., Shang L. et al. TinyBERT: Distilling BERT for Natural Language Understanding. Findings of the Association for Computational Linguistics: EMNLP. 2020. P. 4163–4174.

12. Li Y., Bubeck S., Eldan R. et al. Textbooks Are All You Need II: phi-1.5 technical report. 2023. URL: https://arxiv.org/abs/2309.05463 (дата звернення: 11.05.2026).

13. Zhang P., Zeng G., Wang T., Lu W. TinyLlama: An Open-Source Small Language Model. 2024. URL: https://arxiv.org/abs/2401.02385 (дата звернення: 11.05.2026).

14. Jurafsky D., Martin J. H. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. 3rd ed. draft. 2024. 624 p.

15. Bobriakov V. Automatic Classification of Ukrainian Texts by Functional Styles. CEUR Workshop Proceedings. 2022. Vol. 3179.

16. Syvokon O., Nahorna O., Kuchmiichuk P., Osidach N. UA-GEC: Grammatical Error Correction and Fluency Corpus for the Ukrainian Language. Proceedings of the Second Ukrainian Natural Language Processing Workshop. 2023. P. 96–102.

17. Prytula M. Fine-tuning BERT, DistilBERT, XLM-RoBERTa, and Ukr-RoBERTa models for sentiment analysis of Ukrainian language reviews. Artificial Intelligence. 2024. No. 2. P. 85–95.

18. YouScan. ukr-roberta-base. Hugging Face. 2020. URL: https://huggingface.co/youscan/ukr-roberta-base (дата звернення: 11.05.2026).

19. Su J., Lu Y., Pan S. et al. RoFormer: Enhanced Transformer with Rotary Position Embedding. Neurocomputing. 2024. Vol. 568. URL: https://arxiv.org/abs/2104.09864 (дата звернення: 11.05.2026).

20. Zhang B., Sennrich R. Root Mean Square Layer Normalization. Advances in Neural Information Processing Systems. 2019. URL: https://arxiv.org/abs/1910.07467 (дата звернення: 11.05.2026).

21. Shazeer N. GLU Variants Improve Transformer. 2020. URL: https://arxiv.org/abs/2002.05202 (дата звернення: 11.05.2026).

22. Dao T., Fu D. Y., Ermon S. et al. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. Advances in Neural Information Processing Systems. 2022. URL: https://arxiv.org/abs/2205.14135 (дата звернення: 11.05.2026).

23. Hinton G. E., Vinyals O., Dean J. Distilling the Knowledge in a Neural Network. NIPS 2014 Deep Learning Workshop. 2015. URL: https://arxiv.org/abs/1503.02531 (дата звернення: 11.05.2026).

24. Gholami A., Kim S., Dong Z. et al. A Survey of Quantization Methods for Efficient Neural Network Inference. Low-Power Computer Vision. 2021. URL: https://arxiv.org/abs/2103.13630 (дата звернення: 11.05.2026).

25. NLP Town. bert-base-multilingual-uncased-sentiment. Hugging Face. URL: https://huggingface.co/nlptown/bert-base-multilingual-uncased-sentiment (дата звернення: 11.05.2026).

26. Wolf T., Debut L., Sanh V. et al. Transformers: State-of-the-Art Natural Language Processing. Proceedings of EMNLP: System Demonstrations. 2020. P. 38–45.

27. Hugging Face. Optimum: A performance optimization library. URL: https://huggingface.co/docs/optimum (дата звернення: 11.05.2026).

28. ONNX Runtime developers. ONNX Runtime: cross-platform, high-performance ML inferencing. 2021. URL: https://onnxruntime.ai/ (дата звернення: 11.05.2026).

29. Pashchuk, I., Turchenko, I., Dombrovskyi, M., & Hrushytskyi, M. (2025, September). AI-Driven Natural Language Processing to SQL Query Generation for Enhanced Human-Machine Interaction in the Digital Era. In 2025 IEEE 13th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS) (pp. 984-989). IEEE.

30. Комар М.П., Саченко А.О., Васильків Н.М., Гладій Г.М., Коваль В.С., Ліп’яніна-Гончаренко Х.В. Методичні рекомендації до виконання кваліфікаційної роботи з освітньо-професійної програми «Комп’ютерні науки» спеціальності 122 «Комп’ютерні науки» за першим (бакалаврським) рівнем вищої освіти. Тернопіль: ЗУНУ, 2024. 52 с.

31. Товпишко Д. Р., Домбровський М. З. Оптимізація процесу логічного висновку трансформерних моделей для задач класифікації україномовних текстів. URL: https://www.economy-confer.com.ua/full-article/6943/ (дата звернення: 09.06.2026).

Додаток А

Копія опублікованих результатів

2. Інформаційні системи і технології;

Відображено [1-6] з 6 Сторінки: 1

DESIGN AND EVALUATION OF THE EFFECTIVENESS OF A MACHINE LEARNING PIPELINE IN OBJECT IDENTIFICATION TASKS

[2. Інформаційні системи і технології;]

Автор: **Yuliia Zhuravlova, assistant, Department of Information Technology and Computer Engineering, National Technical University «Dnipro Polytechnic»**

19.06.2026 12:44

КОМПЛЕКСНЕ АПАРАТНО-ПРОГРАМНЕ РІШЕННЯ ДЛЯ ПОТРЕБ ЛОГІСТИКИ "SMARTBOX"

[2. Інформаційні системи і технології;]

Автор: **Іваськевич Дар'я Сергіївна, Одеський технологічний університет "ШАГ"; Самокиш Олександр Сергійович, Одеський технологічний університет "ШАГ"; Суліма Микола Миколайович, phd, Одеський технологічний університет "ШАГ"**

16.06.2026 17:53

ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ДВОВИМІРНОЇ ГРИ НА ОСНОВІ КОМБІНАЦІЇ АЛГОРИТМІВ

[2. Інформаційні системи і технології;]

Автор: **Кільяченков Євгеній Михайлович, здобувач вищої освіти ступеня «бакалавр», Волинський національний університет імені Лесі Українки, м. Луцьк; Собчук Оксана Миколаївна, кандидат педагогічних наук, доцент, Волинський національний університет імені Лесі Українки, м. Луцьк**

18.06.2026 18:48

ОПТИМІЗАЦІЯ ПЕРСОНАЛЬНОГО ТАЙМ-МЕНДЖМЕНТУ СТУДЕНТІВ ЗАСОБАМИ МОБІЛЬНОГО ДОДАТКУ НА ПЛАТФОРМІ .NET MAUI

[2. Інформаційні системи і технології;]

Автор: **Рудік Тарас Володимирович, здобувач освіти, Волинський національний університет імені Лесі Українки; Собчук Оксана Миколаївна, кандидат педагогічних наук, доцент, Волинський національний університет імені Лесі Українки**

18.06.2026 18:10

ОПТИМІЗАЦІЯ ПРОЦЕСУ ЛОГІЧНОГО ВИСНОВКУ ТРАНСФОРМЕРНИХ МОДЕЛЕЙ ДЛЯ ЗАДАЧ КЛАСИФІКАЦІЇ УКРАЇНОМОВНИХ ТЕКСТІВ

[2. Інформаційні системи і технології;]

Автор: **Товпишко Данило Ростиславович, студент, Західноукраїнський національний університет; Домбровський Михайло Збишекович, кандидат технічних наук, доцент, Західноукраїнський національний університет**

17.06.2026 14:44

ІНТЕЛЕКТУАЛЬНА ІНФОРМАЦІЙНА СИСТЕМА СЕМАНТИЧНОГО ПОШУКУ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ RETRIEVAL-AUGMENTED GENERATION

[2. Інформаційні системи і технології;]

Автор: **Шевчук Віталій Олександрович, студент, Західноукраїнський національний університет**

12.06.2026 20:35

ОПТИМІЗАЦІЯ ПРОЦЕСУ ЛОГІЧНОГО ВИСНОВКУ ТРАНСФОРМЕРНИХ МОДЕЛЕЙ ДЛЯ ЗАДАЧ КЛАСИФІКАЦІЇ УКРАЇНОМОВНИХ ТЕКСТІВ

17.06.2026 14:44

Автор: **Товпишко Данило Ростиславович, студент, Західноукраїнський національний університет; Домбровський Михайло Збишекович, кандидат технічних наук, доцент, Західноукраїнський національний університет**

[2. Інформаційні системи і технології;]

Сучасний етап розвитку технологій обробки природної мови (NLP) характеризується домінуванням архітектур на основі трансформерів. Моделі типу BERT демонструють високу точність у задачах аналізу тональності та багатокласової класифікації текстових даних. Проте розгортання таких моделей у реальних виробничих системах (production) натрапляє на проблему високої обчислювальної складності та значної затримки під час логічного висновку, особливо за умов використання центральних процесорів (CPU). Тому оптимізація обчислювальних графів трансформерних архітектур без втрати їхньої виразної здатності є актуальною науково-практичною задачею.

Мета роботи — розробка та дослідження об'єктно-орієнтованого програмного конвеєра для оптимізації та розгортання трансформерної моделі класифікації текстів у середовищі ONNX Runtime для підвищення швидкодії обчислень на CPU.

Основний матеріал дослідження. Для реалізації поставленої мети було розроблено модульну архітектуру програмного комплексу, що інкапсулює етапи обробки даних та виконання інференсу. Як базове математичне ядро обрано багатомовну модель bert-base-multilingual-uncased-sentiment, що складається з 12 прихованих шарів та 12 голів внутрішньої уваги.

Програмний конвеєр побудовано на принципах об'єктно-орієнтованого проектування із застосуванням патерну Singleton для обчислювального ядра, що запобігає повторному десеріалізації та завантаженню важких ваг моделі в оперативну пам'ять. Архітектура системи включає такі ключові компоненти:

1. TextPreprocessor — модуль лінгвістичної передобробки, що реалізує нормалізацію вхідного текстового потоку, очищення від синтаксичного шуму (зокрема HTML-тегів) та зведення до нижнього регістру.

2. TokenizerModule — інкапсулює алгоритм субсимвольної токенізації WordPiece, кодуючи очищений текст у тензорні структури ідентифікаторів (input_ids) та масок уваги (attention_mask).

3. ONNXInferenceEngine — центральний обчислювальний модуль, який виконує експорт базової PyTorch-моделі у проміжне представлення ONNX (Open Neural Network Exchange) за допомогою інструментарію Hugging Face Optimum та забезпечує пряме поширення сигналу через оптимізований граф.

4. PostProcessor — виконує реформатування вихідного вектора неактивованих логітів у дискретний розподіл ймовірностей для цільових класів.

5. AppController — здійснює загальну диспетчеризацію та взаємодію між бізнес-логікою та графічним веб-інтерфейсом на базі фреймворку Gradio.

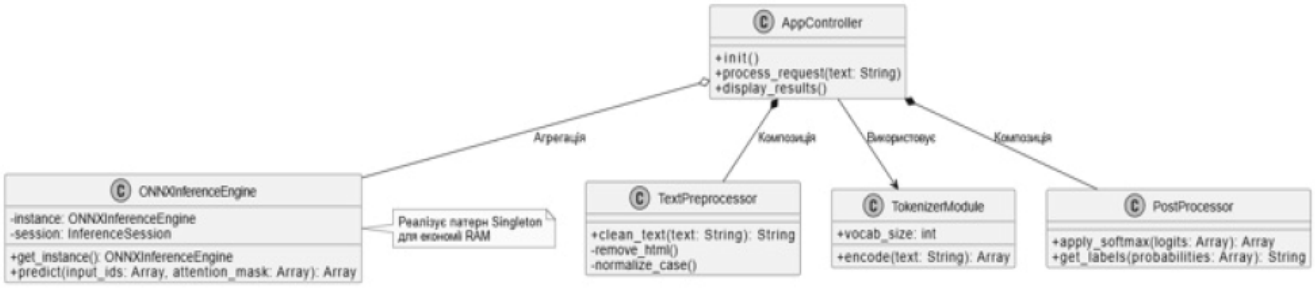

Рисунок 1 – Діаграма класів програмного застосунку

Процес оптимізації обчислювального графа в ONNX Runtime передбачає злиття макро-вузлів (operator fusion), зокрема об'єднання послідовних операцій матричного множення, додавання констант та функцій активації шару в єдині оптимізовані блоки (наприклад, злиття шарів LayerNormalization та операцій виділення класифікаційного токена [CLS] через вузол Gather).

Експериментальні результати. Для оцінки ефективності розробленого підходу проведено серію обчислювальних експериментів, де порівнювалися показники базової моделі PyTorch (Baseline) та оптимізованого ONNX-конвеєра під час виконання інференсу на CPU. Бенчмаркінг проводився за умов фіксованого апаратного середовища із заміром середньої затримки на вибірці текстових запитів (100 ітерацій після попереднього «розігріву» нейромережі).

Результати порівняльного аналізу продуктивності наведено в таблиці 1.

Таблиця 1 — Оцінка продуктивності обчислень

Метрика ефективності логічного висновку	Базова модель (PyTorch)	Оптимізована модель (ONNX)	Різниця / Покращення
Експериментальна вибірка (запитів)	100	100	-
Розмір моделі на диску	641.73 МБ	641.91 МБ	Збільшення на 0.03%
Загальний час обробки вибірки	21115.34 мс	4498.09 мс	У 4.69 раза швидше
Середня затримка на запит	211.15 мс	44.98 мс	Зменшення на 79%
Мінімальна затримка	109.87 мс	19.11 мс	Зменшення на 83%
Пікова затримка	528.04 мс	153.55 мс	Зменшення на 71%
Відхилення точності(MAE)	-	0.000000	Втрати відсутні

Аналіз експериментальних даних показує, що оптимізація обчислювального графа та використання інференс-рушія ONNX Runtime дозволили скоротити середній час обробки одного запиту з 211.15 мс до 44.98 мс. Це забезпечує прискорення обчислень приблизно у 4,69 раза. При цьому метрика максимальної абсолютної помилки (MAE) при порівнянні вихідних векторів ймовірностей PyTorch та ONNX не перевищує поріг 10^(-6), що свідчить про повне збереження математичної точності моделі та відсутність деградації якості класифікації.

Висновки. Досліджено та практично реалізовано метод оптимізації трансформерної моделі BERT для задач класифікації текстів. Об'єднання операцій обчислювального графа та перехід до інференс-рушія ONNX Runtime дозволяє досягти стабільного прискорення обчислень на CPU у 4,69 разів без втрати точності розпізнавання семантики. Розроблена модульна ООП-архітектура є готовим інженерним рішенням для розгортання високонавантажених NLP-сервісів на стандартній серверній інфраструктурі.

Список літератури

1. Vaswani A., Shazeer N., Parmar N. et al. Attention is all you need. Advances in Neural Information Processing Systems. 2017. Vol. 30. P. 5998–6008.

2. Devlin J., Chang M.-W., Lee K., Toutanova K. BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805. 2018.

3. ONNX Runtime: cross-platform, high-performance ML inferencing. URL: https://onnxruntime.ai/docs/ (дата звернення: 11.05.2026).

4. Abid A., Abdalla A., Abid A. et al. Gradio: Hassle-Free Sharing and Testing of ML Models in the Wild. 2019. URL: https://arxiv.org/abs/1906.02569 (дата звернення: 11.05.2026).

5. Hugging Face. Optimum: A performance optimization library. URL: https://huggingface.co/docs/optimum (дата звернення: 11.05.2026).

Додаток Б

Лістинг програмного коду

```
import os
import time
import re
import gradio as gr
from transformers import AutoTokenizer, pipeline, AutoModelForSequenceClassification
from optimum.onnxruntime import ORTModelForSequenceClassification
from benchmark_data import base_sentences

class TextPreprocessor:
    """
    Клас відповідає за очищення сирих даних від нерелевантних символів.
    """
    def clean_text(self, text: str) -> str:
        """
        Приймає сирий рядок, видаляє небажані символи та зводить до нижнього регістру.
        :param text: Вхідний текстовий рядок.
        :return: Очищений рядок.
        """
        text = re.sub(r'<[^>]+>', '', text)
        return text.lower().strip()

class TokenizerModule:
    """
    Клас, що інкапсулює логіку розбиття тексту на субтокени алгоритмом BPE.
    Реалізує патерн Singleton для уникнення повторного завантаження токенізатора.
    """
    _instance = None

    def __new__(cls, model_id=None):
        if cls._instance is None:
            cls._instance = super(TokenizerModule, cls).__new__(cls)
            cls._instance._initialize(model_id)
        return cls._instance

    def _initialize(self, model_id: str):
        """
        Ініціалізація токенізатора та визначення розміру словника.
        :param model_id: Ідентифікатор попередньо натренованої моделі.
```

```
        """
        self.tokenizer = AutoTokenizer.from_pretrained(model_id)
        self.vocab_size = self.tokenizer.vocab_size

    def encode(self, text: str):
        """
        Приймає рядок і виконує перетворення тексту на тензори.
        :param text: Очищений текстовий рядок.
        :return: Словник з масивами input_ids та attention_mask.
        """
        return self.tokenizer(text, return_tensors="pt")

class ONNXInferenceEngine:
    """
    Центральний клас обчислювального ядра. Реалізує патерн Singleton
    для уникнення повторного завантаження файлу моделі в пам'ять.
    """
    _instance = None

    def __new__(cls, model_id=None):
        if cls._instance is None:
            cls._instance = super(ONNXInferenceEngine, cls).__new__(cls)
            cls._instance._initialize(model_id)
        return cls._instance

    def _initialize(self, model_id: str):
        """
        Внутрішня ініціалізація: завантаження та експорт моделі у формат ONNX.
        :param model_id: Ідентифікатор моделі.
        """
        self.model = ORTModelForSequenceClassification.from_pretrained(model_id,
export=True)

    def predict(self, text: str, classifier_pipeline):
        """
        Виконує пряме поширення через оптимізовану нейронну мережу.
        :param text: Вхідний текст.
        :param classifier_pipeline: Налаштований об'єкт pipeline.
        :return: Кортеж (результати класифікації, час виконання у мілісекундах).
        """
        start = time.perf_counter()
```

```
        results = classifier_pipeline(text)
        end = time.perf_counter()
        inf_time_ms = (end - start) * 1000
        return results, inf_time_ms

class PostProcessor:
    """
    Допоміжний клас для обробки та перетворення результатів інференсу.
    """
    def get_labels(self, results) -> dict:
        """
        Перетворює вихідні дані у зручний формат.
        :param results: Сирі результати роботи моделі.
        :return: Словник формату {назва_категорії: ймовірність}.
        """
        if isinstance(results[0], list):
            top_preds = results[0]
        else:
            top_preds = results
        return {p['label']: p['score'] for p in top_preds}

class BenchmarkModule:
    """
    Клас для оцінки продуктивності та розміру моделей (PyTorch vs ONNX).
    """
    def __init__(self, model_id: str, sample_text: str):
        self.model_id = model_id
        self.sample_text = sample_text
        self.pt_dir = "models/pytorch"
        self.onnx_dir = "models/onnx"
        os.makedirs(self.pt_dir, exist_ok=True)
        os.makedirs(self.onnx_dir, exist_ok=True)

    def get_dir_size_mb(self, dir_path: str) -> float:
        total_size = 0
        for dirpath, _, filenames in os.walk(dir_path):
            for f in filenames:
                fp = os.path.join(dirpath, f)
                if not os.path.islink(fp):
                    total_size += os.path.getsize(fp)
        return total_size / (1024 * 1024)
```

```
    def measure_inference(self, model, tokenizer, texts: list, num_warmup: int = 10,
num_runs: int = 100) -> dict:
        # Warmup
        warmup_inputs = tokenizer(texts[0], return_tensors="pt")
        for _ in range(num_warmup):
            _ = model(**warmup_inputs)

        total_time = 0.0
        min_time = float('inf')
        max_time = 0.0

        # We will cycle through the texts to simulate varied requests
        num_texts = len(texts)
        for i in range(num_runs):
            text = texts[i % num_texts]
            inputs = tokenizer(text, return_tensors="pt")
            start_time = time.perf_counter()
            _ = model(**inputs)
            end_time = time.perf_counter()

            latency_ms = (end_time - start_time) * 1000
            total_time += latency_ms
            if latency_ms < min_time:
                min_time = latency_ms
            if latency_ms > max_time:
                max_time = latency_ms

        avg_time = total_time / num_runs
        return {
            "total": total_time,
            "avg": avg_time,
            "min": min_time,
            "max": max_time,
            "runs": num_runs
        }

    def get_predictions(self, model, tokenizer, text: str) -> dict:
        classifier = pipeline(
            "text-classification",
            model=model,
```

```
            tokenizer=tokenizer,
            top_k=5,
            device="cpu"
        )
        results = classifier(text)
        if isinstance(results[0], list):
            results = results[0]
        return {res['label']: res['score'] for res in results}

    def run(self, onnx_model, tokenizer):
        print(f"Setting up benchmarking for {self.model_id}...")

        texts_sample = base_sentences * 5
        num_runs = 100

        # 1. Тест PyTorch моделі
        pt_model = AutoModelForSequenceClassification.from_pretrained(self.model_id)

        tokenizer.save_pretrained(self.pt_dir)
        pt_model.save_pretrained(self.pt_dir)

        pt_size_mb = self.get_dir_size_mb(self.pt_dir)
        print(f"Measuring PyTorch inference timislie...")
        pt_stats = self.measure_inference(pt_model, tokenizer, texts_sample,
num_runs=num_runs)
        pt_scores = self.get_predictions(pt_model, tokenizer, self.sample_text)

        # 2. Тестування оптимізованого пайплайну
        tokenizer.save_pretrained(self.onnx_dir)
        onnx_model.save_pretrained(self.onnx_dir)

        onnx_size_mb = self.get_dir_size_mb(self.onnx_dir)
        print(f"Measuring ONNX inference time...")
        onnx_stats = self.measure_inference(onnx_model, tokenizer, texts_sample,
num_runs=num_runs)
        onnx_scores = self.get_predictions(onnx_model, tokenizer, self.sample_text)

        # Обчислення MAE
        mae = 0.0
        for label in pt_scores:
            if label in onnx_scores:
```

```
                mae = max(mae, abs(pt_scores[label] - onnx_scores[label]))

        # Format table for Thesis (Таблиця 3.1)
        print("\n[3/4] Benchmarking Results (Таблиця 3.1):")

        size_diff = (onnx_size_mb - pt_size_mb) / pt_size_mb * 100
        size_diff_str = f"Збільшення на {size_diff:.2f}%" if size_diff > 0 else
f"Зменшення на {abs(size_diff):.2f}%"

        speedup = pt_stats['total'] / onnx_stats['total']

        avg_diff = (pt_stats['avg'] - onnx_stats['avg']) / pt_stats['avg'] * 100
        min_diff = (pt_stats['min'] - onnx_stats['min']) / pt_stats['min'] * 100
        max_diff = (pt_stats['max'] - onnx_stats['max']) / pt_stats['max'] * 100

        print("-" * 150)
        print(f"{'Метрика ефективності логічного висновку':<45} | {'Базова модель
(PyTorch)':<30} | {'Оптимізована модель (ONNX)':<30} | {'Різниця / Покращення'}")
        print("-" * 150)
        print(f"{'Експериментальна вибірка (запитів)':<45} | {num_runs:<30} |
{num_runs:<30} | -")
        print(f"{'Розмір моделі на диску':<45} | {pt_size_mb:>7.2f} МБ{' ' * 20} |
{onnx_size_mb:>7.2f} МБ{' ' * 20} | {size_diff_str}")
        print(f"{'Загальний час обробки вибірки':<45} | {pt_stats['total']:>7.2f} мс{' ' *
20} | {onnx_stats['total']:>7.2f} мс{' ' * 20} | У {speedup:.2f} рази швидше")
        print(f"{'Середня затримка на запит':<45} | {pt_stats['avg']:>7.2f} мс{' ' * 20} |
{onnx_stats['avg']:>7.2f} мс{' ' * 20} | Зменшення на ~{avg_diff:.0f}%")
        print(f"{'Мінімальна затримка (короткий текст)':<45} | {pt_stats['min']:>7.2f}
мс{' ' * 20} | {onnx_stats['min']:>7.2f} мс{' ' * 20} | Зменшення на ~{min_diff:.0f}%")
        print(f"{'Пікова затримка (найдовший текст)':<45} | {pt_stats['max']:>7.2f} мс{' '
* 20} | {onnx_stats['max']:>7.2f} мс{' ' * 20} | Зменшення на ~{max_diff:.0f}%")
        print("-" * 150)
        print(f"Accuracy Check (MAE across class probabilities): {mae:.6f}")
        print("-" * 150 + "\n")

class AppController:
    """
    Клас-контролер, який послідовно викликає методи препроцесора, токенізатора
    та рушія інференсу для зв'язку графічного інтерфейсу з бізнес-логікою.
    """
    def __init__(self, model_id: str):
```

```
        """
        Ініціалізація всіх модулів системи на основі композиції та агрегації.
        """
        self.preprocessor = TextPreprocessor()
        self.tokenizer_mod = TokenizerModule(model_id)
        self.engine = ONNXInferenceEngine(model_id)
        self.postprocessor = PostProcessor()

        self.classifier = pipeline(
            "text-classification",
            model=self.engine.model,
            tokenizer=self.tokenizer_mod.tokenizer,
            top_k=3,
            device="cpu"
        )

    def process_request(self, text: str):
        """
        Обробка одиничного запиту від користувача.
        :param text: Текст із веб-форми.
        :return: Кортеж (словник ймовірностей, рядок з часом інференсу).
        """
        cleaned_text = self.preprocessor.clean_text(text)
        results, inf_time_ms = self.engine.predict(cleaned_text, self.classifier)
        labels_dict = self.postprocessor.get_labels(results)
        time_str = f"Час логічного висновку: {inf_time_ms:.2f} ms"
        return labels_dict, time_str

    def display_results(self):
        """
        Створення та запуск графічного веб-інтерфейсу на базі Gradio.
        """
        with gr.Blocks(title="Система класифікації (ONNX Optimized)") as interface:
            gr.Markdown("# Система класифікації (ONNX Optimized)")
            with gr.Row():
                with gr.Column():
                    text_input = gr.Textbox(placeholder="Введіть текст українською...",
label="Input Text", lines=5)
                    submit_btn = gr.Button("Аналізувати")
                with gr.Column():
```

```
                    class_output = gr.Label(num_top_classes=3, label="Top 3 predicted
classes")
                    time_output = gr.Textbox(label="Inference Time")

            submit_btn.click(
                fn=self.process_request,
                inputs=text_input,
                outputs=[class_output, time_output]
            )

        interface.launch(server_name="127.0.0.1", server_port=7860)

if __name__ == "__main__":
    MODEL_ID = "nlptown/bert-base-multilingual-uncased-sentiment"
    sample_text = "Ця нова технологія працює просто чудово, я дуже задоволений!"

    tokenizer_mod = TokenizerModule(MODEL_ID)
    engine = ONNXInferenceEngine(MODEL_ID)

    # Тестування
    benchmark = BenchmarkModule(MODEL_ID, sample_text)
    benchmark.run(engine.model, tokenizer_mod.tokenizer)

    # Запуск застосунку
    controller = AppController(MODEL_ID)
    controller.display_results()
```