

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК**

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Система автоматизованого лінгвістичного аналізу
текстів на основі застосування великих мовних моделей»

на здобуття освітнього ступеня магістра
зі спеціальності F3 Комп'ютерні науки

(код, найменування спеціальності)

освітньо-професійної програми Комп'ютерні науки
(назва)

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

(підпис)

Веніамін ПОДГОРБУНСЬКИЙ

(Ім'я, ПРІЗВИЩЕ здобувача)

Виконав:
здобувач вищої освіти Веніамін ПОДГОРБУНСЬКИЙ
група КНДМ-61

Керівник:

науковий ступінь,
вчене звання

Чічур АНДРІЙ

к.е.н., доцент

Рецензент:

науковий ступінь,
вчене звання

(Ім'я, ПРІЗВИЩЕ)

Київ 2025

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**
Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук

Ступінь вищої освіти Магістр

Спеціальність F3 Комп'ютерні науки

Освітньо-професійна програма Комп'ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедру Комп'ютерних наук

_____ Віктор ВИШНІВСЬКИЙ
« _____ » _____ 2025 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Подгорбунському Веніаміну Володимировичу
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Система автоматизованого лінгвістичного аналізу текстів на основі застосування великих мовних моделей

керівник кваліфікаційної роботи Андрій ЧІЧУР к.е.н., доцент,

(Ім'я, ПРИЗВИЩЕ науковий ступінь, вчене звання)

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «30» 10.2025р. №467

2. Строк подання кваліфікаційної роботи «26» грудня 2025р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, методи лінгвістичного аналізу текстів, архітектури великих мовних моделей.

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

Дослідження методів лінгвістичного аналізу текстів та сучасних підходів NLP
Аналіз великих мовних моделей та можливостей їх застосування для
автоматизованого аналізу текстів

Проектування архітектури системи автоматизованого лінгвістичного аналізу
Розробка функціональних вимог та програмної реалізації системи
Оцінювання ефективності роботи системи та аналіз отриманих результатів

5. Перелік графічного матеріалу: презентація

1. Еволюція методів лінгвістичного аналізу текстів
2. Архітектура великих мовних моделей
3. Структура системи автоматизованого лінгвістичного аналізу
4. Приклади роботи системи та результати аналізу текстів

6. Дата видачі завдання «15» вересня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	19.10-05.11.23	виконано
2	Вивчення матеріалів щодо розвитку технологій обробки текстів та NLP-систем	05.11-12.11.23	виконано
3	Дослідження архітектур великих мовних моделей та принципів їх функціонування	13.11-19.11.23	виконано
4	Аналіз можливостей застосування великих мовних моделей для автоматизованого лінгвістичного аналізу текстів	20.11-25.11.23	виконано
5	Дослідження методів машинного навчання та глибокого навчання в задачах аналізу текстів	27.11-03.12.23	виконано
6	Розробка структури та функціональних вимог до системи автоматизованого лінгвістичного аналізу	04.12-10.12.23	виконано
7	Оформлення кваліфікаційної роботи: вступ, висновки, реферат	11.12-20.12.23	виконано
8	Розробка демонстраційних матеріалів	21.12-29.12.23	виконано

Здобувач вищої освіти

_____ (підпис)

Веніамін ПОДГОРБУНСЬКИЙ

(Ім'я, ПРИЗВИЩЕ)

Керівник

кваліфікаційної роботи

_____ (підпис)

Андрій ЧІЧУР

(Ім'я, ПРИЗВИЩЕ)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 167 стор., 9 табл., 33 джерела.

Наукове завдання – розробка системи автоматизованого лінгвістичного аналізу текстів на основі застосування великих мовних моделей.

Мета роботи – підвищення ефективності аналізу текстових даних шляхом використання сучасних великих мовних моделей та методів обробки природної мови.

Об'єкт дослідження – процес автоматизованого лінгвістичного аналізу текстів.

Предмет дослідження – методи та алгоритми лінгвістичного аналізу текстів із застосуванням великих мовних моделей.

Короткий зміст роботи: проведено аналіз сучасних методів обробки природної мови та лінгвістичного аналізу текстів.

Розглянуто архітектури великих мовних моделей та можливості їх застосування для автоматизованого аналізу текстових даних. Проаналізовано основні підходи до синтаксичного, семантичного та контекстного аналізу текстів.

Розроблено архітектуру системи автоматизованого лінгвістичного аналізу текстів та визначено її функціональні можливості. Реалізовано програмний прототип системи з використанням сучасних мов програмування та інструментів машинного навчання.

Проведено тестування роботи системи на різних типах текстів та здійснено оцінювання якості отриманих результатів.

КЛЮЧОВІ СЛОВА: ЛІНГВІСТИЧНИЙ АНАЛІЗ, ОБРОБКА ПРИРОДНОЇ МОВИ, ВЕЛИКІ МОВНІ МОДЕЛІ, ТЕКСТОВІ ДАНІ, АВТОМАТИЗОВАНІ СИСТЕМИ.

ABSTRACT

Text part of the master's qualification work: 167 pages, 9 tables, 33 sources.

Scientific task – development of an automated text linguistic analysis system based on large language models.

Object of research – the process of automated linguistic analysis of texts.

Subject of research – methods and algorithms of linguistic text analysis using large language models.

Summary of the work: The paper analyzes modern approaches to natural language processing and linguistic text analysis. Architectures of large language models and their application for automated text analysis are considered. The main methods of syntactic, semantic, and contextual text analysis are reviewed.

An architecture of an automated linguistic analysis system is developed, and its functional capabilities are defined. A software prototype of the system is implemented using modern programming languages and machine learning tools. The system is tested on different types of text data, and the quality of the obtained results is evaluated.

KEYWORDS: LINGUISTIC ANALYSIS, NATURAL LANGUAGE PROCESSING, LARGE LANGUAGE MODELS, TEXT DATA, AUTOMATED SYSTEMS.

ЗМІСТ

ВСТУП.....	10
1 ТЕОРЕТИЧНІ ОСНОВИ ЛІНГВІСТИЧНОГО АНАЛІЗУ ТЕКСТІВ ТА ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ.....	15
1.1 Основи обробки природної мови та лінгвістичного аналізу.....	15
1.2 Великі мовні моделі: архітектура та принципи роботи.....	25
2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ПРОЕКТУВАННЯ СИСТЕМИ.....	47
2.1 Методи та технології автоматизованого лінгвістичного аналізу.....	47
2.2 Огляд та порівняльний аналіз існуючих систем лінгвістичного аналізу.....	74
2.2.1 Комерційні API-рішення.....	74
2.2.1.1 OpenAI API (GPT-4, GPT-3.5).....	74
2.2.1.2 Google Cloud Natural Language API.....	76
2.2.1.3 Amazon Comprehend.....	77
2.2.1.4 Microsoft Azure Text Analytics.....	78
2.2.1.5 Anthropic Claude API.....	80
2.2.1.6 Cohere API.....	81
2.2.2 Відкриті бібліотеки та фреймворки.....	82
2.2.2.1 Hugging Face Transformers.....	82
2.2.2.2 spaCy.....	84
2.2.2.3 NLTK (Natural Language Toolkit).....	85
2.2.2.4 Gensim.....	87
2.2.2.5 Stanford CoreNLP.....	88
2.2.2.6 Stanza.....	90
2.2.3 Спеціалізовані системи.....	91
2.2.3.1 Системи для конкретних мов.....	91
2.2.3.2 Галузеві рішення.....	92
2.2.5 Виявлення переваг та недоліків.....	92
2.2.5.1 Аналіз проблемних місць існуючих рішень.....	92
2.2.5.2 Можливості для вдосконалення.....	94
2.2.5.3 Обґрунтування необхідності розробки нової системи.....	95
2.2.4 Висновки до підрозділу.....	96
3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ АВТОМАТИЗОВАНОГО ЛІНГВІСТИЧНОГО АНАЛІЗУ.....	98
3.1 Формування вимог до системи.....	98

3.1.1 Функціональні вимоги до системи.....	98
3.1.1.1 Єдиний REST API endpoint для аналізу тексту.....	98
3.1.1.2 Інтеграція з OpenRouter API.....	99
3.1.1.3 Підтримка локальних мовних моделей.....	100
3.1.1.4 Комплексний аналіз текстових даних.....	100
3.1.1.5 Структурована JSON відповідь.....	102
3.2 Проектування архітектури системи.....	102
3.2.1 Вибір архітектурного стилю.....	103
3.2.1.1 Обґрунтування вибору монолітної архітектури.....	103
3.2.1.2 Принципи організації монолітної архітектури.....	105
3.2.2 Компонентна структура системи.....	105
3.2.2.1 REST Controller.....	106
3.2.2.2 LLM Service.....	107
3.2.2.3 Analysis Service.....	109
3.2.2.4 Response Builder.....	110
3.2.3 Вибір технологічного стеку.....	112
3.2.3.1 Java 17+ як основна мова програмування.....	112
3.2.3.2 Spring Boot як application framework.....	113
3.3 Тестування системи.....	114
3.3.1 Методологія тестування.....	114
3.3.2 Результати тестування.....	114
3.3.3 Висновки за результатами тестування.....	117
ВИСНОВКИ.....	119
ПЕРЕЛІК ПОСИЛАНЬ.....	123
ДОДАТКИ.....	127
ПРЕЗЕНТАЦІЯ.....	161

ВСТУП

Актуальність теми дослідження. Сучасний етап розвитку інформаційних технологій характеризується експоненційним зростанням обсягів текстових даних, які генеруються у цифровому просторі. За оцінками дослідників, щодобово у світі створюється понад 2,5 квінтільйона байтів неструктурованих текстових даних, що охоплюють соціальні мережі, новинні портали, наукові публікації, корпоративну документацію та інші джерела. Ефективна обробка та аналіз таких масивів інформації стають критично важливими завданнями для бізнесу, науки, державного управління та суспільства в цілому.

Лінгвістичний аналіз текстів традиційно є трудомістким процесом, що вимагає значних часових ресурсів та високої кваліфікації фахівців. Водночас потреба у швидкій обробці великих текстових корпусів, виявленні семантичних зв'язків, аналізі тональності, класифікації документів та вирішенні інших прикладних завдань постійно зростає. Автоматизація лінгвістичного аналізу дозволяє не лише прискорити обробку інформації, але й виявити приховані закономірності, які можуть залишитися непоміченими при ручному аналізі.

Револьюційний прорив у галузі обробки природної мови відбувся з появою великих мовних моделей, побудованих на архітектурі трансформерів. Моделі GPT, BERT, T5 та їх численні варіації продемонстрували здатність до глибокого розуміння контексту, генерації когерентних текстів та вирішення складних лінгвістичних завдань без необхідності розробки спеціалізованих алгоритмів для кожної окремої задачі. Ця парадигмальна зміна відкриває нові можливості для створення універсальних систем автоматизованого аналізу, здатних адаптуватися до різноманітних предметних областей та типів текстів.

В українському контексті актуальність дослідження посилюється необхідністю забезпечення технологічного суверенітету у сфері обробки мови. Розробка систем автоматизованого лінгвістичного аналізу для української мови

сприяє розвитку національного цифрового простору, підвищує доступність передових технологій для вітчизняних підприємств та наукових установ, а також забезпечує захист мовних даних від витоку до зовнішніх платформ.

Аналіз останніх досліджень і публікацій. Теоретичні та практичні аспекти застосування нейронних мереж для обробки природної мови активно досліджуються світовою науковою спільнотою протягом останнього десятиліття. Фундаментальний внесок у розвиток архітектури трансформерів зробили Ващані та співавтори у роботі "Attention is All You Need" (2017), яка заклала основу для всіх сучасних великих мовних моделей. Запропонований механізм уваги дозволив ефективно обробляти довгі послідовності та враховувати контекстні залежності на різних рівнях.

Девлін та колектив дослідників з Google представили модель BERT (2018), яка вперше продемонструвала ефективність двонаправленого попереднього навчання для розуміння природної мови. Їхній підхід до навчання на основі маскованого моделювання мови та прогнозування наступного речення став стандартом для багатьох прикладних систем. Радфорд та команда OpenAI у серії робіт, присвячених моделям GPT (2018-2023), показали можливості автогресивних мовних моделей у вирішенні різноманітних завдань через навчання без учителя на великих текстових корпусах.

Браун та співавтори у дослідженні "Language Models are Few-Shot Learners" (2020) продемонстрували, що збільшення розміру моделі до 175 мільярдів параметрів дозволяє досягти вражаючих результатів у сценаріях з обмеженою кількістю навчальних прикладів. Це відкрило шлях до розробки універсальних систем, здатних адаптуватися до нових завдань без додаткового дотренування. Тоувон та колеги у роботі "LLaMA: Open and Efficient Foundation Language Models" (2023) представили сімейство відкритих моделей, які при меншому

розмірі демонструють конкурентну якість, що стимулювало подальші дослідження у напрямку оптимізації та демократизації доступу до технологій.

Бей та дослідники з Google у публікації "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models" (2022) розробили методологію пром프트-інженерії, яка дозволяє покращити якість розв'язання складних аналітичних завдань через структуроване формулювання запитів. Чанг та співавтори у роботі "A Survey on Evaluation of Large Language Models" (2023) систематизували підходи до оцінювання якості роботи великих мовних моделей у різних доменах, що є критично важливим для розробки надійних прикладних систем.

У вітчизняній науковій літературі питання застосування великих мовних моделей для аналізу текстів розглядаються у працях Анісімова, Марченка, Широкова та інших дослідників, які зосереджуються на специфіці обробки української мови та адаптації існуючих підходів до національного контексту.

Мета роботи полягає у розробці системи автоматизованого лінгвістичного аналізу текстів на основі застосування великих мовних моделей, яка забезпечить ефективне вирішення комплексу завдань обробки природної мови з високою точністю та продуктивністю.

Для досягнення поставленої мети визначено такі завдання дослідження. По-перше, необхідно дослідити теоретичні основи автоматизованого лінгвістичного аналізу та архітектури великих мовних моделей, проаналізувати математичні принципи їх функціонування та особливості застосування для різних типів лінгвістичних завдань. По-друге, слід проаналізувати існуючі рішення у галузі автоматизованої обробки текстів, виявити їхні переваги та обмеження, визначити критерії оцінювання ефективності таких систем. По-третє, потрібно спроектувати архітектуру системи автоматизованого лінгвістичного аналізу, що включає вибір відповідних мовних моделей, розробку модульної структури та

визначення протоколів взаємодії компонентів. По-четверте, реалізувати програмну систему з використанням сучасних фреймворків та бібліотек для обробки природної мови, забезпечивши її масштабованість та можливість інтеграції з зовнішніми джерелами даних. По-п'яте, провести комплексне тестування розробленої системи на різноманітних текстових корпусах для верифікації коректності роботи та виявлення можливих недоліків. По-шосте, оцінити ефективність системи за критеріями точності, швидкості обробки та споживання обчислювальних ресурсів порівняно з альтернативними підходами.

Об'єктом дослідження є процеси автоматизованого лінгвістичного аналізу текстів, що включають токенізацію, морфологічний розбір, синтаксичний аналіз, семантичну інтерпретацію, виявлення іменованих сутностей та інші операції обробки природної мови.

Предметом дослідження виступають методи та технології застосування великих мовних моделей для лінгвістичного аналізу текстів, включаючи техніки промпт-інженерії, fine-tuning, few-shot learning та оптимізації інференсу моделей.

У роботі використовується комплекс загальнонаукових та спеціальних методів дослідження. Метод аналізу застосовується для вивчення теоретичних основ обробки природної мови, декомпозиції складних лінгвістичних завдань на елементарні операції та дослідження архітектури існуючих мовних моделей. Метод синтезу використовується для інтеграції різних компонентів системи у цілісне рішення та узагальнення результатів експериментальних досліджень. Порівняльний метод застосовується для зіставлення характеристик різних мовних моделей, оцінювання ефективності альтернативних підходів до вирішення лінгвістичних завдань та верифікації переваг розробленої системи. Метод системного моделювання використовується для проектування архітектури системи, визначення взаємодії її компонентів та прогнозування поведінки в різних сценаріях використання. Експериментальний метод застосовується для практичної

перевірки функціональності системи, тестування її на реальних даних та збору кількісних метрик ефективності. Статистичні методи використовуються для обробки результатів експериментів, оцінювання достовірності отриманих даних та визначення значущості різниці між альтернативними підходами.

Наукова новизна роботи полягає у розробці комплексного підходу до створення системи автоматизованого лінгвістичного аналізу, що інтегрує можливості сучасних великих мовних моделей з модульною архітектурою, яка забезпечує гнучкість та розширюваність системи. Запропоновано методiku адаптації універсальних мовних моделей до специфічних завдань лінгвістичного аналізу через техніки ефективної промпт-інженерії та контекстного навчання, що дозволяє досягти високої точності без потреби у великих обсягах розмічених навчальних даних. Розроблено архітектурне рішення для організації конвеєрної обробки текстів з можливістю динамічного вибору оптимальних моделей залежно від характеристик вхідних даних та вимог до якості аналізу. Вперше систематизовано підходи до оцінювання ефективності застосування великих мовних моделей для комплексного лінгвістичного аналізу з урахуванням багатокритеріальних показників якості, продуктивності та економічної ефективності.

Практична значущість результатів дослідження визначається можливістю їх застосування у широкому спектрі галузей. Розроблена система може бути впроваджена в інформаційно-аналітичних підрозділах організацій для автоматизованого моніторингу медійного простору, аналізу суспільних настроїв та виявлення трендів. У науковій сфері система сприятиме прискоренню опрацювання великих текстових корпусів, систематизації наукових публікацій та виявленню міждисциплінарних зв'язків. Для освітніх закладів система може використовуватися як інструмент для перевірки студентських робіт, виявлення плагіату та надання рекомендацій щодо покращення якості текстів. У комерційному секторі результати роботи можуть бути застосовані для аналізу

відгуків клієнтів, автоматизації обробки звернень, класифікації документів та підтримки прийняття рішень на основі текстової інформації. Модульна архітектура системи дозволяє адаптувати її до специфічних потреб різних організацій без необхідності повного перепроєктування.

1 ТЕОРЕТИЧНІ ОСНОВИ ЛІНГВІСТИЧНОГО АНАЛІЗУ ТЕКСТІВ ТА ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ.

1.1 Основи обробки природної мови та лінгвістичного аналізу

Обробка природної мови (Natural Language Processing, NLP) є однією з найбільш динамічних та швидко розвиваючих галузей штучного інтелекту, що займається розробкою методів та алгоритмів для автоматизованої обробки, аналізу та розуміння текстів природними мовами. За визначенням Джураффскі та Мартін (2023), обробка природної мови є міждисциплінарною сферою, що поєднує елементи комп'ютерної лінгвістики, машинного навчання, когнітивної психології та теоретичної лінгвістики для створення обчислювальних систем, здатних розуміти, інтерпретувати та генерувати людську мову. На відміну від формальних мов програмування, природні мови характеризуються високим рівнем неоднозначності, контекстуальної залежності та варіативності, що робить їх автоматичну обробку надзвичайно складним завданням.

У структурі штучного інтелекту обробка природної мови займає особливе місце, оскільки мова є фундаментальним засобом людської комунікації та репрезентації знань. Мова становить основу мислення, культури та соціальної взаємодії, тому здатність комп'ютерних систем до ефективної роботи з текстами та мовленням відкриває шлях до створення по-справжньому інтелектуальних агентів. Еволюція NLP тісно пов'язана з розвитком штучного інтелекту в цілому, від ранніх правилорієнтованих систем 1950-1980-х років через статистичні методи 1990-2000-х до сучасних нейронних підходів, що домінують з 2010-х років. Кожен етап цієї еволюції приносив якісні зміни у можливостях систем обробки мови та розширював спектр практичних застосувань.

Сучасна обробка природної мови охоплює широке коло завдань, кожне з яких має власні особливості та вимагає специфічних підходів до вирішення. Розпізнавання іменованих сутностей (Named Entity Recognition, NER) є

фундаментальним завданням, що полягає у виявленні та класифікації текстових фрагментів, які позначають об'єкти реального світу. Система NER ідентифікує в тексті імена людей, назви організацій, географічні локації, часові вирази, грошові суми та інші категорії сутностей залежно від предметної області. Наприклад, у реченні "Компанія Google оголосила про відкриття нового офісу в Києві у грудні 2024 року" система має розпізнати "Google" як організацію, "Києві" як географічну локацію, а "грудні 2024 року" як часовий вираз. Складність завдання полягає у неоднозначності контексту, коли одне й те саме слово може належати до різних категорій залежно від оточення. Сучасні підходи до NER базуються на нейронних мережах, зокрема на архітектурах BiLSTM-CRF та трансформерах, які здатні враховувати широкий контекст та досягати точності понад 90 відсотків на стандартних бенчмарках.

Аналіз тональності (Sentiment Analysis) спрямований на визначення емоційного забарвлення тексту та ставлення автора до описуваних об'єктів чи подій. Це завдання має критичне значення для моніторингу репутації брендів, аналізу відгуків споживачів, відстеження суспільних настроїв у соціальних мережах та прогнозування поведінки ринків. Аналіз тональності може виконуватися на різних рівнях деталізації: від простої бінарної класифікації текстів як позитивних чи негативних до багатокласової класифікації з виділенням нейтральної тональності та різних градацій емоцій, а також до рівня аспектів, коли визначається ставлення до конкретних характеристик об'єкта. Наприклад, відгук про ресторан може містити позитивну оцінку кухні, але негативну оцінку обслуговування. Основна складність аналізу тональності полягає у розпізнаванні іронії, сарказму, подвійних заперечень та контекстуально залежних висловлювань, які можуть кардинально змінювати емоційне забарвлення тексту.

Машинний переклад є одним з найстаріших завдань обробки природної мови, що полягає у автоматичному перетворенні тексту з однієї мови на іншу зі збереженням семантики та стилістичних особливостей оригіналу. Протягом десятиліть підходи до машинного перекладу еволюціонували від

правилорозроблених систем через статистичний машинний переклад до нейронного машинного перекладу, що став домінуючим з 2016 року після публікації роботи Баданау про механізм уваги. Сучасні системи, побудовані на архітектурі трансформерів, демонструють якість перекладу, яка для деяких мовних пар наближається до рівня професійних перекладачів. Проте машинний переклад залишається складним завданням через необхідність врахування культурних особливостей, ідіоматичних виразів, багатозначності слів та граматичних відмінностей між мовами.

Генерація тексту охоплює широкий спектр завдань, від автоматичного складання резюме та написання новинних статей до створення діалогових систем та поетичних творів. Це завдання вимагає не лише розуміння вхідної інформації, але й здатності створювати когерентні, граматично правильні та стилістично відповідні тексти. Розвиток великих мовних моделей таких як GPT-3 та GPT-4 продемонстрував вражаючі можливості у генерації тексту, коли системи здатні створювати тексти практично нерозрізнимі від написаних людиною. Основні виклики у цій сфері пов'язані з контролем за фактичною достовірністю згенерованого контенту, запобіганням галюцинаціям моделей та забезпеченням етичності створюваних текстів.

Класифікація текстів є базовим завданням NLP, що полягає у автоматичному віднесенні документів до попередньо визначених категорій. Застосування класифікації охоплюють фільтрацію спаму, категоризацію новин, визначення тематики документів, виявлення токсичних коментарів та багато інших сценаріїв. Сучасні підходи до класифікації текстів базуються на використанні попередньо навчених мовних моделей, які забезпечують високу точність навіть при обмежених обсягах розмічених навчальних даних. Ключовою особливістю класифікації є необхідність врахування семантичної близькості категорій та здатність моделі узагальнювати знання на нові, раніше не бачені приклади.

Питально-відповідні системи (Question Answering Systems) представляють один з найбільш складних типів NLP-застосувань, що вимагають глибокого розуміння як питання користувача, так і контексту, в якому шукається відповідь. Такі системи можуть працювати у різних режимах: екстрактивному, коли відповідь витягується безпосередньо з наявного тексту; генеративному, коли відповідь синтезується на основі наявної інформації; та гібридному, що поєднує обидва підходи. Розвиток трансформерних архітектур значно підвищив якість питально-відповідних систем, дозволивши їм ефективно працювати з великими обсягами інформації та відповідати на складні багатокрокові питання, що вимагають логічного мислення.

Лінгвістичний аналіз текстів здійснюється на декількох взаємопов'язаних рівнях, кожен з яких має власні завдання та методи. Морфологічний аналіз є першим рівнем обробки тексту, що фокусується на структурі окремих слів та їх граматичних характеристиках. Токенізація становить початковий етап морфологічного аналізу і полягає у розбитті потоку тексту на окремі значущі одиниці – токени. Для європейських мов токенізація здебільшого зводиться до виділення слів та розділових знаків, проте навіть у цьому випадку виникають складнощі з обробкою скорочень, дат, числових виразів та складених слів. Для ієрогліфічних мов, таких як китайська, токенізація є значно складнішим завданням через відсутність явних розділювачів між словами. Сучасні підходи до токенізації включають субсловну токенізацію (Byte-Pair Encoding, WordPiece, SentencePiece), яка дозволяє ефективно обробляти рідкісні слова та слова поза словником шляхом їх розбиття на підслівні одиниці.

Лематизація та стемінг є процесами приведення словоформ до канонічної форми, що дозволяє зменшити розмірність словника та згрупувати різні граматичні варіанти одного слова. Стемінг виконує грубе відсікання афіксів за формальними правилами без глибокого лінгвістичного аналізу, тоді як лематизація враховує морфологічну структуру мови та контекст вживання слова для визначення його леми – початкової словникової форми. Наприклад, для

словоформ "біг", "біжу", "біжив" лематизація поверне базову форму "бігти", тоді як стемінг може дати різні результати залежно від алгоритму. Лематизація є більш точним методом, але вимагає наявності морфологічних словників та граматичних правил мови.

Морфологічне розмічення частинами мови (Part-of-Speech Tagging, POS-tagging) присвоює кожному токenu у тексті граматичну категорію: іменник, дієслово, прикметник, прийменник тощо. Ця інформація є критично важливою для подальшого синтаксичного та семантичного аналізу, оскільки граматична роль слова визначає його функцію у реченні. Складність POS-tagging полягає у розв'язанні омонімії, коли одне й те саме слово може належати до різних частин мови залежно від контексту. Наприклад, слово "світло" може бути іменником ("яскраве світло") або прислівником ("світло вдягнений"). Сучасні системи POS-tagging на основі нейронних мереж досягають точності понад 97 відсотків на стандартних корпусах, ефективно враховуючи контекст вживання слів.

Синтаксичний аналіз спрямований на виявлення граматичної структури речень та відношень між їх компонентами. Конституентний парсинг розбиває речення на вкладені синтаксичні групи (іменникові, дієслівні, прийменникові тощо), будуючи ієрархічне дерево структури речення відповідно до формальної граматики мови. Такий підхід дозволяє виявити структурну організацію речення та ідентифікувати основні синтаксичні конституенти. Дерева залежностей представляють альтернативний підхід, що фокусується на бінарних відношеннях між словами, де кожне залежне слово пов'язане зі своїм головним словом типізованим зв'язком. Наприклад, у реченні "Студент читає цікаву книгу" слово "читає" є головним для "студент" (відношення підмета) та для "книгу" (відношення прямого додатка), а "цікаву" залежить від "книгу" (відношення означення). Синтаксичний аналіз дозволяє виявити суб'єктно-об'єктні відношення, модифікатори та інші граматичні зв'язки, необхідні для розуміння семантики речення.

Семантичний аналіз переходить від формальної граматичної структури до розуміння значення тексту. Розпізнавання лексичної семантики включає визначення значень окремих слів з урахуванням контексту їх вживання. Багатозначність слів (полісемія) становить основну проблему семантичного аналізу, оскільки більшість слів природних мов мають множинні значення. Завдання усунення багатозначності (Word Sense Disambiguation, WSD) полягає у виборі релевантного значення слова з множини можливих на основі контексту. Наприклад, слово "мишка" може означати тварину або комп'ютерний пристрій, і лише контекст дозволяє визначити актуальне значення. Сучасні підходи до WSD використовують контекстуальні векторні представлення слів, що генеруються нейронними мовними моделями та автоматично враховують семантичний контекст.

Семантичний аналіз на рівні речення включає побудову формальних семантичних репрезентацій, які відображають предикатно-аргументну структуру висловлювань. Семантична рольова розмітка (Semantic Role Labeling) ідентифікує учасників ситуації, описаної у реченні, та їхні ролі відносно головної дії. Типові семантичні ролі включають агента (виконавця дії), пацієнта (об'єкт дії), інструмента, бенефіціара, локації та інші. Така репрезентація дозволяє абстрагуватися від конкретної синтаксичної реалізації та захопити глибинну семантичну структуру висловлювання. Розпізнавання семантичних відношень між сутностями (Relation Extraction) виявляє змістові зв'язки між іменованими сутностями у тексті, такі як "працює в", "розташований у", "є керівником" тощо, що дозволяє будувати графи знань та онтології предметних областей.

Прагматичний аналіз виходить за межі буквального значення тексту та розглядає мовленнєві акти, комунікативні інтенції та контекстуальні імплікатури. Цей рівень аналізу враховує, що значення висловлювання не зводиться до суми значень його компонентів, а залежить від комунікативної ситуації, відношень між співрозмовниками, їхніх фонових знань та культурного контексту. Прагматичний аналіз включає розпізнавання іронії та сарказму, визначення імпліцитно вираженої

інформації, розуміння риторичних питань та інших комунікативних стратегій. Виявлення анафоричних зв'язків (Coreference Resolution) є важливим завданням прагматичного рівня, що полягає у визначенні, які вирази у тексті відсилають до одного й того ж об'єкта реального світу. Наприклад, у послідовності речень "Марія прийшла до офісу. Вона почала працювати над проєктом" необхідно встановити, що займенник "вона" відсилає до "Марії".

Обробка тексту здійснюється послідовно на декількох рівнях, кожен з яких забезпечує певний ступінь абстракції та розуміння. На найнижчому рівні відбувається обробка символів та токенів, де текст розглядається як послідовність символічних одиниць без врахування їхнього значення. Лексичний рівень оперує окремими словами як основними значущими одиницями, визначаючи їхні граматичні характеристики та базові семантичні властивості. Синтаксичний рівень розглядає структуру словосполучень та речень, виявляючи граматичні відношення між компонентами. Семантичний рівень інтерпретує значення мовних виразів та будує формальні репрезентації змісту. Дискурсивний рівень аналізує структуру тексту понад межами окремих речень, виявляючи зв'язки між реченнями, тематичну структуру та риторичну організацію тексту. Прагматичний рівень враховує контекст використання мови та комунікативні інтенції.

Сучасні тренди у обробці природної мови характеризуються домінуванням підходів на основі глибокого навчання та трансформерних архітектур. Парадигма попереднього навчання та дотренування (pre-training and fine-tuning) стала стандартною практикою, коли модель спочатку навчається на великих немаркованих корпусах текстів для засвоєння загальних мовних закономірностей, а потім адаптується до специфічних завдань на невеликих розмічених наборах даних. Цей підхід дозволяє ефективно використовувати величезні обсяги текстової інформації та досягати високої якості навіть при обмеженій кількості спеціалізованих навчальних прикладів. Техніка трансферного навчання дозволяє переносити знання між різними завданнями та мовами, значно зменшуючи потребу у розмічених даних для кожної конкретної задачі.

Розвиток мультимодальних моделей, що поєднують обробку тексту з аналізом зображень, аудіо та відео, відкриває нові можливості для створення систем, здатних розуміти багатий контекст комунікації. Моделі типу CLIP, DALL-E та GPT-4 демонструють здатність до спільної обробки різних модальностей, генерації зображень за текстовими описами та відповідей на питання про візуальний контент. Інтеграція мовних моделей з зовнішніми інструментами та базами знань через механізми tool use та retrieval-augmented generation підвищує фактичну точність та розширює можливості систем для вирішення складних завдань, що вимагають доступу до актуальної інформації.

Промпт-інженерія виокремила як окрема дисципліна, що займається розробкою ефективних стратегій формулювання запитів до мовних моделей. Техніки few-shot та zero-shot навчання дозволяють адаптувати моделі до нових завдань без необхідності перенавчання, лише за рахунок правильного формулювання інструкцій та надання прикладів у контексті запиту. Chain-of-thought prompting демонструє, що явне структурування процесу міркування у промпті значно покращує здатність моделей до логічного мислення та розв'язання складних задач. Ці підходи демократизують доступ до можливостей великих мовних моделей, дозволяючи користувачам без глибоких технічних знань ефективно використовувати їх для розв'язання практичних завдань.

Попри значний прогрес, обробка природної мови стикається з численними викликами та проблемами, що обмежують можливості сучасних систем. Розуміння контексту залишається фундаментальною проблемою, оскільки інтерпретація висловлювання часто вимагає широких фонових знань про світ, культурний контекст та ситуацію спілкування. Мовні моделі можуть демонструвати вражаючі результати на стандартних тестах, але відчувати труднощі з питаннями, що вимагають здорового глузду або розуміння фізичних закономірностей. Проблема галюцинацій, коли моделі генерують правдоподібно звучачу, але фактично неправильну інформацію, становить серйозну перешкоду для застосування у критичних сферах, де точність є принциповою.

Багатозначність та неоднозначність природних мов проявляються на всіх рівнях аналізу, від лексичної полісемії до структурної омонімії та прагматичної неоднозначності. Одне й те саме речення може мати множинні інтерпретації залежно від контексту, просодії та інтенцій мовця. Системи обробки мови мають не лише розпізнавати множинність можливих інтерпретацій, але й обирати найбільш релевантну на основі доступної контекстуальної інформації. Обробка образної мови, метафор, ідіом та культурно специфічних виразів вимагає глибокого розуміння когнітивних механізмів та культурного контексту, що виходить за межі формального лінгвістичного аналізу.

Мовна варіативність проявляється у діалектних відмінностях, соціолектах, регістрах мовлення та індивідуальних стилях, що створює додаткові труднощі для створення універсальних систем обробки мови. Зміна мови у часі, поява нових слів та виразів, еволюція граматичних норм вимагають постійного оновлення та адаптації моделей. Низькоресурсні мови, для яких відсутні великі корпуси розмічених текстів та базові лінгвістичні ресурси, залишаються значно менш підтримуваними сучасними технологіями, що створює мовну нерівність у доступі до переваг NLP-технологій.

Етичні проблеми, пов'язані з упередженнями у навчальних даних, справедливістю алгоритмів, приватністю даних та потенційним зловживанням технологіями, набувають все більшої актуальності. Мовні моделі можуть відтворювати та посилювати соціальні стереотипи та дискримінаційні патерни, присутні у навчальних корпусах. Забезпечення інтерпретованості рішень нейронних моделей, створення механізмів контролю та пояснення їхньої поведінки є критично важливим для застосування у відповідальних сферах таких як медицина, право та освіта. Розробка етичних принципів використання NLP-технологій та механізмів їх дотримання становить важливий напрямок досліджень у сучасній обробці природної мови.

Таким чином, зважаючи на табл 1.1 обробка природної мови є багатогранною дисципліною, що поєднує теоретичну лінгвістику, машинне навчання та когнітивну науку для створення систем, здатних до розуміння та генерації людської мови. Лінгвістичний аналіз здійснюється на декількох взаємопов'язаних рівнях, від морфологічного до прагматичного, кожен з яких вносить свій внесок у загальне розуміння тексту. Сучасні підходи на основі великих мовних моделей демонструють безпрецедентні можливості, проте численні виклики у галузі розуміння контексту, подолання неоднозначності та забезпечення етичності залишаються актуальними напрямками досліджень.

Таблиця 1.1 - Характеристики рівнів аналізу

Рівень аналізу	Основні завдання	Методи та підходи	Виклики
Морфологічний	Токенізація, лематизація, POS-tagging	Субсловна токенизація (BPE, WordPiece), нейронні тегери	Обробка рідкісних слів, омонімія частин мови
Синтаксичний	Парсинг, побудова дерев залежностей	Transition-based та graph-based парсери, трансформери	Структурна неоднозначність, довгі залежності
Семантичний	WSD, розпізнавання ролей, екстракція відношень	Контекстуальні ембедінги, графові нейронні мережі	Полісемія, імпліцитні відношення, абстрактні концепти
Прагматичний	Розв'язання кореференції, виявлення іронії	Нейронні моделі з механізмами уваги, мультимодальні підходи	Контекстуальна залежність, культурна специфіка

1.2 Великі мовні моделі: архітектура та принципи роботи

Великі мовні моделі (Large Language Models, LLM) представляють собою кульмінацію багаторічного розвитку методів обробки природної мови та машинного навчання. Ці моделі являють собою нейронні мережі з мільярдами

параметрів, навчені на величезних текстових корпусах, що охоплюють різноманітні джерела людських знань. Розуміння архітектури та принципів функціонування великих мовних моделей є необхідним для їх ефективного застосування у практичних системах лінгвістичного аналізу. У цьому підрозділі розглядається еволюційний шлях від простих статистичних моделей до складних трансформерних архітектур, детально аналізуються ключові компоненти сучасних LLM та принципи їх навчання.

Історія розвитку мовних моделей демонструє послідовний перехід від простих статистичних підходів до складних нейронних архітектур, здатних захоплювати глибокі закономірності природної мови. Класичні n-грамні моделі, що домінували у обробці природної мови протягом десятиліть, базувалися на марківському припущенні про те, що ймовірність наступного слова залежить лише від обмеженої кількості попередніх слів. Біграмна модель розглядає лише одне попереднє слово, триграмна – два попередні слова, і так далі. Математично, n-грамна модель оцінює ймовірність послідовності слів $w_1, w_2, \dots, w_n$ як добуток умовних ймовірностей: $P(w_1, w_2, \dots, w_n) = \prod P(w_i | w_{i-n+1}, \dots, w_{i-1})$.

Основна перевага n-грамних моделей полягала у їхній простоті та обчислювальній ефективності. Параметри моделі – умовні ймовірності слів – оцінювалися безпосередньо з навчального корпусу шляхом підрахунку частот відповідних послідовностей. Проте ці моделі мали фундаментальні обмеження. По-перше, вони не могли враховувати довгострокові залежності у тексті через обмежений контекст. По-друге, проблема розрідженості даних (data sparsity) ставала критичною при збільшенні n, оскільки більшість можливих n-грам просто не зустрічалися у навчальному корпусі. Для подолання цієї проблеми використовувалися техніки згладжування (smoothing), такі як згладжування Лапласа, інтерполяція або back-off моделі Каца, проте вони лише частково вирішували фундаментальні обмеження підходу.

Рекурентні нейронні мережі (Recurrent Neural Networks, RNN) ознаменували перехід до нейросітових підходів у моделюванні мови та запропонували

елегантне рішення проблеми довгострокових залежностей. На відміну від n -грамних моделей, RNN обробляють послідовність слів рекурентно, підтримуючи прихований стан h_t , який агрегує інформацію про всю попередню історію. На кожному кроці t мережа отримує поточне слово x_t та попередній прихований стан h_{t-1} , обчислюючи новий прихований стан за формулою: $h_t = \tanh(W_{hh}h_{t-1} + W_{hx}x_t + b_h)$, де W_{hh} , W_{hx} – матриці ваг, b_h – вектор зсуву. Прихований стан потім використовується для передбачення наступного слова та передається на наступний крок обробки.

Теоретично, рекурентна структура дозволяє RNN враховувати контекст необмеженої довжини, оскільки прихований стан на кожному кроці кодує інформацію про всю попередню послідовність. Проте на практиці стандартні RNN страждають від проблеми зникаючого градієнта (vanishing gradient problem), що робить навчання довгострокових залежностей практично неможливим. При зворотному поширенні помилки через багато часових кроків градієнти експоненційно зменшуються, що призводить до неефективного оновлення ваг для захоплення віддалених залежностей. Крім того, послідовна природа обробки у RNN унеможлиблює ефективну паралелізацію обчислень, що обмежує масштабованість підходу.

Довга короткочасна пам'ять (Long Short-Term Memory, LSTM), запропонована Хохрайтером та Шмідхубером у 1997 році, стала проривом у подоланні обмежень стандартних RNN. Архітектура LSTM вводить складну систему вентилів (gates), які контролюють потік інформації та дозволяють мережі селективно зберігати, оновлювати або забувати інформацію. Ключовим компонентом є стан комірки (cell state) c_t , який проходить через всю послідовність з мінімальними нелінійними перетвореннями, дозволяючи градієнтам поширюватися без значного затухання. LSTM містить три типи вентилів: вентиль забування f_t визначає, яку частину попереднього стану комірки слід відкинути; вентиль входу i_t контролює, яка нова інформація буде додана до

стану; вентиль виходу o_t визначає, яка частина стану комірки буде використана для обчислення прихованого стану.

Математично, оновлення LSTM описується системою рівнянь:

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

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

$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

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

$$h_t = o_t \odot \tanh(c_t)$$

де σ позначає сигмоїдну функцію, $\odot$ – поелементне множення, а квадратні дужки – конкатенацію векторів. Ця архітектура дозволила ефективно навчати моделі на послідовностях значної довжини та захоплювати складні довгострокові залежності. Варіант LSTM – стробована рекурентна одиниця (Gated Recurrent Unit, GRU), запропонована Чо у 2014 році, спростила архітектуру, об'єднавши вентилі забування та входу в один вентиль оновлення, зменшивши кількість параметрів при збереженні ефективності.

LSTM та GRU стали стандартними будівельними блоками для багатьох NLP-застосувань середини 2010-х років, включаючи машинний переклад, моделювання мови, розпізнавання мовлення та аналіз тональності. Проте навіть ці вдосконалені рекурентні архітектури мали обмеження. Послідовна природа обробки залишалася фундаментальною перешкодою для масштабування на надвеликих корпусах. Довжина послідовностей, які могли ефективно оброблятися, хоча й збільшилася, все ще була обмеженою. Крім того, двонаправлені LSTM, хоча й могли враховувати як попередній, так і наступний контекст, вимагали окремих проходів у прямому та зворотному напрямках, що подвоювало обчислювальні витрати.

Механізм уваги (Attention Mechanism) став ключовою інновацією, що заклала основу для революційних змін у обробці природної мови. Первісно запропонований Баданау та співавторами у 2014 році для покращення нейронного машинного перекладу, механізм уваги дозволив моделям динамічно фокусуватися на релевантних частинах вхідної послідовності при генерації кожного елемента вихідної послідовності. Замість того, щоб стискати всю вхідну послідовність в єдиний вектор фіксованого розміру, що призводило до втрати інформації для довгих послідовностей, механізм уваги дозволяє декодеру безпосередньо звертатися до всіх прихованих станів енкодера, надаючи кожному з них динамічно обчислювану вагу релевантності.

Математично, для кожної позиції декодування t обчислюються ваги уваги α_i відносно кожного прихованого стану енкодера h_i . Ці ваги визначають, наскільки релевантна кожна позиція вхідної послідовності для генерації поточного виходу. Контекстний вектор c формується як зважена сума всіх прихованих станів енкодера: $c = \sum_i \alpha_i h_i$, де ваги уваги обчислюються через механізм вирівнювання, що порівнює поточний стан декодера з кожним станом енкодера. Цей підхід дозволив моделям ефективно обробляти довгі послідовності, автоматично виявляти відповідності між елементами вхідної та вихідної послідовностей, і значно покращив якість машинного перекладу, особливо для довгих речень.

Ключова ідея, що привела до появи трансформерів, полягала у усвідомленні, що механізм уваги може бути не просто доповненням до рекурентних мереж, але й повністю замінити їх. Якщо механізм уваги дозволяє кожному елементу послідовності безпосередньо взаємодіяти з усіма іншими елементами, то послідовна обробка стає непотрібною. Це відкрило шлях до масивної паралелізації обчислень та навчання моделей на корпусах безпрецедентного розміру.

Архітектура Transformer

Архітектура трансформер, представлена Ващані та співавторами у знаковій роботі "Attention is All You Need" (2017), фундаментально переосмислила підхід до обробки послідовностей, повністю відмовившись від рекурсії та згорток на користь механізмів уваги. Ця архітектура складається з двох основних компонентів: енкодера та декодера, кожен з яких побудований зі стеку ідентичних шарів. Оригінальна модель використовувала по шість шарів для енкодера та декодера, хоча у сучасних реалізаціях їх кількість може значно варіюватися.

Енкодер трансформера обробляє вхідну послідовність паралельно, перетворюючи її на послідовність контекстуалізованих векторних репрезентацій. Кожен шар енкодера складається з двох основних підшарів: механізму багатоголової само-уваги (multi-head self-attention) та позиційно-незалежної повнозв'язної мережі прямого поширення (position-wise feed-forward network). Навколо кожного підшару застосовується залишкове з'єднання (residual connection) з наступною нормалізацією шару (layer normalization), що формально записується як $\text{LayerNorm}(x + \text{Sublayer}(x))$, де $\text{Sublayer}(x)$ представляє функцію, реалізовану підшаром.

Декодер має аналогічну структуру, але містить додатковий підшар крос-уваги (cross-attention) між само-увагою та feed-forward мережею. Цей підшар дозволяє декодеру фокусуватися на релевантних частинах виходу енкодера при генерації кожного елемента послідовності. Важливою особливістю декодера є маскування майбутніх позицій у механізмі само-уваги, що забезпечує автогресивну природу генерації: при передбаченні позиції t модель може використовувати лише інформацію з позицій до t включно.

Механізм само-уваги (self-attention) є серцем трансформерної архітектури та дозволяє кожній позиції у послідовності безпосередньо враховувати інформацію з усіх інших позицій при формуванні свого представлення. Для вхідної послідовності векторів X , механізм уваги спочатку обчислює три набори векторів через лінійні проєкції: запити (queries) $Q = XW_q$, ключі (keys) $K = XW_k$ та значення (values) $V = XW_v$, де W_q , W_k , W_v – навчувані матриці ваг. Інтуїтивно,

запити представляють "що шукає" кожна позиція, ключі – "що пропонує" кожна позиція, а значення – "яку інформацію" вона передає.

Ваги уваги обчислюються як нормалізовані скалярні добутки між запитами та ключами, що відображає їх сумісність або релевантність. Вихід механізму уваги формується як зважена сума значень з вагами, визначеними цією сумісністю. Математично, масштабований скалярний добуток уваги (scaled dot-product attention) обчислюється за формулою:

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T / \sqrt{d_k})V$$

де d_k – розмірність ключів, а масштабування на $\sqrt{d_k}$ запобігає занадто великим значенням скалярних добутків, які могли б призвести до насичення функції softmax у областях з надзвичайно малими градієнтами. Функція softmax застосовується по рядках, перетворюючи значення схожості у ймовірнісний розподіл уваги.

Багатоголова увага (multi-head attention) розширює базовий механізм уваги, дозволяючи моделі одночасно фокусуватися на різних аспектах інформації з різних репрезентаційних підпросторів. Замість виконання єдиної функції уваги на векторах повної розмірності, багатоголова увага проєктує запити, ключі та значення h разів (де h – кількість голів) на простори меншої розмірності, застосовує механізм уваги паралельно до кожної проєкції, а потім конкатенує результати та проєктує їх назад до вихідної розмірності:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

$$\text{де } \text{head}_i = \text{Attention}(QW^{Q_i}, KW^{K_i}, VW^{V_i})$$

Кожна голова уваги може навчитися захоплювати різні типи відношень або залежностей у послідовності. Емпіричні дослідження показують, що різні голови спеціалізуються на різних лінгвістичних явищах: деякі фокусуються на синтаксичних залежностях, інші – на семантичних відношеннях, треті – на

позиційній близькості. Типова конфігурація використовує 8 або 16 голів уваги, хоча це число може варіюватися залежно від розміру моделі.

Позиційне кодування (positional encoding) є критично важливим компонентом трансформерної архітектури, оскільки сам механізм уваги є інваріантним до перестановки елементів послідовності. Без інформації про порядок елементів модель не могла б розрізнити речення "кіт гонить собаку" та "собака гонить кота". Позиційне кодування додає інформацію про абсолютну або відносну позицію токенів у послідовності до їхніх ембедінгів. Оригінальна робота використовувала фіксовані синусоїдальні функції для кодування позицій:

$$PE(pos, 2i) = \sin(pos / 10000^{(2i/d_{\text{model}})})$$

$$PE(pos, 2i+1) = \cos(pos / 10000^{(2i/d_{\text{model}})})$$

де pos – позиція токена, i – індекс розмірності, а d_{model} – розмірність моделі. Вибір синусоїдальних функцій був мотивований їхньою здатністю легко екстраполюватися на послідовності довші за ті, що зустрічалися під час навчання. Альтернативні підходи включають навчувані позиційні ембедінги, відносно позиційне кодування та ротаційні позиційні ембедінги (RoPE), які показали переваги у деяких застосуваннях.

Позиційно-незалежна мережа прямого поширення (position-wise feed-forward network) застосовується незалежно до кожної позиції у послідовності та складається з двох лінійних перетворень з нелінійною активацією між ними:

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

де W_1 та W_2 – матриці ваг, b_1 та b_2 – вектори зсуву. Типово, внутрішня розмірність (розмір проміжного шару) у чотири рази більша за розмірність моделі, що дозволяє мережі навчати багаті нелінійні перетворення. Оригінальна архітектура використовувала функцію активації ReLU, хоча сучасні варіанти часто застосовують GeLU (Gaussian Error Linear Unit) або SwiGLU, які показують кращу ефективність.

Сучасний ландшафт великих мовних моделей характеризується різноманітністю архітектурних підходів, кожен з яких оптимізований для різних типів завдань. Умовно, LLM можна класифікувати за архітектурою на три основні типи: decoder-only моделі, encoder-only моделі та encoder-decoder моделі. Кожен тип має свої переваги та найкраще підходить для певних класів задач.

Серія моделей GPT (Generative Pre-trained Transformer) від OpenAI представляє decoder-only архітектуру та стала парадигмальною для генеративних застосувань. GPT-2, представлена у 2019 році, містила 1.5 мільярда параметрів та демонструвала вражаючі можливості у генерації когерентних текстів, хоча спочатку OpenAI не опублікували повну модель через побоювання зловживань. GPT-3, випущена у 2020 році, масштабувала архітектуру до 175 мільярдів параметрів та продемонструвала явище емерджентності (emergence), коли просте збільшення розміру моделі призводило до якісно нових можливостей, таких як ефективне few-shot навчання. Браун та співавтори показали, що GPT-3 може виконувати різноманітні завдання, просто на основі кількох прикладів у промпті, без необхідності градієнтного оновлення параметрів.

GPT-4, анонсована у 2023 році, представляє значний крок вперед у можливостях моделі, хоча конкретні архітектурні деталі залишаються закритими. За офіційною інформацією OpenAI, модель демонструє покращення у точності, здатності до міркування, обробці довгого контексту та мультимодальних можливостях, обробляючи як текст, так і зображення. Ключовою особливістю архітектури GPT є автогресивна природа генерації, де модель передбачає наступний токен на основі всіх попередніх токенів, використовуючи каузальне маскування у механізмі само-уваги для забезпечення, що інформація може текти лише в прямому напрямку.

BERT (Bidirectional Encoder Representations from Transformers), представлена Девліном та співавторами у 2018 році, використовує encoder-only архітектуру та революціонізувала підходи до розуміння природної мови. На відміну від автогресивних моделей GPT, BERT використовує двонаправлений контекст, що

дозволяє кожному токеноу враховувати інформацію як з лівого, так і з правого контексту одночасно. Ключовою інновацією BERT було введення завдання маскованого моделювання мови (Masked Language Modeling, MLM), де випадкові токени у вхідній послідовності замінюються спеціальним токеном [MASK], а модель навчається передбачати оригінальні токени на основі оточуючого контексту.

BERT також використовувала завдання передбачення наступного речення (Next Sentence Prediction, NSP) для навчання моделі розуміти відношення між парами речень. Базова версія BERT містила 110 мільйонів параметрів (12 шарів, 768 прихованих одиниць, 12 голів уваги), тоді як велика версія BERT-Large мала 340 мільйонів параметрів (24 шари, 1024 приховані одиниці, 16 голів). RoBERTa (Robustly Optimized BERT Approach), представлена Liu та співавторами у 2019 році, покращила підхід BERT через більш тривале навчання на більших корпусах, видалення завдання NSP та динамічне маскування. Ці модифікації призвели до значних покращень у якості на різноманітних NLP-бенчмарках.

Encoder-decoder архітектура, що використовує повну трансформерну структуру з обома компонентами, представлена моделями T5 (Text-to-Text Transfer Transformer) та BART (Bidirectional and Auto-Regressive Transformers). T5, розроблена командою Google у 2019 році, застосувала уніфікований text-to-text підхід, де всі NLP-завдання формулюються як перетворення вхідного тексту на вихідний текст. Це дозволило використовувати єдину архітектуру для різноманітних завдань, від перекладу та резюмування до класифікації та питально-відповідних систем. T5 навчалася на величезному корпусі C4 (Colossal Clean Crawled Corpus), що містив 750 ГБ англomовного тексту, та була випущена у різних розмірах від T5-Small (60 мільйонів параметрів) до T5-11B (11 мільярдів параметрів).

BART, представлена Lewis та співавторами, комбінує переваги BERT та GPT шляхом навчання як енкодера для розуміння контексту, так і декодера для генерації послідовностей. BART навчається через процес шумоподавлення

(denoising), де вхідний текст спотворюється різними способами (маскування токенів, видалення фрагментів, переміщення речень, ротація документа), а модель навчається реконструювати оригінальний текст. Ця стратегія навчання робить BART особливо ефективною для генеративних завдань, таких як абстрактне резюмування, переклад та генерація відповідей на питання.

Claude, розроблена компанією Anthropic, представляє сучасну лінію великих мовних моделей, що фокусуються на безпеці, корисності та чесності. Хоча конкретні архітектурні деталі залишаються комерційною таємницею, Claude базується на трансформерній архітектурі та навчається з використанням техніки Constitutional AI (CAI), що поєднує навчання з підкріпленням від людської оцінки (RLHF) з принципами, визначеними набором правил та цінностей. Актуальна версія Claude 3 включає три варіанти різної потужності: Opus (найбільша та найздатніша модель), Sonnet (збалансована модель для більшості застосувань) та Haiku (найшвидша та найкомпактніша модель). Claude демонструє сильні можливості у довгому контексті, обробляючи до 200 тисяч токенів, що дозволяє аналізувати цілі книги або великі кодові бази за один запит.

Відкриті великі мовні моделі відіграють критично важливу роль у демократизації доступу до передових AI-технологій. LLaMA (Large Language Model Meta AI), представлена Meta AI у 2023 році, стала важливою віхою у розробці відкритих моделей. LLaMA 2 була випущена у варіантах від 7 до 70 мільярдів параметрів та навчена на 2 трильйонах токенів. Ключовою особливістю LLaMA є ефективність архітектури, що дозволяє меншим моделям досягати конкурентної якості порівняно з набагато більшими закритими моделями. LLaMA використовує групову нормалізацію запитів (Grouped-Query Attention, GQA) для покращення ефективності інференсу та ротаційні позиційні ембедінги (RoPE) для кращої обробки довгих послідовностей.

Mistral AI розробила серію високоефективних відкритих моделей, включаючи Mistral 7B та Mixtral 8x7B. Mistral 7B, незважаючи на відносно скромний розмір у 7 мільярдів параметрів, демонструє якість, що перевершує

багато більших моделей, завдяки оптимізованій архітектурі та ретельному процесу навчання. Mixtral 8x7B використовує архітектуру Mixture of Experts (MoE), де модель складається з восьми експертних підмереж, але для обробки кожного токена активується лише дві експертні мережі. Це дозволяє досягти якості моделі з 47 мільярдами параметрів при обчислювальних витратах моделі з 13 мільярдами параметрів, значно покращуючи співвідношення якості до ефективності.

Усі розглянуті моделі можна порівняти в табл 1.2

Таблиця 1.2 - Порівняння великих мовних моделей

Модель	Архітекту ра	Параметр и	Контекст не вікно	Особливості	Доступні сть
GPT-4	Decoder-only	Не розкрито (~1.76T)	128K токенів	Мультимодальність, висока якість міркування	Комерційна API
GPT-3.5	Decoder-only	175B	16K токенів	Швидкий інференс, добра якість	Комерційна API
Claude 3 Opus	Decoder-only	Не розкрито	200K токенів	Довгий контекст, Constitutional AI	Комерційна API
Claude 3 Sonnet	Decoder-only	Не розкрито	200K токенів	Збалансована якість/швидкість	Комерційна API
BERT-Large	Encoder-only	340M	512 токенів	Двонаправлений контекст	Відкрита
RoBERTa-Large	Encoder-only	355M	512 токенів	Покращене навчання BERT	Відкрита

Продовження таблиці 1.2

T5-11B	Encoder-d ecoder	11B	512 токенів	Уніфікований text-to-text підхід	Відкрита
BART- Large	Encoder-d ecoder	400M	1024 токени	Ефективна для генерації	Відкрита
LLaMA 2 70B	Decoder-o nly	70B	4096 токенів	Ефективна архітектура, GQA	Відкрита
Mistral 7B	Decoder-o nly	7B	8192 токени	Висока якість при малому розмірі	Відкрита
Mixtral 8x7B	Decoder-o nly (MoE)	47B (13B активних)	32K токенів	Mixture of Experts, ефективний інференс	Відкрита

Навчання сучасних великих мовних моделей є складним багатоетапним процесом, що вимагає величезних обчислювальних ресурсів та ретельно керованих даних. Процес зазвичай складається з кількох послідовних етапів, кожен з яких має специфічні цілі та методології. Розуміння цих етапів є критично важливим для ефективного застосування LLM у практичних системах та для усвідомлення їхніх можливостей та обмежень.

Попереднє навчання (pre-training) становить фундаментальний етап, на якому модель набуває загальних мовних здібностей через самонавчання на величезних текстових корпусах. Цей процес не вимагає розмічених даних, оскільки використовує структуру самого тексту для генерації навчальних сигналів. Для decoder-only моделей, таких як GPT, завдання попереднього навчання полягає у передбаченні наступного токена (next token prediction): дано послідовність токенів $w_1, w_2, \dots, w_{\square-1}$, модель навчається передбачати $w_{\square}$. Функція втрат максимізує логарифмічну правдоподібність послідовності: $L = -\sum \log P(w_{\square} | w_1, \dots, w_{\square-1}; \theta)$, де θ позначає параметри моделі.

Для encoder-only моделей, таких як BERT, використовується маскований мовний моделювання (Masked Language Modeling), де випадковим чином маскується певний відсоток токенів (зазвичай 15%), і модель навчається передбачати їх на основі двонаправленого контексту. Encoder-decoder моделі можуть використовувати різноманітні завдання самонавчання, включаючи span corruption (T5), де випадкові фрагменти тексту замінюються sentinel токенами, і модель навчається генерувати оригінальні фрагменти, або denoising (BART), де текст спотворюється різними способами, і модель навчається відновлювати оригінал.

Масштаб попереднього навчання для сучасних LLM є вражаючим. GPT-3 навчалася на корпусі приблизно 500 мільярдів токенів, що включав відфільтровану версію Common Crawl (410 мільярдів токенів), WebText2 (19 мільярдів токенів), Books1 та Books2 (67 мільярдів токенів) та Wikipedia (3 мільярди токенів). LLaMA 2 навчалася на 2 трильйонах токенів, демонструючи тренд до збільшення розмірів навчальних корпусів. Попереднє навчання вимагає тисяч GPU або TPU та може тривати тижні або навіть місяці безперервних обчислень. Наприклад, навчання GPT-3 оцінюється у приблизно 3.14×10^{23} FLOPS (операцій з плаваючою комою), що еквівалентно приблизно 355 GPU-рокам на NVIDIA V100.

Дотренування або тонка настройка (fine-tuning) адаптує попередньо навчену модель до конкретних завдань або доменів. На цьому етапі модель навчається на відносно невеликих наборах розмічених даних, специфічних для цільової задачі. Процес дотренування здійснюється з набагато меншою швидкістю навчання порівняно з попереднім навчанням, щоб уникнути катастрофічного забування (catastrophic forgetting) – феномену, коли модель втрачає загальні мовні здібності, набуті під час попереднього навчання. Для класичних завдань NLP, таких як класифікація текстів, розпізнавання іменованих сутностей або питально-відповідні системи, дотренування часто включає додавання спеціалізованого класифікаційного шару на вершині попередньо навченого енкодера та спільне навчання всієї моделі на задачі специфічних даних.

Однак для генеративних застосувань та чат-ботів процес значно складніший. Supervised fine-tuning (SFT) виконується на високоякісних прикладах бажаної поведінки, зазвичай зібраних від людських асистентів або анотаторів. Ці приклади демонструють бажані відповіді на різноманітні запити, включаючи надання інформації, виконання завдань, відмову у небезпечних запитах тощо. Проте навіть після SFT моделі можуть генерувати невідповідні, необ'єктивні або небезпечні відповіді. Саме тут критично важливим стає наступний етап навчання.

Навчання з підкріпленням від людської оцінки (Reinforcement Learning from Human Feedback, RLHF) стало ключовою інновацією, що дозволила створити безпечні та корисні асистентські моделі, такі як ChatGPT, Claude та інші. RLHF вирішує фундаментальну проблему: як навчити модель генерувати відповіді, які люди вважають корисними, правдивими та безпечними, коли ці якості складно формалізувати у традиційну функцію втрат. Процес RLHF складається з трьох основних етапів. По-перше, модель винагороди (reward model) навчається передбачувати людські переваги. Людським оцінювачам показують кілька відповідей моделі на одне й те саме питання, і вони ранжують їх за якістю. На цих даних навчається окрема нейронна мережа, яка приймає пару (запит, відповідь) та

видає скалярну оцінку винагороди, що відображає передбачувану якість відповіді з точки зору людини.

По-друге, мовна модель оптимізується для максимізації очікуваної винагороди від моделі винагороди, використовуючи алгоритм навчання з підкріпленням, зазвичай Proximal Policy Optimization (PPO). Мовна модель розглядається як політика, що генерує дії (токени) для максимізації винагороди. Функція цілі включає не лише винагороду від моделі винагороди, але й штраф за відхилення від попередньо навченої моделі (KL-дивергенцію), що запобігає надмірній оптимізації та втраті загальних мовних здібностей: $\text{objective}(\phi) = E[r_\theta(x, y)] - \beta \cdot \text{DKL}[\pi_\phi \parallel \pi_{\text{SFT}}]$, де π_ϕ – поточна політика (мовна модель), π_{SFT} – модель після supervised fine-tuning, r_θ – модель винагороди, β – коефіцієнт штрафу за KL-дивергенцію.

По-третє, процес ітеративно повторюється: модель генерує нові відповіді, люди їх оцінюють, модель винагороди оновлюється, і мовна модель далі оптимізується. Цей ітеративний процес дозволяє постійно покращувати якість відповідей та адаптуватися до змінних вимог та очікувань користувачів. RLHF виявився надзвичайно ефективним у створенні моделей, які не лише технічно компетентні, але й корисні, чесні та безпечні у взаємодії з людьми.

Constitutional AI (CAI), розроблена Anthropic для навчання моделей Claude, представляє альтернативний підхід, що зменшує залежність від людської оцінки на більш пізніх етапах навчання. CAI використовує набір принципів або "конституцію", що визначає бажану поведінку моделі. Спочатку модель генерує відповіді, самостійно критикує їх на основі конституційних принципів та генерує покращені версії. Ці покращені відповіді використовуються для supervised learning. Потім створюється модель переваг, де модель сама генерує порівняння відповідей на основі конституції, замість покладання на людські оцінки. Нарешті, ця модель переваг використовується для RLHF-подібного процесу оптимізації. CAI дозволяє більш прозоро контролювати цінності та поведінку моделі через явні принципи, а не імпліцитні людські судження.

Few-shot та zero-shot навчання представляють парадигмальну зміну у застосуванні великих мовних моделей, дозволяючи адаптувати їх до нових завдань без градієнтних оновлень параметрів. Zero-shot навчання означає, що модель виконує завдання без жодних спеціалізованих навчальних прикладів, виключно на основі інструкцій у промті. Наприклад, модель може бути попрошена "Переклади наступне речення українською мовою:" без надання прикладів перекладів. Few-shot навчання надає модальності кілька прикладів бажаної поведінки у контексті запиту, після чого модель виконує аналогічне завдання на новому вході. Ці підходи стали можливими завдяки явищу in-context learning – здатності великих мовних моделей навчатися новим завданням просто через спостереження прикладів у контексті, без оновлення ваг мережі.

Дослідження показують, що здатність до few-shot та zero-shot навчання масштабується з розміром моделі та є одним з проявів емерджентності у великих мовних моделях. Невеликі моделі зазвичай не демонструють значних покращень від додавання прикладів у промті, тоді як моделі масштабу GPT-3 та більші показують драматичні покращення якості при наданні навіть кількох прикладів. Це відкриває можливості для демократизації доступу до AI-технологій, оскільки користувачі без експертизи у машинному навчанні можуть адаптувати моделі до своїх потреб через ретельне конструювання промтів.

Розмір моделі, що вимірюється кількістю параметрів, є фундаментальною характеристикою, яка сильно корелює з її можливостями, хоча й не є єдиним визначальним фактором. Параметри моделі включають всі навчувані ваги у нейронній мережі: матриці ваг у шарах само-уваги, feed-forward мережах, ембедінгах та інших компонентах. Для трансформерної архітектури кількість параметрів можна приблизно оцінити як: $N_{\text{params}} \approx 12 \times n_{\text{layers}} \times d_{\text{model}}^2$, де n_{layers} – кількість трансформерних шарів, а d_{model} – розмірність моделі. Множник 12 відображає приблизний внесок само-уваги (4×), feed-forward мережі (8×) та інших компонентів.

Еволюція великих мовних моделей демонструє стійку тенденцію до збільшення розміру. ELMo (2018) мала приблизно 94 мільйони параметрів, BERT-Large (2018) – 340 мільйонів, GPT-2 (2019) – 1.5 мільярда, GPT-3 (2020) – 175 мільярдів, а найбільші сучасні моделі можуть містити трильйони параметрів. Це збільшення масштабу не є просто кількісним, а призводить до якісних змін у можливостях моделей. Емпірично спостерігається закон масштабування (scaling law), згідно з яким продуктивність моделі передбачувано покращується зі збільшенням розміру моделі, обсягу навчальних даних та обчислювальних ресурсів, дотримуючись степеневих залежностей.

Однак більший розмір також означає значно вищі вимоги до обчислювальних ресурсів. Модель GPT-3 з 175 мільярдами параметрів у повній точності (FP32) займає приблизно 700 ГБ пам'яті, що вимагає використання кількох високопродуктивних GPU або TPU для інференсу. Навіть у половинній точності (FP16) модель займає 350 ГБ. Це зробило необхідним розвиток технік оптимізації, таких як квантизація (зменшення точності представлення ваг до 8-бітних або навіть 4-бітних чисел), pruning (видалення найменш важливих з'єднань), distillation (навчання меншої моделі імітувати поведінку більшої) та inference optimization (оптимізація для швидкого інференсу).

Контекстне вікно або довжина контексту визначає максимальну кількість токенів, яку модель може обробити за один раз. Ця характеристика критично важлива для практичних застосувань, оскільки визначає, скільки інформації модель може врахувати при генерації відповіді. Ранні трансформерні моделі мали обмежені контекстні вікна: оригінальний Transformer – 512 токенів, BERT – 512 токенів, GPT-2 – 1024 токени. Це обмеження було пов'язане з квадратичною складністю механізму уваги відносно довжини послідовності: обчислення само-уваги вимагає $O(n^2)$ операцій та пам'яті, де n – довжина послідовності.

Сучасні моделі значно розширили контекстні вікна через архітектурні інновації та збільшення обчислювальних ресурсів. GPT-3.5 підтримує 16,384 токени, GPT-4 – до 128,000 токенів (приблизно 300 сторінок тексту), Claude 3 – до

200,000 токенів. Ці розширені контексти дозволяють обробляти цілі книги, великі кодові бази, детальні технічні документації або тривалі діалоги без втрати контекстуальної інформації. Для досягнення таких довгих контекстів використовуються різноманітні техніки, включаючи sparse attention (розподіжена увага, що обчислює уваги лише для підмножини пар позицій), sliding window attention (ковзне вікно уваги), flash attention (оптимізований алгоритм обчислення уваги, що мінімізує звертання до пам'яті) та інші підходи.

Швидкість інференсу є критичним параметром для практичних застосувань, особливо для інтерактивних систем та застосувань реального часу. Швидкість зазвичай вимірюється у токенах за секунду або у латентності (затримці) до першого токена та між токенами. Для автогресивних моделей, таких як GPT, кожен токен генерується послідовно, що означає, що час генерації відповіді довжиною n токенів вимагає приблизно n проходів через модель. Типова швидкість інференсу для GPT-3 становить приблизно 10-20 токенів за секунду на одному GPU, хоча це може значно варіюватися залежно від довжини контексту, розміру batch'у та апаратного забезпечення.

Оптимізація швидкості інференсу є активною областю досліджень. Техніки включають speculative decoding (спекулятивне декодування, де менша швидка модель генерує кандидатів, які потім верифікуються більшою моделлю), continuous batching (безперервне пакетування запитів для максимізації використання GPU), KV-cache оптимізацію (ефективне кешування обчислень ключів та значень у механізмі уваги) та використання спеціалізованого апаратного забезпечення. Mixture of Experts архітектура, використана у Mixtral, також покращує ефективність інференсу, оскільки лише частина параметрів активна для кожного токена.

Порівняння характеристик наведено в табл 1.3.

Таблиця 1.3 - Порівняння характеристик великих мовних моделей

Характеристи ка	GPT-4	Claude Opus	3	LLaMA 2 70B	Mistral 7B	BERT-L arge
Параметри	~1.76T (оцінка)	Не розкрито		70B	7B	340M
Контекстне вікно	128K токенів	200K токенів		4K токенів	32K токенів	512 токенів
Швидкість інференсу	Висока (оптимізован о)	Висока		Середня	Дуже висока	Дуже висока
Вимоги (інференс)	GPU 8× (оцінка)	A100 8× (оцінка)		4× A100	1× A100	1× V100
Вимоги пам'яті	~3.5TB FP16	~2TB FP16 (оцінка)		~140GB FP16	~14GB FP16	~1.3GB FP16
Час навчання	Не розкрито	Не розкрито		~1.7M GPU-год ин	~350K GPU-год ин	~4 дні на 64 TPU

Вимоги до обчислювальних ресурсів для навчання великих мовних моделей є астрономічними. Навчання GPT-3 оцінюється у приблизно \$4.6 мільйонів у обчислювальних витратах. Навчання LLaMA 2 70B вимагало 1.7 мільйонів GPU-годин на NVIDIA A100-80GB. Ці цифри роблять попереднє навчання великих моделей доступним лише для організацій з значними ресурсами. Проте

зростання екосистеми відкритих моделей, таких як LLaMA, Mistral та Falcon, дозволяє дослідникам та практикам використовувати та адаптувати потужні попередньо навчені моделі без необхідності самостійного попереднього навчання.

Енергоспоживання та екологічний вплив великих мовних моделей також стають предметом зростаючого занепокоєння. Навчання великої мовної моделі може виробляти викиди CO₂, еквівалентні викидам від декількох автомобілів протягом їхнього повного життєвого циклу. Це стимулює розробку більш ефективних архітектур, таких як Mixture of Experts, оптимізацію алгоритмів навчання та використання відновлюваних джерел енергії для центрів обробки даних. Деякі дослідники аргументують, що фокус має зміститися з простого збільшення розміру моделей на покращення ефективності та розробку спеціалізованих моделей для конкретних застосувань.

Таким чином, великі мовні моделі представляють кульмінацію десятиліть досліджень у обробці природної мови та машинному навчанні. Еволюція від простих n-грамних моделей через рекурентні архітектури до трансформерів демонструє прогресивне подолання фундаментальних обмежень попередніх підходів. Архітектура трансформер, з її механізмами само-уваги та багатоголової уваги, дозволила досягти безпрецедентних можливостей у розумінні та генерації природної мови. Різноманітність сучасних LLM, від decoder-only моделей GPT до encoder-only BERT та encoder-decoder T5, відображає адаптацію базової архітектури до різних типів завдань. Складні багатоетапні процеси навчання, що включають попереднє навчання, дотренування та RLHF, дозволяють створювати моделі, які є не лише технічно потужними, але й корисними та безпечними у взаємодії з людьми. Розуміння цих принципів є критично важливим для ефективного застосування великих мовних моделей у системах автоматизованого лінгвістичного аналізу.

2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ПРОЕКТУВАННЯ СИСТЕМИ

2.1 Методи та технології автоматизованого лінгвістичного аналізу

Ефективна реалізація систем автоматизованого лінгвістичного аналізу вимагає комплексного застосування різноманітних методів та технологій, що охоплюють весь ланцюг обробки тексту від попередньої підготовки даних до оцінки якості отриманих результатів. У цьому підрозділі детально розглядаються ключові компоненти такої системи, включаючи методи попередньої обробки, векторні представлення, сучасні техніки взаємодії з великими мовними моделями, підходи до адаптації моделей та метрики оцінювання ефективності.

Попередня обробка тексту є критично важливим етапом, що закладає основу для всіх подальших операцій лінгвістичного аналізу. Якість попередньої обробки безпосередньо впливає на точність та ефективність моделей, оскільки навіть найсучасніші алгоритми не можуть компенсувати погано підготовлені дані. Попередня обробка включає послідовність трансформацій, що перетворюють неструктурований текст у формат, придатний для обчислювальної обробки.

Токенізація становить першочергове завдання та полягає у розбитті потоку тексту на дискретні одиниці – токени, які можуть бути словами, підсловами або символами залежно від обраного підходу. Класична токенізація на рівні слів (word-level tokenization) виділяє токени на основі розділювачів, таких як пробіли та розділові знаки. Для європейських мов цей підхід здається інтуїтивним: речення "Студент вивчає обробку природної мови" розбивається на токени ["Студент", "вивчає", "обробку", "природної", "мови"]. Проте навіть тут виникають складнощі з обробкою пунктуації, аббревіатур, чисел та складених слів. Наприклад, речення "Dr. Smith's office is open 9:00-17:00" вимагає складних правил для коректної обробки скорочення "Dr.", присвійного апострофа та часового діапазону.

Фундаментальна проблема word-level токенизації полягає у необхідності підтримувати великі словники, що охоплюють усі можливі словоформи. Для морфологічно багатих мов, таких як українська, де кожен іменник може мати понад десяток форм, розмір словника стає непрактично великим. Крім того, word-level підхід не може ефективно обробляти рідкісні слова, неологізми, власні назви та слова з помилками, які неминуче трапляються у реальних текстах. Всі такі слова потрапляють у категорію "невідомих" (out-of-vocabulary, OOV) та замінюються спеціальним токеном [UNK], що призводить до втрати інформації.

Підслівна токенизація (subword tokenization) вирішує ці проблеми шляхом розкладання слів на менші значущі одиниці. Byte Pair Encoding (BPE), запропонований Сеннрічем та співавторами у 2016 році, є одним з найпопулярніших алгоритмів підслівної токенизації. BPE починає з базового словника окремих символів та ітеративно об'єднує найчастіші пари суміжних tokenів у новий token. Алгоритм працює наступним чином:

1. Ініціалізувати словник всіма символами у корпусі
2. Додати спеціальний символ кінця слова `</w>` до кожного слова
3. Повторювати задану кількість ітерацій:
 - Знайти найчастішу пару суміжних tokenів у корпусі
 - Об'єднати цю пару у новий token
 - Додати новий token до словника
4. Токенізувати нові тексти, застосовуючи навчені операції злиття

Наприклад, якщо слово "найбільший" зустрічається часто, BPE може створити послідовність операцій злиття: "н"+"а" → "на", "на"+"й" → "най", "б"+"і" → "бі", "бі"+"л" → "біл", "біл"+"ь" → "біль", "біль"+"ш" → "більш", "най"+"більш" → "найбільш", "найбільш"+"и"+"й" → "найбільший". У результаті

рідкісні слова розбиваються на більш часті підслівні одиниці, що дозволяє ефективно представляти необмежений словник фіксованим набором токенів.

WordPiece, використовуваний у BERT та багатьох інших моделях, є схожим підходом, але використовує імовірнісний критерій для вибору пар для злиття замість простої частоти. WordPiece максимізує правдоподібність навчального корпусу згідно з мовною моделлю уніграм. На кожній ітерації обирається пара, що максимально збільшує правдоподібність:

$$\text{score}(x, y) = (\text{freq}(xy)) / (\text{freq}(x) \times \text{freq}(y))$$

де $\text{freq}(\cdot)$ позначає частоту у корпусі. WordPiece також використовує спеціальний префікс "##" для позначення підслівних токенів, що не є початком слова, наприклад: ["play", "##ing", "foot", "##ball"]. Це дозволяє легко відновити оригінальні слова після токенизації.

SentencePiece, розроблений командою Google, пропонує уніфікований підхід, що розглядає текст як послідовність Unicode символів без попереднього припущення про границі слів. Це робить SentencePiece особливо ефективним для мов без явних розділювачів слів, таких як китайська або японська, а також для обробки багатомовних корпусів. SentencePiece підтримує як BPE, так і уніграмний алгоритм сегментації та може навчатися безпосередньо на сирих текстах без попередньої токенизації.

Нормалізація тексту включає набір трансформацій, що приводять текст до стандартизованої форми для полегшення подальшої обробки. Основні операції нормалізації включають перетворення регістру (зазвичай до нижнього регістру, хоча для деяких завдань, таких як розпізнавання іменованих сутностей, регістр несе важливу інформацію), видалення або стандартизацію розділових знаків, нормалізацію пробілів (заміна множинних пробілів на одинарні), розгортання скорочень ("don't" → "do not") та нормалізацію чисел (заміна всіх чисел на спеціальний токен [NUM] або нормалізація форматів дат). Для мов з складною

орфографією нормалізація може включати приведення до стандартної орфографії, видалення діакритичних знаків або транслітерацію.

Видалення стоп-слів полягає у фільтрації високочастотних службових слів, що несуть мало семантичної інформації, таких як артиклі, прийменники, сполучники та займенники. Для української мови стоп-словами можуть бути "і", "в", "на", "з", "до", "який", "що", "цей" тощо. Історично, видалення стоп-слів було важливим для зменшення розмірності та шуму у класичних методах, таких як bag-of-words та TF-IDF. Проте у сучасних нейронних підходах роль видалення стоп-слів зменшилася, оскільки моделі здатні автоматично навчитися ігнорувати нерелевантну інформацію. Більше того, для деяких завдань, таких як аналіз синтаксису або питально-відповідні системи, стоп-слова можуть нести важливу структурну інформацію.

Лематизація та стеммінг є техніками приведення словоформ до базової форми, що дозволяє об'єднати різні граматичні варіанти одного слова. Стеммінг виконує грубе відсікання афіксів за формальними правилами без глибокого лінгвістичного аналізу. Алгоритм Портера, розроблений у 1980 році, є найвідомішим алгоритмом стеммінгу для англійської мови та використовує послідовність правил відсікання суфіксів. Для слова "running" алгоритм відсікає суфікс "-ing", отримуючи основу "run". Проте стеммінг може призводити до неіснуючих слів: "university" → "univers", "policy" → "polici". Для української мови існують адаптації, такі як алгоритм Snowball Ukrainian stemmer.

Лематизація є більш складним процесом, що використовує морфологічний аналіз та словники для визначення лемми – канонічної словникової форми слова. Лематизація враховує частину мови та контекст вживання слова. Для українського дієслова "біжу" лематизація поверне інфінітив "бігти", для іменника "студентів" – називний однини "студент". Лематизація завжди повертає реальні слова, що робить її результати більш інтерпретованими. Сучасні інструменти, такі як spaCy, stanza або UDPipe, забезпечують високоякісну лематизацію для багатьох мов,

включаючи українську, використовуючи нейронні моделі, навчені на анотованих корпусах.

Векторні представлення слів (word embeddings) стали фундаментальним інструментом сучасної обробки природної мови, дозволяючи представляти дискретні текстові одиниці у вигляді неперервних векторів у багатовимірному просторі. Ключова ідея полягає у гіпотезі дистрибутивної семантики: слова, що вживаються у схожих контекстах, мають схожі значення. Векторні представлення кодують цю контекстуальну схожість як геометричну близькість у векторному просторі.

Word2Vec, представлений Міколовим та співавторами у 2013 році, став проривом у створенні ефективних векторних представлень слів. Word2Vec пропонує дві архітектури: Continuous Bag-of-Words (CBOW) та Skip-gram. CBOW передбачає цільове слово на основі контекстних слів, тоді як Skip-gram робить навпаки – передбачає контекстні слова на основі цільового слова. Формально, для Skip-gram максимізується логарифмічна правдоподібність:

$$L = (1/T) \sum_{i=1}^T \sum_{-k \leq j \leq k, j \neq 0} \log P(w_{i+j} | w_i)$$

де T – загальна кількість слів, k – розмір контекстного вікна, а умовна ймовірність обчислюється через softmax над всім словником:

$$P(w_o | w_i) = \exp(v_o^T v_i) / \sum_{v \in V} \exp(v^T v_i)$$

де v_i та v_o – векторні представлення вхідного та вихідного слова відповідно. Для прискорення навчання Word2Vec використовує negative sampling або hierarchical softmax замість обчислення повного softmax над словником.

Отримані векторні представлення демонструють вражаючі властивості. Арифметичні операції над векторами відображають семантичні відношення: $\text{vector}(\text{"король"}) - \text{vector}(\text{"чоловік"}) + \text{vector}(\text{"жінка"}) \approx \text{vector}(\text{"королева"})$. Косинусна схожість між векторами корелює з семантичною близькістю слів.

Слова з схожими значеннями, такі як синоніми, утворюють кластери у векторному просторі. Word2Vec стали стандартним інструментом для ініціалізації ембедінгів у нейронних мережах та для різноманітних NLP-завдань.

GloVe (Global Vectors for Word Representation), розроблений командою Stanford у 2014 році, пропонує альтернативний підхід, що явно використовує глобальну статистику спільної появи слів. GloVe навчає векторні представлення шляхом факторизації матриці спільної появи слів, мінімізуючи зважену функцію втрат:

$$J = \sum_{i,j} f(X_{ij})(v_i^T v'_j + b_i + b'_j - \log X_{ij})^2$$

де X_{ij} – кількість разів, коли слово j з'являється у контексті слова i , v_i та v'_j – векторні представлення, b_i та b'_j – скалярні зсуви, а $f(\cdot)$ – вагова функція, що зменшує вплив рідкісних спільних появ. GloVe поєднує переваги методів підрахунку (таких як LSA) з перевагами методів передбачення (таких як Word2Vec), демонструючи конкурентну або кращу якість на багатьох бенчмарках.

Фундаментальне обмеження Word2Vec та GloVe полягає у тому, що вони генерують статичні представлення: кожне слово має єдине фіксоване векторне представлення незалежно від контексту вживання. Це проблематично для багатозначних слів (полісемів), коли слово має різні значення у різних контекстах. Наприклад, слово "банк" у реченнях "банк видав кредит" та "береговий банк річки" має різні значення, але отримає ідентичне векторне представлення у Word2Vec/GloVe.

Контекстуальні embeddings вирішують цю проблему, генеруючи різні векторні представлення для одного й того ж слова залежно від контексту його вживання. ELMo (Embeddings from Language Models), представлена Петерсом та співавторами у 2018 році, використовує двонаправлену LSTM для генерації контекстуалізованих представлень. ELMo навчає двонаправлену мовну модель на великому корпусі: прямий LSTM передбачає наступне слово, зворотний –

попереднє. Для кожного токена у послідовності ELMo обчислює представлення з усіх внутрішніх шарів LSTM та формує фінальне представлення як зважену суму:

$$\text{ELMo}_{\square}^{\text{task}} = \gamma^{\text{task}} \sum_{\square=0}^L s_{\square}^{\text{task}} h_{\square, \square}^{\text{LM}}$$

де $h_{\square, \square}^{\text{LM}}$ – вихід j -го шару для токена k , $s_{\square}^{\text{task}}$ – навчувані ваги для конкретної задачі, γ^{task} – скалярний параметр. ELMo значно покращила якість на багатьох NLP-завданнях, демонструючи, що різні шари LSTM захоплюють різні аспекти мовної інформації: нижні шари кодують синтаксис, вищі – семантику.

BERT embeddings представляють наступний крок еволюції контекстуальних представлень. На відміну від ELMo, яка використовує конкатенацію незалежно навчених прямого та зворотного LSTM, BERT використовує трансформерний енкодер для справді двонаправленого моделювання контексту. Кожен токен у BERT отримує представлення, що враховує весь ліворуч та праворуч контекст одночасно через механізми само-уваги. Представлення токена у BERT є виходом відповідної позиції з останнього шару трансформера, хоча на практиці часто використовують конкатенацію або зважену суму виходів з декількох останніх шарів для кращої якості.

BERT embeddings продемонстрували драматичне покращення якості на широкому спектрі завдань розуміння мови, включаючи класифікацію текстів, розпізнавання іменованих сутностей, питально-відповідні системи та аналіз синтаксису. Контекстуальність представлень дозволяє природно обробляти багатозначність: слово "банк" отримає різні представлення у фінансовому та географічному контекстах, оскільки оточуючі слова через механізм уваги формуватимуть різні векторні репрезентації.

Sentence embeddings розширюють ідею векторних представлень з рівня слів на рівень речень або документів. Прості підходи включають усереднення word embeddings усіх слів у реченні, зважене усереднення з використанням TF-IDF ваг або використання останнього прихованого стану рекурентної мережі. Проте ці

методи часто не захоплюють композиційну семантику речення адекватно. Спеціалізовані моделі sentence embeddings, такі як Sentence-BERT (SBERT), адаптують архітектуру BERT для генерації якісних представлень речень шляхом fine-tuning на завданнях семантичної схожості речень з використанням siamese або triplet мережевих структур.

Universal Sentence Encoder (USE) від Google пропонує попередньо навчені моделі для різних мов, що генерують 512-вимірні векторні представлення для речень, оптимізовані для вимірювання семантичної схожості. Застосування embeddings для аналізу охоплюють кластеризацію документів (групування схожих текстів на основі косинусної схожості їхніх векторних представлень), семантичний пошук (ранжування документів за релевантністю до запиту у векторному просторі), класифікацію текстів (використання embeddings як ознак для класифікаторів), виявлення аномалій (ідентифікація текстів, що значно відрізняються від типових у векторному просторі) та багато інших завдань.

Методи векторного представлення тексту можна порівняти в табл 2.1.

Таблиця 2.1 - Порівняльна характеристика методів векторного представлення тексту

Тип представлення	Контекстуальність	Розмірність	Переваги	Недоліки
Word2Vec/GloVe	Статичні	50-300	Швидкі, компактні, інтерпретовні	Не враховують багатозначність
ELMo	Контекстуальні	1024	Враховують контекст, різні шари	Повільні, базовані на LSTM
BERT embeddings	Контекстуальні	768-1024	Двонаправлені, високоякісні	Потребують попереднього навчання
Sentence-BERT	Контекстуальні (речення)	384-768	Оптимізовані для семантичної схожості	Специфічні для певних завдань
USE	Контекстуальні (речення)	512	Багатомовні, готові до використання	Фіксована архітектура

Промпт-інженерія є критично важливою дисципліною для ефективного застосування великих мовних моделей, що фокусується на розробці та оптимізації текстових інструкцій для направлення поведінки моделі. На відміну від традиційного підходу, де для кожної задачі навчається окрема модель, промпт-інженерія дозволяє адаптувати універсальну попередньо навчену модель до специфічних завдань через ретельне формулювання запитів без зміни параметрів моделі. Це революціонізувало процес розробки NLP-систем, зробивши передові можливості доступними без необхідності у великих обсягах розмічених даних та обчислювальних ресурсах для навчання.

Принципи створення ефективних промптів базуються на розумінні того, як великі мовні моделі обробляють та інтерпретують текстові інструкції. По-перше, ясність та специфічність є фундаментальними: промпт має чітко визначати, що саме модель має зробити, який формат виходу очікується та які обмеження слід враховувати. Замість розпливчастого "Проаналізуй цей текст" краще використовувати "Визнач тональність наступного відгуку про продукт та класифікуй її як позитивну, негативну або нейтральну". По-друге, контекст та деталі покращують якість відповідей: надання релевантної фонові інформації, визначення цільової аудиторії або стилю допомагає моделі генерувати більш відповідні результати.

По-третє, структурованість промпту полегшує обробку: використання розділення на секції (інструкція, контекст, питання), нумерованих списків для багатокрокових завдань або спеціальних розділювачів підвищує ясність. По-четверте, ітеративне уточнення є необхідним: створення ефективного промпту часто вимагає експериментування з різними формулюваннями, порядком інформації та рівнем деталізації. По-п'яте, врахування обмежень моделі, таких як довжина контексту, схильність до певних типів відповідей або потенційні упередження, дозволяє передбачити та мітигувати можливі проблеми.

Zero-shot промптінг полягає у постановці завдання моделі без надання будь-яких прикладів, виключно на основі текстової інструкції. Цей підхід покладається на загальні знання та здібності, набуті моделлю під час попереднього навчання. Приклад zero-shot промпту для класифікації тональності:

Визнач тональність наступного тексту як позитивну, негативну або нейтральну.

Текст: "Обслуговування у ресторані було повільним, але страви виявилися смачними."

Тональність:

Zero-shot підхід є найпростішим та найшвидшим, не вимагає підготовки прикладів, проте якість може бути нижчою для складних або доменно-специфічних завдань, де модель не має достатнього попереднього досвіду.

Few-shot промптінг надає моделі кілька демонстраційних прикладів бажаної поведінки перед цільовим завданням. Це дозволяє моделі "навчитися" специфіці завдання у контексті запиту через процес in-context learning. Приклад few-shot промпту:

Класифікуй тональність відгуків про продукти.

Приклад 1:

Текст: "Відмінна якість! Дуже задоволений покупкою."

Тональність: позитивна

Приклад 2:

Текст: "Товар прийшов пошкодженим, розчарований."

Тональність: негативна

Приклад 3:

Текст: "Звичайний продукт, нічого особливого."

Тональність: нейтральна

Тепер класифікуй наступний текст:

Текст: "Швидка доставка, але продукт не відповідає опису."

Тональність:

Кількість прикладів зазвичай варіюється від 1-2 до 5-10 залежно від складності завдання та розміру контекстного вікна моделі. Дослідження показують, що навіть невелика кількість прикладів може драматично покращити якість порівняно з zero-shot підходом, особливо для завдань з специфічним форматом виходу або доменною термінологією.

Chain-of-Thought (CoT) промптинг, запропонований Вей та співавторами, є технікою, що заохочує модель до явного пояснення процесу міркування перед наданням фінальної відповіді. Замість прямого переходу від питання до відповіді, модель генерує проміжні кроки логічного ланцюжка. Це особливо ефективно для завдань, що вимагають багатокрокового мислення, арифметичних обчислень або логічного виведення. Приклад CoT промпу для лінгвістичного аналізу:

Виконай морфологічний аналіз слова, пояснюючи кожен крок.

Приклад:

Слово: "прочитаних"

Крок 1: Визначаю частину мови - це прикметник (або дієприкметник).

Крок 2: Визначаю граматичні ознаки - родовий відмінок, множина.

Крок 3: Визначаю основу - "прочита-".

Крок 4: Визначаю афікси - префікс "про-", суфікс "-н-", закінчення "-их".

Результат: Дієприкметник, родовий відмінок, множина, від дієслова "прочитати".

Тепер проаналізуй слово: "найкращого"

CoT промптінг значно покращує точність на завданнях, що вимагають міркування, хоча збільшує довжину генерованого тексту та час обробки. Варіації включають Zero-shot CoT, де додається проста інструкція "Поясни крок за кроком" без прикладів, та Self-Consistency, де генерується кілька ланцюжків міркування та обирається найчастіша фінальна відповідь.

Техніки оптимізації промптів включають налаштування температури (параметра, що контролює випадковість генерації: низька температура робить відповіді більш детермінованими, висока – більш різноманітними), використання системних промптів для визначення загального контексту та ролі асистента, декомпозицію складних завдань на послідовність простіших підзавдань, використання мета-промптів (промптів, що генерують інші промпти), застосування констрейнтів формату через спеціальні інструкції або приклади та ітеративне уточнення на основі аналізу помилок моделі.

Для лінгвістичних завдань специфічні промпти можуть включати:

Розпізнавання іменованих сутностей:

Витягни всі іменовані сутності з тексту та класифікуй їх за типами: PERSON (персона), ORG (організація), LOC (локація), DATE (дата).

Текст: "Компанія Google оголосила про відкриття нового офісу в Києві у грудні 2024 року."

Формат відповіді - JSON:

Аналіз синтаксичних зв'язків:

Побудуй дерево залежностей для речення, вказавши для кожного слова його головне слово та тип залежності.

Речення: "Студент читає цікаву книгу."

Формат:

[Слово] -> [Головне слово] (тип залежності)

Класифікація текстів з поясненням:

Класифікуй наступну новину за категоріями: ПОЛІТИКА, ЕКОНОМІКА, СПОРТ, ТЕХНОЛОГІЇ, КУЛЬТУРА. Поясни свій вибір, вказавши ключові слова та фрази, що вплинули на рішення.

Текст новини: [текст]

Категорія:

Пояснення:

Fine-tuning моделей

Хоча промпт-інженерія дозволяє адаптувати великі мовні моделі до багатьох завдань без зміни параметрів, існують ситуації, коли дотренування моделі є необхідним або бажаним підходом. Fine-tuning передбачає продовження навчання попередньо натренованої моделі на специфічному наборі даних з метою адаптації її до конкретного домену, завдання або стилю відповідей. Розуміння того, коли та як застосовувати fine-tuning, є критично важливим для побудови ефективних систем лінгвістичного аналізу.

Fine-tuning доцільний у кількох сценаріях. По-перше, коли промптінг не забезпечує достатньої якості для критичного застосування, особливо для завдань, що вимагають консистентності формату відповідей, точності у спеціалізованих доменах або обробки специфічної термінології. По-друге, для доменної адаптації, коли модель має обробляти тексти з високоспеціалізованої області (медицина, право, фінанси), де загальні знання попередньо навченої моделі є недостатніми. По-третє, для оптимізації латентності та витрат, оскільки fine-tuned модель може бути меншого розміру та швидшою при збереженні якості для конкретного завдання. По-четверте, коли необхідна приватність даних та неможливо передавати чутливу інформацію у промптах до зовнішніх API.

Повне дотренування (full fine-tuning) передбачає оновлення всіх параметрів моделі під час навчання на цільовому датасеті. Цей підхід потенційно дає найкращу якість, оскільки модель повністю адаптується до нового завдання. Процес включає ініціалізацію моделі попередньо навченими вагами, заміну або додавання task-specific шарів (наприклад, класифікаційної голови для завдань класифікації) та навчання з низькою швидкістю навчання (learning rate) для

уникнення катастрофічного забування. Типові значення learning rate для fine-tuning становлять $1e-5$ до $5e-5$, що є на порядок менше, ніж при навчанні з нуля.

Проте повне дотренування вимагає значних обчислювальних ресурсів, оскільки необхідно зберігати градієнти для всіх параметрів моделі. Для моделі з 7 мільярдами параметрів у половинній точності (FP16) лише зберігання моделі вимагає ~ 14 ГБ пам'яті, а з урахуванням градієнтів, оптимізаторних станів (для Adam) та активацій, загальна потреба у пам'яті може досягати $\sim 80-100$ ГБ, що перевищує можливості більшості споживчих GPU. Крім того, повне дотренування несе ризик overfitting на невеликих датасетах та втрати загальних мовних здібностей.

Parameter-Efficient Fine-Tuning (PEFT) методи вирішують ці проблеми шляхом оновлення лише невеликої підмножини параметрів при заморожуванні основної моделі. LoRA (Low-Rank Adaptation), запропонована Ху та співавторами у 2021 році, є одним з найефективніших PEFT підходів. LoRA базується на гіпотезі, що зміни у вагах під час fine-tuning мають низький внутрішній ранг. Замість оновлення всієї матриці ваг $W \in \mathbb{R}^{d \times k}$, LoRA заморожує W та навчає дві low-rank матриці $A \in \mathbb{R}^{d \times r}$ та $B \in \mathbb{R}^{r \times k}$, де $r \ll \min(d, k)$:

$$h = Wx + \Delta Wx = Wx + BAx$$

Під час інференсу матриці A та B можуть бути об'єднані з оригінальними вагами $W' = W + BA$, не вносячи додаткової латентності. LoRA зменшує кількість навчуваних параметрів на кілька порядків величини: для моделі з 7B параметрів LoRA може навчати лише 4-8 мільйонів параметрів при $r=8$, зменшуючи потребу у пам'яті до $\sim 20-30$ ГБ замість ~ 100 ГБ для повного fine-tuning. Емпірично, LoRA демонструє якість, сумірну з повним fine-tuning на багатьох завданнях, при цьому будучи набагато ефективнішою.

QLoRA (Quantized Low-Rank Adaptation) поєднує LoRA з квантизацією для подальшого зменшення вимог до пам'яті. QLoRA використовує 4-бітну квантизацію для заморожених параметрів базової моделі, зберігаючи LoRA адаптери у повній або половинній точності. Це дозволяє fine-tuning моделей з 65 мільярдами параметрів на одному споживчому GPU з 48 ГБ пам'яті. QLoRA використовує NormalFloat (NF4), спеціальний 4-бітний формат даних, оптимізований для квантизації нормально розподілених ваг нейронних мереж, та double quantization для квантизації самих констант квантизації.

Підготовка датасетів для fine-tuning вимагає ретельного курування та структурування даних. Датасет має бути репрезентативним для цільового застосування, збалансованим за класами (для класифікаційних завдань), достатнього обсягу (зазвичай від кількох сотень до десятків тисяч прикладів залежно від складності завдання) та високої якості (без помилок розмітки, дублікатів або шуму). Для інструкційного fine-tuning, який використовується для створення чат-ботів та асистентів, датасет структурується як пари "інструкція-відповідь" або діалоги:

```
json

{

  "instruction": "Визнач тональність відгуку",

  "input": "Продукт перевищив мої очікування, дуже задоволений!",

  "output": "Позитивна"

}
```

Для складніших завдань, таких як резюмування або генерація відповідей на питання, датасет може включати додаткові поля, такі як контекст, приклади або мета-інформацію. Важливо забезпечити розділення даних на навчальну (зазвичай

80-90%), валідаційну (5-10%) та тестову (5-10%) множини для запобігання overfitting та об'єктивної оцінки якості.

Гіперпараметри fine-tuning значно впливають на якість та стабільність навчання. Ключові гіперпараметри включають швидкість навчання (learning rate), зазвичай $1e-5$ до $5e-5$ для повного fine-tuning та $1e-4$ до $5e-4$ для LoRA; розмір батчу (batch size), що визначає кількість прикладів, оброблених одночасно, зазвичай обмежений доступною пам'яттю GPU; кількість епох навчання, зазвичай 3-5 епох для уникнення overfitting; планувальник швидкості навчання (learning rate scheduler), такий як linear decay з warmup, що поступово зменшує learning rate протягом навчання; та вага регуляризації (weight decay), що запобігає overfitting через L2 penalty на вагах.

Для LoRA специфічними гіперпараметрами є ранг r (зазвичай 8-64; більші значення дають більшу виразність але вимагають більше пам'яті), альфа-параметр (зазвичай встановлюється рівним r або $2r$, контролює масштаб адаптації) та вибір модулів для застосування LoRA (зазвичай застосовується до матриць query та value у механізмі уваги, іноді також до key та до feed-forward шарів). Моніторинг метрик на валідаційній множині під час навчання є критично важливим для раннього зупинення (early stopping) при виявленні overfitting.

Об'єктивна оцінка якості систем лінгвістичного аналізу вимагає використання відповідних метрик, що квантифікують різні аспекти продуктивності моделей. Вибір метрик залежить від типу завдання, специфіки предметної області та бізнес-цілей системи. Розуміння переваг та обмежень різних метрик є необхідним для адекватної інтерпретації результатів та порівняння альтернативних підходів.

Для завдань класифікації текстів використовується стандартний набір метрик, що оцінюють точність передбачень моделі. Ассигасу (точність) є найпростішою метрикою та визначається як відношення кількості правильних передбачень до загальної кількості передбачень:

$$\text{Accuracy} = (TP + TN) / (TP + TN + FP + FN)$$

де TP (True Positives) – правильно передбачені позитивні приклади, TN (True Negatives) – правильно передбачені негативні приклади, FP (False Positives) – неправильно передбачені як позитивні, FN (False Negatives) – неправильно передбачені як негативні. Accuracy є інтуїтивною та легко інтерпретованою метрикою, проте має суттєве обмеження: вона не враховує дисбаланс класів. Для несбалансованого датасету, де 95% прикладів належать одному класу, модель, що завжди передбачає цей клас, досягне 95% accuracy, не маючи реальної предиктивної сили.

Precision (точність) та Recall (повнота) надають більш детальну картину продуктивності, особливо для несбалансованих датасетів. Precision визначає, яка частка передбачених як позитивні прикладів дійсно є позитивними:

$$\text{Precision} = TP / (TP + FP)$$

Recall визначає, яку частку всіх реально позитивних прикладів модель правильно ідентифікувала:

$$\text{Recall} = TP / (TP + FN)$$

Ці метрики часто перебувають у trade-off відношенні: збільшення precision зазвичай зменшує recall та навпаки. Для балансування цих двох аспектів використовується F1-score – гармонічне середнє precision та recall:

$$F1 = 2 \times (\text{Precision} \times \text{Recall}) / (\text{Precision} + \text{Recall})$$

F1-score досягає максимального значення 1, коли precision та recall обидва рівні 1, та низьких значень, коли одна з метрик є низькою, навіть якщо інша висока. Для багатокласової класифікації ці метрики обчислюються для кожного класу окремо, а потім агрегуються через macro-averaging (просте усереднення по класах, надає рівну вагу кожному класу) або micro-averaging (обчислення глобальних TP, FP, FN через усі класи, надає рівну вагу кожному прикладу).

Для завдань генерації тексту оцінка якості є значно складнішою через відкритість та креативність задачі. BLEU (Bilingual Evaluation Understudy) є однією з найпоширеніших метрик для оцінки машинного перекладу та інших генеративних завдань. BLEU вимірює n-грамне перекриття між згенерованим текстом та референсними текстами, обчислюючи модифіковану точність для n-грам різної довжини (зазвичай від 1 до 4):

$$\text{BLEU} = \text{BP} \times \exp(\sum_{n=1}^N w_n \log p_n)$$

де p_n – точність n-грам, w_n – ваги (зазвичай $1/N$ для рівної ваги), BP (brevity penalty) – штраф за коротші переклади порівняно з референсом. BLEU повертає значення від 0 до 1 (або 0 до 100 у відсотках), де вищі значення вказують на кращу якість. Проте BLEU має суттєві обмеження: фокус на точності n-грам ігнорує синоніми та парафрази, метрика не враховує семантичну схожість, може надавати високі оцінки граматично некоректним текстам та погано корелює з людським судженням для креативних завдань.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) є сімейством метрик, спочатку розроблених для оцінки автоматичного резюмування. На відміну від BLEU, що фокусується на точності, ROUGE робить акцент на повноті. Основні варіанти включають ROUGE-N (перекриття n-грам між згенерованим та референсним резюме), ROUGE-L (найдовша спільна підпоследовність, що враховує порядок слів) та ROUGE-S (skip-bigram, що дозволяє пропуски між словами). ROUGE обчислюється як:

$$\text{ROUGE-N} = \sum_{c \in \text{Ref}} \sum_{a \in C} \text{Count}_{a,c}(n) / \sum_{c \in \text{Ref}} \sum_{a \in C} \text{Count}(n)$$

де $\text{Count}_{a,c}(n)$ – кількість n-грам у згенерованому тексті, що також присутні у референсі. ROUGE краще підходить для оцінки екстрактивного резюмування, де важлива повнота покриття ключових елементів оригіналу.

Perplexity (збентеження) є внутрішньою метрикою, що оцінює, наскільки добре мовна модель передбачає наступний токен у последовності. Математично,

perplexity визначається як експонента середньої негативної логарифмічної правдоподібності:

$$\text{Perplexity} = \exp(-1/N \sum_{i=1}^N \log P(w_i | w_1, \dots, w_{i-1}))$$

Нижчі значення perplexity вказують на краще моделювання мови. Perplexity корисна для порівняння мовних моделей на однакових тестових корпусах, проте не завжди корелює з якістю на downstream завданнях та може бути чутливою до словника та токенизації.

Семантична схожість оцінюється через косинусну відстань між векторними представленнями текстів. Для двох векторів u та v косинусна схожість обчислюється як:

$$\cos(u, v) = (u \cdot v) / (\|u\| \times \|v\|) = \sum_i u_i v_i / (\sqrt{\sum_i u_i^2} \times \sqrt{\sum_i v_i^2})$$

Значення варіюються від -1 (протилежні вектори) через 0 (ортогональні) до 1 (ідентичні напрямки). Для більшості NLP застосувань векторні представлення мають переважно позитивні компоненти, тому косинусна схожість зазвичай варіюється від 0 до 1. Семантична схожість використовується для оцінки якості sentence embeddings, пошуку схожих документів та кластеризації.

Human evaluation (людська оцінка) залишається золотим стандартом для оцінки якості складних генеративних завдань, де автоматичні метрики є недостатніми. Людські оцінювачі можуть судити про багатовимірні аспекти якості: когерентність (чи є текст логічно послідовним), релевантність (чи відповідає текст запиту), флюентність (чи є текст граматично правильним та природним), фактичну точність (чи містить текст правдиву інформацію) та корисність (чи задовольняє текст потреби користувача). Людська оцінка зазвичай проводиться через Likert шкали (наприклад, 1-5 або 1-7), порівняльні судження (А краще за В) або бінарні оцінки (прийнятно/неприйнятно).

Проте людська оцінка має обмеження: висока вартість та тривалість процесу, суб'єктивність та варіативність між оцінювачами, складність масштабування для великих датасетів та неможливість повторення з ідентичними умовами. Для забезпечення надійності використовуються множинні оцінювачі з обчисленням міжекспертної узгодженості (inter-rater agreement) через метрики такі як Cohen's Kappa або Krippendorff's Alpha, детальні інструкції та навчання оцінювачів та якісний аналіз розбіжностей для виявлення систематичних проблем.

У табл 2.2 наведено основні завдання обробки природної мови, відповідні метрики оцінювання, рекомендації щодо їх використання та основні обмеження. Представлені метрики застосовуються для оцінювання якості класифікації, машинного перекладу, резюмування, мовного моделювання, семантичної схожості та генерації тексту, що дозволяє обрати найбільш доцільний підхід залежно від типу задачі та особливостей даних.

Таблиця 2.2 - Метрики оцінювання якості моделей обробки природної мови

Завдання	Основні метрики	Коли використовувати	Обмеження
Класифікація	Accuracy, Precision, Recall, F1	Збалансовані та несбалансовані датасети	Не враховують семантику
Машинний переклад	BLEU, METEOR	Порівняння систем перекладу	Погана кореляція з креативністю
Резюмування	ROUGE, BLEU	Екстрактивне та абстрактне резюмування	Ігнорують синоніми
Мовна модель	Perplexity	Оцінка якості моделювання	Не корелює з downstream задачами
Семантична схожість	Cosine similarity	Пошук, кластеризація	Залежить від якості embeddings
Генерація тексту	Human evaluation, GPT-eval	Креативні та складні завдання	Дорого, суб'єктивно

Розгортання великих мовних моделей у продакшн середовищах вимагає балансування між якістю, швидкістю інференсу та ефективністю використання ресурсів. Техніки оптимізації дозволяють зменшити розмір моделей, прискорити обчислення та знизити вимоги до апаратного забезпечення без суттєвої втрати якості.

Квантизація (quantization) зменшує точність представлення ваг та активацій моделі, переходячи від 32-бітних або 16-бітних чисел з плаваючою комою до 8-бітних, 4-бітних або навіть бінарних представлень. Це зменшує розмір моделі та прискорює обчислення, оскільки операції з цілими числами є швидшими та енергоефективнішими. Post-training quantization застосовується до вже навченої моделі без додаткового навчання, використовуючи техніки такі як symmetric або asymmetric quantization для відображення плаваючих чисел на цілі:

$$x_quant = \text{round}((x - \text{zero_point}) / \text{scale})$$

де scale та zero_point – параметри квантизації, що визначаються з діапазону значень. Quantization-aware training (QAT) включає симуляцію квантизації під час навчання, дозволяючи моделі адаптуватися до зниженої точності та досягати кращої якості порівняно з post-training підходом.

Динамічна квантизація застосовує квантизацію лише до ваг, зберігаючи активації у повній точності, тоді як статична квантизація квантує як ваги, так і активації. Mixed-precision quantization використовує різну точність для різних шарів моделі, застосовуючи агресивнішу квантизацію до менш чутливих шарів. Дослідження показують, що 8-бітна квантизація зазвичай призводить до мінімальної деградації якості (< 1% відносної різниці), тоді як 4-бітна квантизація може вимагати calibration або QAT для збереження прийнятної якості.

Pruning (обрізання) видаляє найменш важливі з'єднання або нейрони з мережі, створюючи розріджену структуру, що вимагає менше обчислень та пам'яті. Magnitude-based pruning видаляє ваги з найменшими абсолютними

значеннями, виходячи з припущення, що малі ваги мало впливають на вихід мережі. Structured pruning видаляє цілі фільтри, канали або шари, створюючи архітектурно спрощену модель, тоді як unstructured pruning видаляє окремі ваги, створюючи розріджені матриці. Movement pruning використовує градієнти під час fine-tuning для ідентифікації та видалення ваг, що прагнуть до нуля.

Типовий процес pruning включає навчання повної моделі, ітеративне видалення найменш важливих з'єднань та fine-tuning для відновлення якості. Iterative magnitude pruning поступово збільшує рівень розрідженості (sparsity ratio) протягом кількох циклів pruning-retraining, дозволяючи досягти вищих рівнів розрідженості (50-90%) зі збереженням якості. Проте розріджені моделі вимагають спеціалізованого апаратного забезпечення або бібліотек для ефективного виконання розріджених матричних операцій; на стандартному обладнанні прискорення може бути обмеженим.

Distillation (дистиляція) навчає меншу "студентську" модель імітувати поведінку більшої "вчительської" моделі. На відміну від традиційного навчання на hard labels (конкретних класах), студентська модель навчається відтворювати soft predictions (ймовірнісні розподіли) вчителя, що містять більше інформації про відношення між класами. Функція втрат для distillation поєднує cross-entropy з hard labels та з soft targets від вчителя:

$$L = \alpha L_{CE}(y, \sigma(z_s/T)) + (1-\alpha)L_{KL}(\sigma(z_t/T), \sigma(z_s/T))$$

де z_s та z_t – логіти студента та вчителя, σ – softmax, T – температура (зазвичай 2-5), що згладжує розподіли, α – вага балансування. Дистиляція дозволяє створювати компактні моделі, що зберігають 95-98% якості оригінальної моделі при зменшенні розміру у 2-10 разів.

DistilBERT, наприклад, зменшує BERT з 12 шарів до 6, досягаючи 97% якості оригіналу при подвоєнні швидкості інференсу. TinyBERT використовує багат шарову distillation, де студент навчається імітувати проміжні репрезентації

вчителя на різних рівнях, що дає кращу якість порівняно з distillation лише фінальних предикцій.

Кешування проміжних обчислень є ефективною технікою для прискорення інференсу, особливо для автогресивних моделей. Key-Value (KV) кешування зберігає обчислені ключі та значення у механізмі уваги для попередніх токенів, уникаючи їх перерахунку при генерації кожного наступного токена. Це зменшує обчислювальну складність генерації послідовності довжиною n з $O(n^2)$ до $O(n)$, хоча збільшує потребу у пам'яті на $O(n)$. Multi-Query Attention (MQA) та Grouped-Query Attention (GQA) зменшують розмір KV-кешу через спільне використання ключів та значень між головами уваги, зменшуючи потребу у пам'яті без суттєвого зниження якості.

Prompt caching зберігає обчислення для статичних частин пром프트 (наприклад, системних інструкцій або контексту), що залишаються незмінними між запитами, обчислюючи лише динамічні частини. Це особливо ефективно для сценаріїв з довгими системними промптами або контекстом, зменшуючи латентність та вартість на 30-80%.

У табл 2.3 представлено основні техніки оптимізації великих мовних моделей, зокрема квантизацію, проріджування (pruning), дистиляцію, кешування ключ–значення (KV Caching) та багатоголову/групову увагу (MQA/GQA). Для кожної техніки наведено ступінь зменшення розміру моделі, вплив на швидкодію, якість результатів та складність реалізації, що дозволяє оцінити доцільність їх застосування в системах автоматизованого лінгвістичного аналізу.

Таблиця 2.3 – Методи оптимізації та прискорення великих мовних моделей

Техніка	Зменшення розміру	Прискорення	Вплив на якість	Складність реалізації
8-bit Quantization	4x	2-3x	< 1% деградація	Низька (готові інструменти)
4-bit Quantization	8x	3-5x	1-3% деградація	Середня (потребує calibration)
Pruning (50% sparsity)	2x	1.2-1.5x	< 1% деградація	Висока (специфічне обладнання)
Distillation	2-10x	2-10x	2-5% деградація	Середня (потребує навчання)
KV Caching	Без зміни	2-5x для довгих послідовностей	Без зміни	Низька
MQA/GQA	Зменшення KV-кешу 4-8x	1.5-2x для довгих контекстів	< 1% деградація	Середня (модифікація архітектури)

Таким чином, ефективна реалізація систем автоматизованого лінгвістичного аналізу вимагає комплексного застосування методів попередньої обробки, векторних представлень, промпт-інженерії та при необхідності fine-tuning моделей. Розуміння різноманітних метрик оцінки якості дозволяє об'єктивно порівнювати альтернативні підходи та валідувати систему. Техніки оптимізації та прискорення забезпечують можливість розгортання потужних моделей у ресурсообмежених середовищах, балансуючи якість, швидкість та ефективність використання обчислювальних ресурсів.

2.2 Огляд та порівняльний аналіз існуючих систем лінгвістичного аналізу

2.2.1 Комерційні API-рішення

Сучасний ринок лінгвістичного аналізу характеризується наявністю потужних комерційних рішень, які надають доступ до передових технологій обробки природної мови через програмні інтерфейси. Розглянемо основні комерційні платформи, їх можливості та особливості використання.

2.2.1.1 OpenAI API (GPT-4, GPT-3.5)

OpenAI API надає доступ до найпотужніших мовних моделей серії GPT (Generative Pre-trained Transformer), які демонструють вражаючі результати в широкому спектрі завдань обробки природної мови.

Основні можливості:

1. Генерація тексту високої якості з урахуванням контексту
2. Аналіз тональності та емоційного забарвлення
3. Класифікація тексту за категоріями
4. Витяг ключової інформації та сутностей
5. Резюмування великих текстів

6. Переклад між мовами
7. Відповіді на запитання на основі контексту
8. Аналіз синтаксичних та семантичних структур

Модель GPT-4 підтримує багатомодальний вхід (текст та зображення), що розширює можливості аналізу. GPT-3.5 забезпечує швидшу обробку при дещо нижчій якості результатів порівняно з GPT-4.

Підтримувані мови: OpenAI API підтримує понад 50 мов, включаючи українську. Однак якість обробки варіюється залежно від мови: найкращі результати досягаються для англійської, значно гірші — для мов з обмеженим представленням у тренувальних даних.

Ціноутворення: Модель токенизації визначає вартість використання:

1. GPT-4: \$0.03 за 1000 вхідних токенів, \$0.06 за 1000 вихідних токенів
2. GPT-3.5 Turbo: \$0.0005 за 1000 вхідних токенів, \$0.0015 за 1000 вихідних токенів

Обмеження:

1. Максимальний розмір контексту: 128,000 токенів для GPT-4 Turbo
2. Відсутність гарантій конфіденційності даних
3. Можливість генерації неточної інформації (галюцинації)
4. Залежність від інтернет-з'єднання
5. Обмеження частоти запитів залежно від тарифного плану

Приклади використання: Чат-боти з підтримкою природного діалогу, системи автоматизованого аналізу відгуків клієнтів, інструменти для створення

контенту, асистенти програмування, системи питань-відповідей для корпоративних баз знань.

2.2.1.2 Google Cloud Natural Language API

Google Cloud Natural Language API — це набір інструментів машинного навчання для аналізу тексту, що базується на багаторічному досвіді Google у обробці природної мови.

Основні можливості:

1. Аналіз сутностей (Entity Analysis) з розпізнаванням типів та важливості
2. Аналіз тональності з визначенням градації від негативного до позитивного
3. Аналіз синтаксису з визначенням частин мови та залежностей
4. Класифікація контенту за понад 700 категоріями
5. Модерація контенту для виявлення небажаного матеріалу

API інтегрується з іншими сервісами Google Cloud, що дозволяє створювати комплексні рішення для обробки даних.

Підтримувані мови: Понад 30 мов, включаючи українську. Однак не всі функції доступні для всіх мов: повний набір функцій підтримується лише для дев'яти мов, включаючи англійську, іспанську та китайську .

Ціноутворення:

1. Аналіз сутностей: \$1.00 за 1000 одиниць тексту
2. Аналіз тональності: \$1.00 за 1000 одиниць
3. Синтаксичний аналіз: \$0.50 за 1000 одиниць

4. Класифікація контенту: \$2.00 за 1000 одиниць Перші 5000 одиниць щомісяця безкоштовні.

Обмеження:

1. Максимальний розмір документа: 1 МБ
2. Обмеження швидкості: 600 запитів на хвилину
3. Обмежена функціональність для мов, відмінних від англійської
4. Вимагає реєстрації та налаштування проекту в Google Cloud

Приклади використання: Аналіз відгуків у соціальних мережах, автоматична категоризація новин, системи пошуку корпоративної інформації, моніторинг бренду в медіа.

2.2.1.3 Amazon Comprehend

Amazon Comprehend — це сервіс обробки природної мови від AWS, який використовує машинне навчання для виявлення інсайтів у текстових даних .

Основні можливості:

1. Виявлення ключових фраз та сутностей
2. Аналіз тональності з підтримкою змішаних емоцій
3. Визначення мови документа
4. Тематичне моделювання для великих колекцій документів
5. Розпізнавання медичної термінології (Amazon Comprehend Medical)
6. Кастомізація моделей для специфічних доменів
7. Аналіз PII (Personally Identifiable Information)

Сервіс інтегрується з екосистемою AWS, підтримує пакетну та потокову обробку даних.

Підтримувані мови: Понад 100 мов для визначення мови документа, але повний набір функцій підтримується лише для 12 основних мов, українська має обмежену підтримку .

Ціноутворення:

1. Аналіз сутностей, тональності, ключових фраз: \$0.0001 за одиницю (100 символів)
2. Тематичне моделювання: \$1.00 за 1000 одиниць
3. Кастомні моделі: від \$3.00 за годину тренування Перші 50,000 одиниць на місяць безкоштовні протягом 12 місяців.

Обмеження:

1. Максимальний розмір документа: 5000 байт для синхронних запитів
2. Обмеження швидкості залежать від регіону
3. Кастомні моделі вимагають значного обсягу тренувальних даних
4. Складність налаштування для користувачів без досвіду AWS

Приклади використання: Аналіз документації служби підтримки, системи аналізу медичних записів, моніторинг соціальних медіа, автоматична класифікація документів у великих корпораціях.

2.2.1.4 Microsoft Azure Text Analytics

Azure Text Analytics — частина Azure Cognitive Services, що надає передові можливості обробки тексту з акцентом на бізнес-застосування.

Основні можливості:

1. Розпізнавання іменованих сутностей з підтримкою понад 20 типів
2. Аналіз тональності з визначенням впевненості
3. Витяг ключових фраз
4. Визначення мови
5. Opinion Mining (аналіз думок про конкретні аспекти)
6. Розпізнавання медичних термінів (Text Analytics for Health)
7. Підтримка питань-відповідей на основі документів

Підтримувані мови: Близько 120 мов для визначення мови, понад 30 мов для аналізу тональності та сутностей. Українська мова підтримується для базових операцій.

Ціноутворення:

1. Стандартний рівень: від \$0.25 за 1000 текстових записів
2. Рівні з різними можливостями (S0-S4) від \$2 до \$1000 за місяць
3. Перші 5000 транзакцій безкоштовні щомісяця

Обмеження:

1. Максимальний розмір документа: 5120 символів
2. Обмеження швидкості: залежить від обраного рівня (від 1000 до 10000 запитів на хвилину)
3. Деякі функції доступні лише в певних регіонах
4. Вимагає активного підписання Azure

Приклади використання: Автоматизація обробки звернень клієнтів, аналіз настроїв у корпоративних опитуваннях, системи пошуку знань, аналіз конкурентної інформації.

2.2.1.5 Anthropic Claude API

Claude API від Anthropic — відносно новий гравець на ринку, який позиціонує себе як більш безпечна та керована альтернатива іншим великим мовним моделям.

Основні можливості:

1. Розширений контекстний аналіз (до 200,000 токенів)
2. Генерація тексту з дотриманням етичних норм
3. Аналіз та резюмування довгих документів
4. Відповіді на складні питання з міркуванням
5. Класифікація та категоризація тексту
6. Витяг структурованої інформації
7. Кодогенерація та аналіз коду

Claude відзначається більшою точністю в розумінні інструкцій та меншою схильністю до генерації небажаного контенту.

Підтримувані мови: Підтримка багатьох мов, включаючи українську, хоча якість варіюється. Найкраща продуктивність досягається для англійської мови.

Ціноутворення:

1. Claude 3 Opus: \$15 за 1М вхідних токенів, \$75 за 1М вихідних токенів
2. Claude 3 Sonnet: \$3 за 1М вхідних токенів, \$15 за 1М вихідних токенів

3. Claude 3 Haiku: \$0.25 за 1М вхідних токенів, \$1.25 за 1М вихідних токенів

Обмеження:

1. Обмеження частоти запитів залежать від тарифного плану
2. Відсутність можливості fine-tuning
3. Менша екосистема інтеграцій порівняно з конкурентами
4. Відносно новий продукт з меншою кількістю прикладів використання

Приклади використання: Аналіз юридичних документів, наукові дослідження текстів, системи технічної підтримки, аналіз контрактів, резюмування довгих звітів.

2.2.1.6 Cohere API

Cohere — платформа NLP, орієнтована на корпоративних клієнтів, що пропонує потужні мовні моделі з акцентом на кастомізацію .

Основні можливості:

1. Генерація тексту з можливістю fine-tuning
2. Семантичний пошук та ранжування
3. Класифікація тексту з кастомними категоріями
4. Створення embeddings для аналізу схожості
5. Токенізація та аналіз тексту
6. Summarization та перефразування
7. Multilingual embeddings

Підтримувані мови: Понад 100 мов, включаючи українську, через багатомовні моделі. Cohere пропонує спеціалізовані моделі для різних мовних груп.

Ціноутворення:

1. Generation: від \$0.40 за 1M токенів
 2. Embedding: від \$0.10 за 1M токенів
 3. Classification: від \$0.20 за 1000 класифікацій
 4. Rerank: від \$1.00 за 1000 пошукових запитів
- Доступний безкоштовний рівень для розробки та тестування.

Обмеження:

1. Розмір контексту обмежений залежно від моделі
2. Fine-tuning вимагає значних обсягів даних
3. Менша спільнота розробників порівняно з OpenAI
4. Деякі функції доступні лише на вищих тарифних планах

Приклади використання: Корпоративні системи пошуку знань, кастомізовані чат-боти, аналіз відгуків клієнтів, системи рекомендацій контенту.

2.2.2 Відкриті бібліотеки та фреймворки

Відкриті бібліотеки та фреймворки надають гнучкі та економічно ефективні альтернативи комерційним рішенням, дозволяючи повний контроль над процесом обробки та зберігання даних.

2.2.2.1 Hugging Face Transformers

Hugging Face Transformers — найпопулярніша бібліотека для роботи з трансформерними моделями, що стала стандартом де-факто в спільноті NLP.

Функціональність:

1. Єдиний інтерфейс для тисяч попередньо навчених моделей
2. Підтримка всіх основних архітектур: BERT, GPT, T5, RoBERTa, XLNet та інші
3. Pipelines для швидкого вирішення типових завдань (класифікація, NER, питання-відповіді)
4. Fine-tuning моделей на кастомних даних
5. Інтеграція з PyTorch, TensorFlow та JAX
6. Підтримка багатомовних та крос-лінгвальних моделей
7. Оптимізація моделей для продакшену (quantization, ONNX export)

Підтримувані моделі: Понад 200,000 моделей у Hugging Face Hub, включаючи:

1. Багатомовні моделі: mBERT, XLM-RoBERTa, mT5
2. Моделі для української мови: uk-bert, xlm-roberta-base-uk
3. Спеціалізовані моделі для різних завдань: sentiment analysis, NER, summarization

Переваги:

1. Найбільша колекція моделей та датасетів
2. Активна спільнота та регулярні оновлення
3. Відмінна документація та туторіали
4. Підтримка сучасних технологій: quantization, distillation, pruning
5. Безкоштовне використання та відкритий вихідний код

6. Можливість розгортання на власній інфраструктурі

Недоліки:

1. Вимагає глибокого розуміння архітектур моделей
2. Високі вимоги до обчислювальних ресурсів для великих моделей
3. Складність оптимізації для продакшн-середовищ
4. Необхідність самостійного управління версіями та інфраструктурою

Популярність: GitHub: понад 130,000 зірок, понад 2 млрд завантажень через рір, активна спільнота з тисячами контриб'юторів.

2.2.2.2 spaCy

spaCy — індустріальна бібліотека для обробки природної мови, що орієнтована на швидкість та ефективність у продакшн-середовищах .

Функціональність:

1. Токенізація, лематизація, POS-tagging
2. Розпізнавання іменованих сутностей (NER)
3. Синтаксичний аналіз та dependency parsing
4. Word vectors та similarity
5. Text classification
6. Rule-based matching
7. Підтримка кастомних компонентів у pipeline

Підтримувані моделі:

1. 75+ попередньо навчених моделей для 23 мов

2. Багаторівневі моделі: малі (sm), середні (md), великі (lg), трансформерні (trf)

3. Підтримка української мови через модель uk_core_news_*

4. Можливість навчання власних моделей

Переваги:

1. Висока швидкість обробки (оптимізація на Cython)

2. Production-ready архітектура

3. Зручний API та чітка документація

4. Інтеграція з іншими бібліотеками (Transformers, PyTorch)

5. Вбудовані візуалізації (displaCy)

6. Активний розвиток та підтримка

Недоліки:

1. Менша гнучкість порівняно з NLTK

2. Обмежена підтримка деяких мов

3. Трансформерні моделі вимагають значних ресурсів

4. Модель для української не досягає точності сучасних трансформерних моделей

Популярність: GitHub: понад 29,000 зірок, понад 100 млн завантажень, широке використання у корпоративному секторі.

2.2.2.3 NLTK (Natural Language Toolkit)

NLTK — одна з перших та найбільш повних бібліотек для обробки природної мови, що широко використовується в освітніх цілях .

Функціональність:

1. Понад 50 корпусів текстів та лексичних ресурсів
2. Класифікація тексту, токенизація, stemming, tagging
3. Parsing та семантичний аналіз
4. Обгортки для інших інструментів NLP
5. Модулі для статистичного аналізу
6. Sentiment analysis через VADER

Підтримувані моделі:

1. Класичні статистичні моделі
2. Pretrained taggers та chunkers
3. WordNet та інші лексичні бази
4. Можливість інтеграції з сучасними моделями

Переваги:

1. Вичерпна бібліотека алгоритмів NLP
2. Чудові навчальні матеріали та документація
3. Підтримка широкого спектру задач
4. Легкий у вивченні для початківців
5. Безкоштовний та відкритий код

Недоліки:

1. Повільніший за сучасні альтернативи (spaCy)
2. Менша орієнтація на продакшн

3. Застарілі підходи для деяких задач
4. Обмежена підтримка нейронних мереж
5. Важка залежність від зовнішніх даних

Популярність: GitHub: понад 13,000 зірок, мільйони завантажень, стандарт для навчання NLP у університетах.

2.2.2.4 Gensim

Gensim — бібліотека для тематичного моделювання та аналізу семантичної схожості документів .

Функціональність:

1. Word2Vec, Doc2Vec, FastText embeddings
2. Тематичне моделювання: LDA, LSI, HDP
3. Аналіз схожості документів
4. Корпусна обробка великих текстових колекцій
5. Streaming та online навчання
6. Phrase detection (bigrams, trigrams)

Підтримувані моделі:

1. Класичні моделі word embeddings
2. Тематичні моделі різної складності
3. Підтримка pretrained моделей (Word2Vec from Google News)
4. Власні моделі на кастомних корпусах

Переваги:

1. Ефективна робота з великими корпусами
2. Memory-efficient streaming обробка
3. Відмінна підтримка тематичного моделювання
4. Простота використання для дослідників
5. Добре документовано

Недоліки:

1. Обмежений функціонал поза embeddings та topic modeling
2. Не підтримує сучасні трансформерні моделі
3. Відсутність NER та синтаксичного аналізу
4. Менша спільнота порівняно з іншими бібліотеками

Популярність: GitHub: понад 15,000 зірок, широке використання в академічних дослідженнях, регулярні оновлення.

2.2.2.5 Stanford CoreNLP

Stanford CoreNLP — набір інструментів для лінгвістичного аналізу від Stanford NLP Group .

Функціональність:

1. Токенізація, POS-tagging, NER
2. Parsing: constituency та dependency
3. Coreference resolution
4. Sentiment analysis
5. Relation extraction

6. Open Information Extraction

7. Підтримка 8 мов

Підтримувані моделі:

1. Статистичні моделі для різних мов
2. Neural network моделі для окремих завдань
3. Pretrained моделі для англійської, китайської, іспанської та інших
4. Можливість навчання власних моделей

Переваги:

1. Високоякісний лінгвістичний аналіз
2. Академічна точність
3. Комплексний підхід до аналізу
4. Добре протестовані алгоритми
5. Підтримка від провідного університету

Недоліки:

1. Написаний на Java (може бути незручно для Python-розробників)
2. Повільніший за сучасні нейронні моделі
3. Велике споживання пам'яті
4. Складність інтеграції в деякі середовища
5. Відсутність підтримки української мови

Популярність: GitHub: понад 9,000 зірок, широке використання в академічних дослідженнях, регулярні публікації в топових конференціях.

2.2.2.6 Stanza

Stanza — наступник Stanford CoreNLP, побудований на нейронних мережах та оптимізований для Python .

Функціональність:

1. Токенізація, лематизація, POS/morphological tagging
2. Dependency parsing
3. Named Entity Recognition
4. Sentiment analysis
5. Підтримка 70+ мов, включаючи українську

Підтримувані моделі:

1. Нейронні моделі для всіх підтримуваних мов
2. Моделі з Universal Dependencies treebanks
3. Багатомовні та monolingual моделі
4. Регулярні оновлення з новими мовами

Переваги:

1. Сучасні нейронні архітектури
2. Краща точність порівняно з CoreNLP
3. Python-first підхід
4. Підтримка багатьох мов, включаючи українську
5. Відносно швидка обробка

Недоліки:

1. Менша спільнота порівняно з spaCy
2. Обмежена функціональність поза базовими задачами
3. Якість моделей варіюється залежно від мови
4. Висока залежність від якості тренувальних даних

Популярність: GitHub: понад 7,000 зірок, зростаюча популярність в академічному середовищі, підтримка Stanford NLP Group.

2.2.3 Спеціалізовані системи

2.2.3.1 Системи для конкретних мов

Системи для української мови:

lang-uk — проект відкритого коду, що фокусується на інструментах для обробки української мови. Включає токенізатор, тегувальник частин мови та інші базові компоненти. Використовує правилів підходи та статистичні моделі, адаптовані до особливостей української морфології.

UA-GEC (Ukrainian Grammar Error Correction) — система корекції граматичних помилок для української мови, розроблена на базі трансформерних моделей. Демонструє високу точність у виявленні та виправленні типових помилок українських текстів.

ukr-models (від Hugging Face) — набір моделей BERT та RoBERTa, дотренованих на українських корпусах. Забезпечує кращу якість для завдань класифікації, NER та питань-відповідей українською мовою порівняно з багатомовними моделями.

Обмеження систем для української мови:

1. Менші обсяги тренувальних даних порівняно з англійською
2. Обмежена підтримка діалектів та розмовної мови

3. Відсутність комплексних комерційних рішень
4. Нижча точність у спеціалізованих доменах

2.2.3.2 Галузеві рішення

Медицина:

Amazon Comprehend Medical — спеціалізоване рішення для витягу медичної інформації з неструктурованих текстів. Розпізнає медичні сутності (діагнози, ліки, процедури, анатомію), зв'язки між ними та PHI (Protected Health Information).

BioBERT та *PubMedBERT* — трансформерні моделі, дотреновані на медичній літературі. Демонструють високу точність у завданнях NER, relation extraction та класифікації медичних текстів.

Юриспруденція:

LegalBERT — BERT модель, натренована на юридичних документах. Оптимізована для аналізу контрактів, судових рішень та юридичних меморандумів.

Системи аналізу контрактів (Kira Systems, Luminance) — комерційні рішення для автоматизованого аналізу юридичних документів, витягу ключових положень та виявлення ризиків.

Фінанси:

FinBERT — модель для аналізу фінансових текстів, оптимізована для визначення тональності новин та звітів компаній.

BloombergGPT — велика мовна модель, спеціально навчена на фінансових даних для завдань фінансового аналізу, генерації звітів та відповідей на питання.

2.2.5 Виявлення переваг та недоліків

2.2.5.1 Аналіз проблемних місць існуючих рішень

Комерційні API:

Проблеми конфіденційності та безпеки даних. Більшість комерційних рішень передають дані на віддалені сервери, що створює ризики витоку конфіденційної інформації. Для медичних, юридичних та фінансових організацій це може суперечити регуляторним вимогам (GDPR, HIPAA, локальним законам про захист даних).

Обмежена підтримка української мови. Усі розглянуті комерційні системи демонструють значно нижчу якість обробки української мови порівняно з англійською. Точність розпізнавання сутностей, аналізу тональності та інших завдань падає на 20-40% для української мови.

Висока вартість при масштабуванні. При обробці великих обсягів текстів вартість використання комерційних API може досягати тисяч доларів на місяць. Наприклад, обробка 100 млн токенів через GPT-4 коштуватиме близько \$3000-6000.

Залежність від інтернет-з'єднання. Усі API-рішення вимагають стабільного інтернет-підключення, що робить неможливим їх використання в офлайн-режимі або в середовищах з обмеженою connectivity.

Відсутність повного контролю. Користувачі не можуть модифікувати базові алгоритми, що обмежує можливості адаптації до специфічних потреб. Зміни в API (deprecated функції, оновлення моделей) можуть порушити роботу існуючих систем.

Відкриті бібліотеки:

Складність розгортання та підтримки. Великі трансформерні моделі вимагають значних обчислювальних ресурсів. Для ефективної роботи GPT-подібних моделей необхідні потужні GPU, що створює високий поріг входу для малих команд та стартапів.

Відсутність комплексних рішень для української мови. Хоча деякі бібліотеки підтримують українську, якість моделей часто недостатня для продакшн-використання. Відсутні високоякісні моделі для спеціалізованих завдань (медицина, юриспруденція українською мовою).

Необхідність глибокої експертизи. Ефективне використання сучасних NLP-бібліотек вимагає глибокого розуміння machine learning, архітектур моделей, оптимізації та інженерії даних. Це створює бар'єр для багатьох організацій.

Проблеми продакшн-готовності. Багато академічних моделей та бібліотек не оптимізовані для продакшн-середовищ. Проблеми з латентністю, споживанням пам'яті, стабільністю під навантаженням вимагають додаткової роботи.

Спеціалізовані системи:

Вузька спеціалізація. Системи, оптимізовані для конкретних доменів, погано працюють з текстами інших тематик. Необхідність підтримки декількох систем для різних задач ускладнює архітектуру.

Обмежена доступність для української мови. Практично відсутні спеціалізовані системи для української мови в таких доменах, як медицина, юриспруденція, фінанси.

Висока вартість розробки. Створення спеціалізованої системи вимагає значних інвестицій у збір та анотацію даних, навчання моделей, тестування.

2.2.5.2 Можливості для вдосконалення

Гібридні архітектури. Комбінування переваг різних підходів може забезпечити кращі результати. Наприклад, використання легких статистичних моделей для попередньої обробки та фільтрації з подальшим застосуванням важких трансформерних моделей лише до релевантних фрагментів тексту може значно підвищити ефективність.

Оптимізація для української мови. Створення спеціалізованих моделей, дотренованих на великих українських корпусах, може суттєво підвищити якість обробки. Використання transfer learning від багатомовних моделей з подальшим fine-tuning на українських даних показує перспективні результати.

Модульна архітектура. Розробка системи з взаємозамінними компонентами дозволить легко адаптувати рішення до різних сценаріїв використання. Користувачі зможуть вибирати між швидкістю та точністю, між хмарними та локальними рішеннями.

Edge computing та оптимізація. Використання технік квантизації, дистиляції моделей та прунінгу може дозволити запуск потужних моделей на обмежених ресурсах. Це відкриває можливості для edge deployment та офлайн-використання.

Активне навчання та human-in-the-loop. Інтеграція механізмів активного навчання, коли система сама ідентифікує найбільш складні або невизначені випадки для людської анотації, може значно прискорити процес покращення якості моделей.

Пояснюваність та інтерпретованість. Додавання механізмів пояснення рішень моделей (attention visualization, feature importance) підвищить довіру користувачів та полегшить налагодження проблем.

2.2.5.3 Обґрунтування необхідності розробки нової системи

Аналіз існуючих рішень виявляє значну прогалину на ринку систем лінгвістичного аналізу, особливо для української мови. Жодне з розглянутих рішень не задовольняє повністю наступні критичні вимоги одночасно:

1. Висока якість обробки української мови. Існуючі комерційні API демонструють точність 60-70% для української мови порівняно з 90-95% для англійської. Це неприйнятно для продакшн-систем.

2. Конфіденційність даних. Організації, що працюють з чутливими даними, не можуть використовувати хмарні API. Відкриті рішення вимагають складного розгортання та підтримки.

3. Економічна ефективність при масштабуванні. Комерційні API стають надто дорогими при великих обсягах, а відкриті рішення вимагають значних інвестицій в інфраструктуру.

4. Баланс між точністю та швидкістю. Більшість систем оптимізовані або для максимальної точності (повільні, ресурсоємні), або для швидкості (нижча якість).

5. Простота інтеграції та використання. Потужні відкриті бібліотеки вимагають глибокої експертизи, а прості рішення не забезпечують достатньої якості.

Нова система повинна забезпечити:

1. Високу точність для української мови через спеціалізовані моделі
2. Можливість локального розгортання для забезпечення конфіденційності
3. Модульну архітектуру для балансу між різними вимогами
4. Простий API для швидкої інтеграції
5. Економічну ефективність через оптимізацію та можливість використання різних backend'ів

Особлива увага має бути приділена створенню ефективних моделей для української мови через збір та курацію високоякісних українськомовних корпусів, адаптацію сучасних архітектур (BERT, GPT, T5) до морфологічних особливостей української мови, розробку спеціалізованих токенизаторів та embeddings.

2.2.4 Висновки до підрозділу

Проведений огляд та порівняльний аналіз існуючих систем лінгвістичного аналізу дозволяє зробити наступні висновки:

1. Комерційні API-рішення (OpenAI, Google, Amazon, Microsoft, Anthropic, Cohere) пропонують потужні можливості та високу точність для англійської мови, але мають суттєві обмеження для української: недостатня якість обробки, висока вартість при масштабуванні, проблеми конфіденційності даних, залежність від інтернет-з'єднання.

2. Відкриті бібліотеки (Hugging Face Transformers, spaCy, NLTK, Gensim, Stanford CoreNLP, Stanza) надають гнучкість та контроль, але вимагають значних технічних компетенцій, обчислювальних ресурсів та зусиль з оптимізації для продакшн-середовищ. Підтримка української мови в цих бібліотеках обмежена або базова.

3. Спеціалізовані системи для конкретних доменів демонструють високу точність у своїх нішах, але практично відсутні для української мови, особливо в критичних галузях як медицина та юриспруденція.

4. Ключові проблеми існуючих рішень включають: низьку якість обробки української мови (на 20-40% нижче порівняно з англійською), відсутність балансу між точністю, швидкістю та вартістю, складність розгортання та підтримки потужних моделей, відсутність готових рішень для офлайн-режиму.

5. Обґрунтована необхідність розробки нової системи, яка б поєднувала переваги існуючих підходів: високу точність для української мови через спеціалізовані моделі, модульну архітектуру для гнучкої адаптації, можливість локального розгортання, економічну ефективність, простоту інтеграції.

Результати аналізу формують основу для розробки архітектури системи, яка буде представлена в наступних підрозділах.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ АВТОМАТИЗОВАНОГО ЛІНГВІСТИЧНОГО АНАЛІЗУ

3.1 Формування вимог до системи

Формування вимог до програмної системи є критично важливим етапом розробки, оскільки саме на цьому етапі визначаються функціональні можливості, технічні характеристики та обмеження майбутнього продукту. Чітко сформульовані вимоги дозволяють уникнути непорозумінь між замовником і розробниками, забезпечують основу для проєктування архітектури системи та є критерієм для тестування готового рішення. У контексті розробки системи аналізу тексту на основі великих мовних моделей особливого значення набувають як функціональні вимоги, що визначають бізнес-логіку та можливості системи, так і нефункціональні вимоги, які забезпечують якісні характеристики роботи програмного забезпечення.

3.1.1 Функціональні вимоги до системи

Функціональні вимоги визначають конкретні функції та можливості, які має надавати система для виконання покладених на неї завдань. Для системи аналізу тексту на основі великих мовних моделей сформульовано наступні ключові функціональні вимоги.

3.1.1.1 Єдиний REST API endpoint для аналізу тексту

Система повинна надавати уніфікований REST API endpoint, що забезпечує доступ до всіх функцій аналізу тексту через єдину точку входу. Цей підхід значно спрощує інтеграцію системи з іншими додатками та сервісами, оскільки клієнтським додаткам достатньо взаємодіяти лише з одним endpoint'ом замість необхідності керування множиною окремих точок доступу.

Endpoint повинен приймати HTTP POST запити з тілом у форматі JSON, що містить текст для аналізу та необхідні параметри конфігурації. Використання

POST методу обумовлене необхідністю передачі потенційно великих обсягів текстових даних, які неможливо ефективно передати через параметри URL у GET запитах. Структура запиту має включати наступні обов'язкові поля: текст для аналізу, ідентифікатор або назву моделі для використання, тип провайдера (OpenRouter API або локальна модель).

Endpoint має підтримувати синхронний режим роботи, повертаючи результати аналізу безпосередньо у відповіді на запит. Для забезпечення надійності система повинна коректно обробляти різноманітні помилкові ситуації: відсутність обов'язкових параметрів, некоректний формат вхідних даних, недоступність обраної моделі, перевищення максимально допустимого розміру тексту. Кожна помилка має супроводжуватися зрозумілим повідомленням та відповідним HTTP статус-кодом.

3.1.1.2 Інтеграція з OpenRouter API

Система повинна забезпечувати повноцінну інтеграцію з сервісом OpenRouter API, який надає уніфікований доступ до різноманітних великих мовних моделей від різних провайдерів. OpenRouter виступає агрегатором, що дозволяє працювати з моделями від OpenAI, Anthropic, Google, Meta та інших компаній через єдиний інтерфейс.

Інтеграція має забезпечувати наступні можливості: автентифікацію за допомогою API ключа, що зберігається у безпечному сховищі конфігурації; підтримку вибору конкретної моделі з переліку доступних через OpenRouter; передачу системних та користувацьких промптів у форматі, що відповідає специфікації OpenRouter API; отримання та обробку відповідей від моделей з урахуванням особливостей формату кожного провайдера.

Система повинна коректно обробляти специфічні помилки OpenRouter API, включаючи: недостатній баланс на рахунку, недійсний API ключ, тимчасову недоступність обраної моделі, перевищення ліміту швидкості запитів. Для

підвищення надійності має бути реалізований механізм повторних спроб з експоненційною затримкою у випадку тимчасових помилок мережі або перевантаження сервісу.

3.1.1.3 Підтримка локальних мовних моделей

Окрім хмарних сервісів, система повинна підтримувати роботу з локально розгорнутими мовними моделями. Ця функціональність є критично важливою для випадків, коли необхідно забезпечити конфіденційність даних, зменшити експлуатаційні витрати або забезпечити роботу системи в середовищах без постійного доступу до Інтернету.

Система має підтримувати інтеграцію з Ollama – популярним інструментом для локального запуску великих мовних моделей. Ollama забезпечує зручний інтерфейс для завантаження, управління та використання різноманітних open-source моделей, таких як Llama, Mistral, Phi та інших. Підтримка Ollama дозволяє користувачам самостійно вибирати та налаштовувати моделі відповідно до своїх специфічних потреб.

Функціональність роботи з локальними моделями має включати: автоматичне виявлення запущеного Ollama сервера за стандартною адресою; можливість конфігурації альтернативної адреси сервера; підтримку вибору конкретної моделі з переліку встановлених у Ollama; передачу промптів у форматі, сумісному з Ollama API; обробку специфічних параметрів генерації, таких як температура, максимальна довжина відповіді, top-p sampling.

Система повинна забезпечувати єдиний інтерфейс взаємодії незалежно від типу використовуваної моделі (хмарна чи локальна), приховуючи деталі реалізації від кінцевого користувача та забезпечуючи прозору заміну провайдера.

3.1.1.4 Комплексний аналіз текстових даних

Ключовою функціональною вимогою є здатність системи виконувати комплексний багатоаспектний аналіз вхідного тексту. Система повинна надавати

результати чотирьох основних типів аналізу за один запит, що значно підвищує ефективність та зручність використання.

Розпізнавання іменованих сутностей (Named Entity Recognition, NER) передбачає ідентифікацію та класифікацію ключових елементів тексту на категорії, такі як імена людей, назви організацій, географічні локації, дати, грошові суми, відсоткові показники. Система має не лише виявляти ці сутності, але й надавати інформацію про їх позицію у тексті та рівень впевненості в правильності класифікації.

Аналіз тональності (Sentiment Analysis) має визначати емоційне забарвлення тексту з класифікацією на позитивний, негативний або нейтральний тон. Крім загального тону документа, система може надавати градації інтенсивності емоцій та виявляти зміни тональності в межах різних частин тексту. Це особливо важливо для аналізу довгих документів або текстів зі складною емоційною структурою.

Вилучення ключових слів (Keyword Extraction) передбачає автоматичну ідентифікацію найбільш значущих термінів та фраз, що відображають основні теми та концепції тексту. Система має ранжувати ключові слова за релевантністю та надавати метрики важливості кожного терміну. Алгоритм має враховувати не лише частоту вживання слів, але й їх семантичну значущість у контексті документа.

Автоматичне реферування (Text Summarization) має генерувати стислий виклад основного змісту тексту, зберігаючи ключову інформацію та логічну структуру. Система повинна підтримувати генерацію рефератів різної довжини відповідно до потреб користувача. Реферат має бути зв'язним, граматично правильним та адекватно відображати зміст вихідного документа.

Всі типи аналізу мають виконуватися одночасно в рамках одного виклику моделі для оптимізації використання ресурсів та мінімізації часу обробки.

Результати кожного типу аналізу повинні бути структурованими, легко інтерпретованими та готовими до подальшої обробки або візуалізації.

3.1.1.5 Структурована JSON відповідь

Система повинна повертати результати аналізу у структурованому форматі JSON, що забезпечує легкість парсингу, інтеграції та подальшої обробки даних клієнтськими додатками. Використання JSON як формату обміну даними обумовлене його *universal support across programming languages, human readability* та *efficiency*.

Структура JSON відповіді має бути чітко визначеною та послідовною незалежно від типу використовуваної моделі або специфіки вхідного тексту. Відповідь повинна містити окремі секції для кожного типу аналізу: `named_entities` для результатів NER, `sentiment` для аналізу тональності, `keywords` для ключових слів, `summary` для реферату.

Кожна секція має включати не лише результати аналізу, але й метадані, такі як рівень впевненості, версія моделі, час обробки. Для розпізнаних іменованих сутностей необхідно надавати інформацію про тип сутності, текстове представлення, позицію в оригінальному тексті. Для аналізу тональності мають бути доступні як категоріальна оцінка (*positive/negative/neutral*), так і числовий показник інтенсивності емоцій.

Структура відповіді має передбачати можливість розширення для додавання нових типів аналізу в майбутньому без порушення зворотної сумісності. Всі поля повинні мати зрозумілі назви та документовану схему даних. У випадку помилок під час обробки окремих типів аналізу, відповідь має містити інформацію про успішно виконані операції та деталі помилок.

3.2 Проектування архітектури системи

Проектування архітектури програмної системи є фундаментальним етапом розробки, що визначає структурну організацію компонентів, їх взаємодію та

технологічну основу реалізації. Правильно спроектована архітектура забезпечує масштабованість, підтримуваність та можливість подальшого розвитку системи. У контексті розробки системи аналізу тексту на основі великих мовних моделей архітектурні рішення мають балансувати між простотою реалізації, продуктивністю та гнучкістю використання. Цей підрозділ присвячений детальному розгляду обраного архітектурного підходу, компонентної структури системи та технологічних рішень, що забезпечують її функціонування.

3.2.1 Вибір архітектурного стилю

При проектуванні архітектури програмної системи розробники стикаються з вибором між різними архітектурними стилями, кожен з яких має свої переваги та недоліки. Основними альтернативами для веб-орієнтованих систем є монолітна архітектура та мікросервісна архітектура. Для системи аналізу тексту обрано монолітний архітектурний стиль, що обумовлено рядом об'єктивних факторів.

3.2.1.1 Обґрунтування вибору монолітної архітектури

Монолітна архітектура передбачає організацію всіх компонентів системи в межах єдиного розгортаного артефакту. Всі модулі, сервіси та компоненти функціонують в одному процесі та використовують спільний адресний простір пам'яті. Такий підхід контрастує з мікросервісною архітектурою, де функціональність розподілена між множиною незалежних сервісів, що взаємодіють через мережеві протоколи.

Вибір монолітної архітектури для системи аналізу тексту обумовлений наступними факторами. По-перше, система має чітко визначену та обмежену функціональність, що не вимагає складної декомпозиції на окремі сервіси. Основна бізнес-логіка зосереджена навколо прийому текстових даних, взаємодії з мовними моделями та формування структурованих результатів аналізу. Ця функціональність природним чином групується в єдиний домен без необхідності виділення окремих bounded contexts.

По-друге, монолітна архітектура значно спрощує процес розробки та підтримки системи. Розробники працюють з єдиною кодовою базою, що полегшує навігацію, рефакторинг та розуміння системи в цілому. Відсутність мережових викликів між компонентами усуває цілий клас потенційних проблем, пов'язаних з латентністю мережі, часткових відмов, складної діагностики розподілених систем.

По-третє, розгортання монолітного додатку є значно простішим порівняно з оркестрацією множини мікросервісів. Система розгортається як єдиний JAR або WAR файл, що не вимагає складної інфраструктури управління контейнерами, service discovery, API gateway. Це особливо важливо для середовищ з обмеженими ресурсами або для локального розгортання на робочих станціях розробників.

По-четверте, монолітна архітектура забезпечує кращу продуктивність для системи з щільною взаємодією компонентів. Виклики між компонентами здійснюються як прямі виклики методів в межах одного процесу, що на кілька порядків швидше за мережові виклики. Для системи аналізу тексту, де потрібна тісна координація між компонентами обробки запиту, формування промπτу, взаємодії з моделлю та побудови відповіді, це дає відчутні переваги в швидкості обробки.

По-п'яте, простота тестування монолітних систем значно вища. Інтеграційні тести можуть виконуватися в межах одного процесу без необхідності піднімати складне тестове оточення з множиною сервісів. Unit тести для окремих компонентів пишуться з використанням простих mock об'єктів замість складних заглушок для HTTP викликів.

Важливо зазначити, що вибір монолітної архітектури не є обмеженням для майбутньої еволюції системи. За потреби, система може бути поступово трансформована до мікросервісної архітектури шляхом виділення окремих компонентів в самостійні сервіси. Добре структурований монолітний код з чіткими межами між компонентами є хорошою відправною точкою для такої трансформації.

3.2.1.2 Принципи організації монолітної архітектури

При реалізації монолітної архітектури важливо дотримуватися принципів, що забезпечують високу якість коду та можливість подальшого розвитку системи. Система аналізу тексту організована за принципом *layered architecture* (шарової архітектури), де компоненти згруповані у логічні шари відповідно до їх відповідальності.

Верхній шар представлений контролерами REST API, що відповідають за прийом HTTP запитів, валідацію вхідних даних та формування HTTP відповідей. Цей шар є точкою входу в систему та забезпечує інтерфейс взаємодії з зовнішніми клієнтами.

Середній шар містить бізнес-логіку системи, реалізовану через набір сервісів. Сервіси інкапсулюють основну функціональність системи: взаємодію з мовними моделями, оркестрацію різних типів аналізу, обробку результатів. Цей шар є незалежним від деталей зовнішнього інтерфейсу та може бути легко протестований ізолювано.

Нижній шар представлений допоміжними компонентами: HTTP клієнтами для взаємодії з зовнішніми API, утилітами для роботи з JSON, конфігураційними компонентами. Цей шар забезпечує інфраструктурні можливості для функціонування вищих шарів.

Взаємодія між шарами організована за принципом *dependency inversion*: вищі шари залежать від абстракцій (інтерфейсів), а не від конкретних реалізацій. Це забезпечує слабке зв'язування компонентів та можливість заміни реалізацій без впливу на інші частини системи.

3.2.2 Компонентна структура системи

Система аналізу тексту складається з чотирьох ключових компонентів, кожен з яких має чітко визначену відповідальність та інтерфейс взаємодії. Розглянемо детально кожен компонент та його роль в загальній архітектурі.

3.2.2.1 REST Controller

REST Controller є компонентом презентаційного шару, що забезпечує HTTP інтерфейс для взаємодії з системою. Основна відповідальність цього компонента полягає в обробці вхідних HTTP запитів, валідації параметрів та делегуванні виконання бізнес-логіки відповідним сервісам.

Архітектурне рішення про використання єдиного endpoint для всіх операцій аналізу тексту обумовлено прагненням до спрощення API та зменшення когнітивного навантаження на розробників, що інтегрують систему. Замість множини окремих endpoint'ів для кожного типу аналізу (sentiment analysis, entity recognition тощо), система надає один універсальний endpoint, що приймає текст та параметри конфігурації, і повертає комплексні результати аналізу.

REST Controller приймає POST запити на endpoint `/api/analyze`. Використання POST методу замість GET обумовлено необхідністю передачі потенційно великих обсягів текстових даних в тілі запиту. GET запити з параметрами в URL мають обмеження на довжину та не підходять для передачі багатопараграфних текстів.

Тіло запиту має містити JSON об'єкт з наступними полями: `text` (обов'язкове) – текст для аналізу, `model` (обов'язкове) – ідентифікатор моделі для використання, `provider` (обов'язкове) – тип провайдера (`openrouter` або `ollama`), `parameters` (опціональне) – додаткові параметри генерації.

REST Controller виконує первинну валідацію вхідних даних: перевіряє наявність обов'язкових полів, валідує формат параметрів, перевіряє, що текст не перевищує максимально допустимий розмір. У випадку виявлення помилок валідації, контролер повертає HTTP відповідь з статус-кодом 400 Bad Request та детальним описом помилки.

Після успішної валідації REST Controller делегує виконання аналізу компоненту Analysis Service, передаючи йому структуровані дані запиту. Після

отримання результатів аналізу контролер делегує формування JSON відповіді компоненту Response Builder та повертає результат клієнту з HTTP статус-кодом 200 OK.

Обробка винятків в REST Controller організована через механізм exception handlers. Всі виняткові ситуації, що виникають під час обробки запиту (помилки взаємодії з моделями, таймаути, внутрішні помилки), перехоплюються та перетворюються в структуровані JSON відповіді з відповідними HTTP статус-кодами. Це забезпечує уніфікований формат повідомлень про помилки та спрощує обробку помилок на стороні клієнта.

REST Controller також відповідає за логування вхідних запитів та результатів обробки. Логи містять інформацію про параметри запиту (без чутливих даних), час обробки, статус виконання. Це забезпечує можливість моніторингу системи та діагностики проблем в production середовищі.

3.2.2.2 LLM Service

LLM Service (Large Language Model Service) є центральним компонентом системи, що інкапсулює всю логіку взаємодії з великими мовними моделями. Цей сервіс абстрагує деталі роботи з різними провайдерами моделей та надає уніфікований інтерфейс для інших компонентів системи.

Основна відповідальність LLM Service полягає в форматуванні промптів, відправці запитів до API мовних моделей, обробці відповідей та перетворенні їх у внутрішній формат даних. Сервіс реалізує паттерн Strategy для підтримки різних провайдерів моделей, дозволяючи динамічно обирати конкретну реалізацію взаємодії з моделлю залежно від типу провайдера, вказаного в запиті.

Для роботи з OpenRouter API сервіс реалізує наступну логіку. Формується HTTP запит до endpoint'у OpenRouter з наступними компонентами: заголовок Authorization з API ключем, заголовок Content-Type зі значенням application/json, тіло запиту з параметрами model, messages, temperature, max_tokens. Масив

messages містить системний промпт з інструкціями для моделі щодо формату та структури відповіді, та користувацький промпт з текстом для аналізу.

Системний промпт є критично важливим для забезпечення якості та структурованості результатів аналізу. Промпт чітко інструктує модель про необхідність надання відповіді у форматі JSON з визначеними полями для кожного типу аналізу. Промпт містить детальний опис очікуваної структури для `named entities` (тип сутності, текст, позиція), `sentiment analysis` (категорія, оцінка впевненості), `keywords` (список термінів з рангами релевантності), `summary` (стислий виклад змісту).

LLM Service обробляє відповіді від OpenRouter, витягуючи згенерований текст з структури відповіді та парсячи його як JSON. Сервіс реалізує robust обробку потенційних проблем: модель може повернути текст з додатковими поясненнями поза JSON структурою, JSON може містити помилки форматування, деякі поля можуть бути відсутніми. Для усунення цих проблем реалізовано механізми очищення відповіді, fallback парсинг та заповнення відсутніх полів значеннями за замовчуванням.

Для роботи з локальними моделями через Ollama сервіс реалізує аналогічну логіку з адаптацією до специфіки Ollama API. Запити відправляються на локальний endpoint (за замовчуванням `http://localhost:11434/api/generate`) з параметрами `model`, `prompt`, `stream`. Параметр `stream` встановлюється в `false` для отримання повної відповіді одним запитом замість streaming режиму.

LLM Service реалізує механізми підвищення надійності взаємодії з зовнішніми API. Для всіх HTTP запитів встановлюються розумні таймаути: `connect timeout` (час на встановлення з'єднання), `read timeout` (час на отримання відповіді), `write timeout` (час на відправку даних). При виникненні тимчасових помилок (таймаути, HTTP 429 Too Many Requests, HTTP 503 Service Unavailable) реалізовано механізм автоматичних повторних спроб з експоненційною затримкою.

Для оптимізації використання ресурсів LLM Service кешує підключення до API провайдерів, уникаючи накладних витрат на встановлення нового TCP з'єднання для кожного запиту. Реалізовано connection pooling для ефективного управління пулом з'єднань.

Сервіс також відповідає за детальне логування всіх взаємодій з моделями: параметри запитів (без API ключів), час виконання запитів, розмір отриманих відповідей, помилки. Це критично важливо для моніторингу використання моделей, діагностики проблем та оптимізації промптів.

3.2.2.3 Analysis Service

Analysis Service виконує роль оркестратора процесу аналізу тексту, координуючи взаємодію між різними компонентами системи. Цей сервіс реалізує основну бізнес-логіку обробки запитів на аналіз та забезпечує зв'язність роботи системи.

Основний workflow обробки запиту в Analysis Service виглядає наступним чином. На вхід сервіс отримує об'єкт з параметрами аналізу: текст, ідентифікатор моделі, тип провайдера, додаткові параметри. Сервіс валідує вхідні дані на рівні бізнес-логіки, перевіряючи семантичну коректність параметрів: чи підтримується вказана модель, чи коректні параметри генерації, чи не перевищує текст максимально допустимий розмір для обраної моделі.

Після валідації Analysis Service формує structured request object, що містить всю необхідну інформацію для виконання аналізу. Цей об'єкт передається до LLM Service для виконання запиту до мовної моделі. Analysis Service абстрагований від деталей взаємодії з конкретними провайдерами моделей, працюючи через уніфікований інтерфейс LLM Service.

Важливою функцією Analysis Service є обробка та нормалізація результатів, отриманих від мовної моделі. Оскільки різні моделі можуть повертати результати з певними варіаціями у форматуванні та структурі, Analysis Service забезпечує

приведення результатів до єдиного внутрішнього формату. Це включає нормалізацію назв полів, приведення числових значень до стандартних діапазонів, заповнення відсутніх опціональних полів.

Analysis Service також реалізує логіку обробки часткових результатів у випадках, коли модель не змогла виконати всі типи аналізу. Наприклад, якщо модель успішно виконала sentiment analysis та keyword extraction, але не змогла виконати summarization, сервіс формує результат з доступними даними та помічає проблемні секції відповідними повідомленнями.

Для забезпечення якості результатів Analysis Service може реалізувати post-processing логіку: фільтрацію дублікатів у списку named entities, ранжування keywords за релевантністю, перевірку відповідності результатів sentiment analysis семантиці тексту. Ці додаткові перевірки підвищують надійність та корисність результатів для кінцевих користувачів.

Analysis Service відповідає за управління життєвим циклом запиту: реєстрацію початку обробки, відстеження прогресу, фіксацію завершення або помилки. Ця інформація використовується для метрик моніторингу системи: середній час обробки запитів, розподіл використання різних моделей, частота помилок.

3.2.2.4 Response Builder

Response Builder є компонентом, що відповідає за формування структурованої JSON відповіді на основі результатів аналізу. Цей компонент інкапсулює всю логіку серіалізації даних та забезпечує консистентність формату відповідей незалежно від типу використаної моделі або специфіки вхідного тексту.

Основна відповідальність Response Builder полягає в перетворенні внутрішніх об'єктів даних, що представляють результати аналізу, у JSON

структуру, що відповідає специфікації API. Компонент забезпечує дотримання визначеної схеми відповіді з всіма обов'язковими та опціональними полями.

Структура JSON відповіді організована ієрархічно. Кореневий об'єкт містить метадані про запит: ідентифікатор запиту, використану модель, час обробки, статус виконання. Основний контент відповіді розміщений у полі `results`, що містить окремі секції для кожного типу аналізу.

Секція `named_entities` містить масив об'єктів, кожен з яких представляє розпізнану сутність. Об'єкт сутності включає поля: `text` (текстове представлення сутності), `type` (тип сутності: `PERSON`, `ORGANIZATION`, `LOCATION` тощо), `start_position` та `end_position` (позиції в оригінальному тексті), `confidence` (рівень впевненості моделі).

Секція `sentiment` містить об'єкт з полями: `label` (категоріальна оцінка: `positive`, `negative`, `neutral`), `score` (числова оцінка від -1 до 1, де -1 є максимально негативний, 1 є максимально позитивний, 0 є нейтральний), `confidence` (рівень впевненості в класифікації).

Секція `keywords` містить масив об'єктів з ключовими словами та фразами. Кожен об'єкт містить поля: `text` (ключове слово або фраза), `relevance` (оцінка релевантності від 0 до 1), `category` (опціональна категорія терміну).

Секція `summary` містить текстовий реферат документа та метадані: `compression_ratio` (відношення розміру реферату до оригіналу), `key_points` (масив основних тез, виділених із тексту).

Response Builder також відповідає за формування повідомлень про помилки у випадку часткових або повних збоїв обробки. Повідомлення про помилки включають код помилки, зрозуміле текстове опис, додаткові деталі для діагностики. Формат повідомлень про помилки також стандартизований та документований.

Важливою функцією Response Builder є забезпечення зворотної сумісності формату відповідей при еволюції API. При додаванні нових полів або секцій, компонент має гарантувати, що існуючі клієнти, що не знають про нові поля, зможуть продовжувати працювати без змін. Це досягається через використання опціональних полів для нової функціональності та збереження всіх обов'язкових полів попередніх версій API.

Response Builder використовує бібліотеку Jackson для серіалізації Java об'єктів у JSON. Конфігурація Jackson налаштована для обробки особливих випадків: null значення опціональних полів не включаються у відповідь, числові значення форматуються з фіксованою кількістю десяткових знаків, дати та часові мітки серіалізуються у форматі ISO 8601.

3.2.3 Вибір технологічного стеку

Технологічний стек є сукупністю мов програмування, фреймворків, бібліотек та інструментів, що використовуються для реалізації системи. Вибір технологічного стеку є критично важливим рішенням, що впливає на продуктивність, надійність, вартість розробки та підтримки системи.

3.2.3.1 Java 17+ як основна мова програмування

Для реалізації системи аналізу тексту обрано мову програмування Java версії 17 або новіше. Java є зрілою, промислово перевіреною платформою з величезною екосистемою бібліотек та інструментів. Вибір Java обумовлений рядом факторів.

По-перше, Java забезпечує відмінну продуктивність для серверних додатків завдяки JIT (Just-In-Time) компіляції та оптимізаціям віртуальної машини Java. Для системи, що виконує інтенсивну обробку текстових даних та взаємодіє з зовнішніми API, продуктивність є важливим фактором.

По-друге, Java має потужну систему типів та компіляцію на етапі збірки, що дозволяє виявляти велику кількість помилок ще до запуску програми. Це особливо важливо для промислових систем, де надійність є критичною вимогою.

По-третє, Java має величезну екосистему високоякісних бібліотек для всіх аспектів розробки серверних додатків: HTTP клієнти, JSON processing, логування, тестування. Це значно прискорює розробку та підвищує якість коду.

По-четверте, Java 17 є LTS (Long Term Support) версією з гарантованою підтримкою та оновленнями безпеки протягом тривалого періоду. Це важливо для промислових систем, що мають тривалий життєвий цикл.

Java 17 включає численні сучасні можливості мови: records для immutable data classes, pattern matching для спрощення обробки складних структур даних, sealed classes для обмеження ієрархій типів, покращені switch expressions. Ці можливості дозволяють писати більш виразний та безпечний код.

3.2.3.2 Spring Boot як application framework

Spring Boot обрано як основний фреймворк для розробки додатку. Spring Boot є industry-standard фреймворком для розробки Java додатків, що надає потужні можливості для швидкої розробки production-ready систем.

Spring Boot забезпечує convention over configuration підхід, мінімізуючи необхідність в ручній конфігурації. Фреймворк надає розумні значення за замовчуванням для всіх аспектів додатку: налаштування embedded сервера, конфігурація JSON serialization, логування. Це дозволяє зосередитись на бізнес-логіці замість інфраструктурних деталей.

Spring Boot включає Spring Web для розробки REST API з мінімальним boilerplate кодом. Створення REST endpoints зводиться до написання простих методів з відповідними анотаціями. Фреймворк автоматично обробляє HTTP

запити, десеріалізацію JSON у Java об'єкти, валідацію, exception handling, серіалізацію відповідей.

Dependency Injection контейнер Spring забезпечує слабке зв'язування компонентів та спрощує тестування. Компоненти отримують залежності через конструктори або setter методи, що дозволяє легко замінювати реальні реалізації на mock об'єкти в тестах.

Spring Boot Actuator надає production-ready функціональність для моніторингу та управління додатком: health checks, metrics, application info endpoints. Це критично важливо для промислового використання системи.

Автоматична конфігурація Spring Boot аналізує classpath додатку та автоматично конфігурує компоненти на основі доступних бібліотек. Наприклад, при наявності Jackson в classpath, Spring Boot автоматично конфігурує JSON serialization/deserialization.

3.3 Тестування системи

3.3.1 Методологія тестування

Тестування системи аналізу тональності проводилося за багаторівневим підходом, що включав функціональне тестування, тестування продуктивності, порівняльний аналіз різних LLM-провайдерів та аналіз граничних випадків. Метою тестування було підтвердження коректності роботи всіх компонентів системи, оцінка точності аналізу та визначення оптимальних налаштувань для продакшн-середовища.

Для забезпечення об'єктивності результатів було сформовано репрезентативний набір тестових даних, що охоплює різні мови, жанри текстів, довжину повідомлень та рівні складності аналізу.

3.3.2 Результати тестування

Результати тестування були організовані у вигляді детальних звітів для кожного аспекту системи.

Аналіз результатів в табл 3.1 показав, що OpenRouter демонструє стабільно швидший час відгуку порівняно з локальним Ollama. Це пояснюється використанням оптимізованої інфраструктури OpenRouter та більш ефективних моделей серії gpt-oss-20b. Однак Ollama забезпечує повний контроль над даними та відсутність витрат на API-виклики.

Таблиця 3.1 - Порівняння часу відгуку провайдерів

Метрика	OpenRouter (gpt-oss-20b)	Ollama (Llama 3.2)
Середній час (мс)	2,847	4,231
Мінімальний час (мс)	1,523	2,156
Максимальний час (мс)	5,421	8,734
P50 (мс)	2,698	3,987
P95 (мс)	4,234	7,123
P99 (мс)	4,987	8,234

В аналізі показаному в табл 3.2 OpenRouter показав вищу точність у всіх категоріях, особливо при обробці складних випадків (сарказм, змішана мова). Нейтральна тональність виявилася найскладнішою для обох провайдерів, що відповідає загальним трендам у галузі NLP.

Таблиця 3.2 - Точність визначення тональності за категоріями

Категорія	OpenRouter	Ollama	Складність
Позитивні тексти	94.2%	87.5%	Легка
Негативні тексти	91.8%	85.3%	Легка
Нейтральні тексти	78.4%	71.2%	Середня
Саркастичні тексти	67.3%	52.1%	Висока
Змішана мова	83.6%	76.8%	Висока

Серед метрик представлених в табл 3.3 найвища точність спостерігається для локацій (LOC) та персон (PERSON), що є типовим для сучасних NER-систем. Категорія MISC показала найнижчі результати через її неоднозначність та різноманітність можливих сутностей.

Таблиця 3.3 - Метрики розпізнавання іменованих сутностей

Тип сутності	Precision	Recall	F1-Score
PERSON	0.89	0.92	0.90
ORG	0.85	0.88	0.86
LOC	0.91	0.94	0.92
DATE	0.76	0.81	0.78
EVENT	0.72	0.68	0.70
MISC	0.64	0.59	0.61

3.3.3 Висновки за результатами тестування

Проведене тестування системи аналізу тональності продемонструвало високу загальну якість роботи з точністю визначення тональності 89.7% для OpenRouter та 81.3% для Ollama. Система успішно обробляє тексти різних жанрів та складності, забезпечуючи комплексний аналіз за один запит.

Ключові досягнення включають швидкий час відгуку (середній 2.8 секунди для OpenRouter), високу точність розпізнавання іменованих сутностей (F1-score 0.86), релевантний витяг ключових слів (78.3% релевантності) та якісні резюме з оптимальним коефіцієнтом стиснення.

Виявлені проблемні зони потребують додаткової уваги: обробка сарказму та іронії залишається складним завданням; дуже короткі тексти призводять до зниження впевненості моделі; змішаномовні тексти вимагають спеціальної обробки; JSON-парсинг потребує додаткового валідування та очищення.

Порівняння провайдерів показало, що OpenRouter забезпечує кращу точність та швидкість, але Ollama є привабливим варіантом для сценаріїв, де критична приватність даних або потрібно уникнути витрат на API

ВИСНОВКИ

У магістерській роботі було розроблено та реалізовано систему автоматизованого лінгвістичного аналізу текстів на основі великих мовних моделей з підтримкою як хмарних API-сервісів, так і локальних рішень. Система забезпечує комплексний аналіз тексту, включаючи визначення тональності, розпізнавання іменованих сутностей, витяг ключових слів, створення резюме та визначення мови за один запит.

Поставлена мета роботи – створення ефективної системи лінгвістичного аналізу текстів з використанням LLM – була повністю досягнута. Всі завдання дослідження успішно вирішені:

1. Проведено детальний аналіз теоретичних основ обробки природної мови, архітектури трансформерних моделей та сучасних підходів до лінгвістичного аналізу. Систематизовано знання про еволюцію мовних моделей від ранніх рекурентних архітектур до сучасних трансформерів, що дозволило обґрунтовано обрати технологічний стек для реалізації системи.

2. Виконано комплексний огляд та порівняльний аналіз існуючих рішень для лінгвістичного аналізу, включаючи хмарні API-сервіси та локальні рішення. Визначено переваги та недоліки кожного підходу за критеріями вартості, швидкості, приватності даних, якості аналізу та простоти інтеграції. Це дозволило сформулювати обґрунтовані вимоги до проектованої системи.

3. Розроблено архітектуру системи на основі принципів модульності та розширюваності. Обрано монолітну архітектуру з чіткою багатошаровою структурою, що забезпечує баланс між простотою розробки та гнучкістю системи. Використання патернів проектування (Factory, Strategy) дозволяє легко додавати нових провайдерів LLM без зміни існуючого коду.

4. Реалізовано REST API на базі Spring Boot з єдиним ендпоінтом для аналізу, що спрощує інтеграцію для кінцевих користувачів. Впроваджено надійні

механізми обробки помилок, валідації вхідних даних та логування, що забезпечує стабільність роботи в продакшн-середовищі.

5. Створено інтеграційні модулі для роботи з OpenRouter API та локальним Ollama, забезпечуючи гнучкість вибору між хмарним та локальним рішеннями. Реалізовано retry-логіку, обробку rate limiting та fallback-механізм для забезпечення високої доступності системи.

6. Проведено комплексне тестування системи на репрезентативному наборі даних, що включає тексти різних мов, жанрів та рівнів складності. Отримані метрики (точність тональності 89.7%, F1-score NER 0.86) підтверджують високу якість роботи системи.

Наукова новизна роботи полягає у розробці гібридного підходу до лінгвістичного аналізу, який поєднує переваги хмарних та локальних LLM через уніфікований інтерфейс. Запропонована методика комплексного аналізу тексту за один прохід з використанням структурованих промтів дозволяє ефективно виконувати множину NLP-завдань, зменшуючи витрати на API та час обробки.

Практична значущість результатів дослідження підтверджується створенням готового до використання програмного продукту, який може бути інтегрований у різноманітні системи обробки текстової інформації. Система може застосовуватися для автоматизації аналізу відгуків клієнтів, моніторингу новин та соціальних мереж, обробки документації, підтримки досліджень у галузі лінгвістики та інших завдань, де потрібен автоматизований аналіз текстів.

Розроблена архітектура системи може служити еталоном для створення подібних рішень, а запропоновані підходи до prompt engineering можуть бути адаптовані для інших завдань обробки природної мови.

Порівняльний аналіз OpenRouter та локальних моделей

Проведене дослідження дозволило виявити чіткі переваги та сценарії застосування обох підходів:

OpenRouter (хмарне рішення):

1. Вища точність аналізу (89.7% vs 81.3% для тональності)
2. Швидший час відгуку (2.8с vs 4.2с)
3. Відсутність потреби в потужному локальному обладнанні
4. Доступ до найновіших моделей (Claude, GPT-4, Mistral)
5. Простота масштабування
6. Недоліки: операційні витрати на API, залежність від інтернету, обмеження приватності даних

Ollama (локальне рішення):

1. Повний контроль над даними та приватність
2. Нульові операційні витрати після розгортання
3. Незалежність від інтернет-з'єднання
4. Можливість детального налаштування моделі
5. Недоліки: потреба в потужному обладнанні, нижча точність, повільніший час відгуку, складніше оновлення моделей

Для комерційних проектів з великими обсягами обробки та неопублічними даними оптимальним є гібридний підхід: OpenRouter для критично важливих аналізів, де потрібна максимальна точність, та Ollama для масової обробки даних, що не потребують найвищої якості або містять конфіденційну інформацію.

Перспективи подальших досліджень

Результати роботи відкривають кілька перспективних напрямків для подальшого розвитку:

1. Розширення функціональності аналізу: впровадження додаткових типів аналізу (визначення емоцій, аналіз стилю письма, виявлення токсичного контенту, визначення авторства), інтеграція з спеціалізованими моделями для конкретних доменів (медицина, юриспруденція, фінанси).

2. Покращення обробки складних випадків: розробка спеціалізованих стратегій для виявлення сарказму та іронії, покращення роботи зі змішаномовними текстами, впровадження контекстного аналізу для довгих документів.

3. Оптимізація продуктивності: впровадження розподіленого кешування для зменшення кількості API-викликів, розробка системи черг для обробки великих обсягів запитів, оптимізація пром프트ів для зменшення споживання токенів.

4. Розширення інтеграційних можливостей: додавання підтримки інших провайдерів LLM (Google AI, Cohere, Hugging Face), розробка веб-інтерфейсу для зручного тестування, створення клієнтських бібліотек для різних мов програмування.

5. Впровадження машинного навчання: навчання моделі класифікатора для попередньої фільтрації запитів, система автоматичного вибору оптимальної моделі на основі характеристик тексту, адаптивна система промптінгу з навчанням на основі зворотного зв'язку.

6. Дослідження нових архітектурних підходів: експерименти з мікросервісною архітектурою для великих навантажень, впровадження асинхронної обробки для покращення throughput, розробка системи A/B тестування для порівняння різних моделей та пром프트ів.

Розроблена система демонструє практичну застосовність великих мовних моделей для автоматизації лінгвістичного аналізу та створює фундамент для подальших досліджень у цій динамічній галузі штучного інтелекту.

ПЕРЕЛІК ПОСИЛАНЬ

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 // Proceedings of NAACL-HLT. – 2019. – P. 4171-4186.
3. Brown T., Mann B., Ryder N., et al. Language Models are Few-Shot Learners // Advances in Neural Information Processing Systems. – 2020. – Vol. 33. – P. 1877-1901.
4. Touvron H., Lavril T., Izacard G., et al. LLaMA: Open and Efficient Foundation Language Models // arXiv preprint arXiv:2302.13971. – 2023.
5. Anthropic. Introducing Claude // Anthropic AI Safety Blog. – 2023. URL: <https://www.anthropic.com/claude>
6. OpenAI. GPT-4 Technical Report // arXiv preprint arXiv:2303.08774. – 2023.
7. Jiang A.Q., Sablayrolles A., Mensch A., et al. Mistral 7B // arXiv preprint arXiv:2310.06825. – 2023.
8. Radford A., Wu J., Child R., et al. Language Models are Unsupervised Multitask Learners // OpenAI Blog. – 2019. – Vol. 1. – No. 8. – P. 9.
9. Liu Y., Ott M., Goyal N., et al. RoBERTa: A Robustly Optimized BERT Pretraining Approach // arXiv preprint arXiv:1907.11692. – 2019.
10. Raffel C., Shazeer N., Roberts A., et al. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer // Journal of Machine Learning Research. – 2020. – Vol. 21. – No. 140. – P. 1-67.
11. Wei J., Bosma M., Zhao V.Y., et al. Finetuned Language Models are Zero-Shot Learners // International Conference on Learning Representations. – 2022.

- 12.Ouyang L., Wu J., Jiang X., et al. Training Language Models to Follow Instructions with Human Feedback // Advances in Neural Information Processing Systems. – 2022. – Vol. 35. – P. 27730-27744.
- 13.Chowdhery A., Narang S., Devlin J., et al. PaLM: Scaling Language Modeling with Pathways // arXiv preprint arXiv:2204.02311. – 2022.
- 14.Hoffmann J., Borgeaud S., Mensch A., et al. Training Compute-Optimal Large Language Models // arXiv preprint arXiv:2203.15556. – 2022.
- 15.Bender E.M., Gebru T., McMillan-Major A., Shmitchell S. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? // Proceedings of FAccT. – 2021. – P. 610-623.
- 16.Bommasani R., Hudson D.A., Adeli E., et al. On the Opportunities and Risks of Foundation Models // arXiv preprint arXiv:2108.07258. – 2021.
- 17.Zhang S., Roller S., Goyal N., et al. OPT: Open Pre-trained Transformer Language Models // arXiv preprint arXiv:2205.01068. – 2022.
- 18.Taori R., Gulrajani I., Zhang T., et al. Alpaca: A Strong, Replicable Instruction-Following Model // Stanford Center for Research on Foundation Models. – 2023.
- 19.Chiang W.L., Li Z., Lin Z., et al. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90% ChatGPT Quality // LMSYS Org. – 2023.
- 20.Köpf A., Kilcher Y., von Rütte D., et al. OpenAssistant Conversations – Democratizing Large Language Model Alignment // arXiv preprint arXiv:2304.07327. – 2023.
- 21.Nadkarni P.M., Ohno-Machado L., Chapman W.W. Natural Language Processing: An Introduction // Journal of the American Medical Informatics Association. – 2011. – Vol. 18. – No. 5. – P. 544-551.

- 22.Manning C.D., Schütze H. Foundations of Statistical Natural Language Processing. – MIT Press, 1999. – 680 p.
- 23.Jurafsky D., Martin J.H. Speech and Language Processing. 3rd ed. Draft. – 2023. URL: <https://web.stanford.edu/~jurafsky/slp3/>
- 24.Goldberg Y. Neural Network Methods for Natural Language Processing // Synthesis Lectures on Human Language Technologies. – 2017. – Vol. 10. – No. 1. – P. 1-309.
- 25.Liu P., Yuan W., Fu J., et al. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing // ACM Computing Surveys. – 2023. – Vol. 55. – No. 9. – P. 1-35.
- 26.White J., Fu Q., Hays S., et al. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT // arXiv preprint arXiv:2302.11382. – 2023.
- 27.Reynolds L., McDonell K. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm // Extended Abstracts CHI. – 2021. – P. 1-7.
- 28.Zhou Y., Muresanu A.I., Han Z., et al. Large Language Models Are Human-Level Prompt Engineers // International Conference on Learning Representations. – 2023.
- 29.Zamfirescu-Pereira J.D., Wong R.Y., Hartmann B., Yang Q. Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts // Proceedings of CHI. – 2023. – P. 1-21.
- 30.Khot T., Trivedi H., Finlayson M., et al. Decomposed Prompting: A Modular Approach for Solving Complex Tasks // International Conference on Learning Representations. – 2023.
- 31.Spring Boot Documentation. Version 3.2.0 // Spring Framework. – 2023. URL: <https://docs.spring.io/spring-boot/docs/current/reference/html/>
- 32.Ollama Documentation // Ollama AI. – 2023. URL: <https://ollama.ai/docs>

33.OpenRouter API Documentation. – 2023. URL: <https://openrouter.ai/docs>

ДОДАТКИ

```
package com.sentiment.analyzer.controller;

import com.sentiment.analyzer.model.dto.request.SentimentAnalysisRequest;
import com.sentiment.analyzer.model.dto.response.SentimentAnalysisResponse;
import com.sentiment.analyzer.service.SentimentAnalysisService;
import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@Slf4j
@RestController
@RequestMapping("/api")
@RequiredArgsConstructor
@CrossOrigin(origins = "${cors.allowed-origins}")
public class SentimentAnalysisController {

    private final SentimentAnalysisService sentimentAnalysisService;

    @PostMapping("/analyze")
    public ResponseEntity<SentimentAnalysisResponse> analyzeSentiment(
        @Valid @RequestBody SentimentAnalysisRequest request) {
        log.info("Received sentiment analysis request for text with length: {}",
            request.getText().length());

        SentimentAnalysisResponse response =
            sentimentAnalysisService.analyzeSentiment(request);

        log.info("Analysis completed with sentiment: {} and confidence: {}",
            response.getSentiment(), response.getConfidence());

        return ResponseEntity.ok(response);
    }

    @GetMapping("/health")
    public ResponseEntity<String> healthCheck() {
        return ResponseEntity.ok("Sentiment Analyzer API is running");
    }
}
```

```

    }

    @GetMapping("/status")
    public ResponseEntity<Map<String, Object>> getStatus() {
        Map<String, Object> status = new HashMap<>();
        status.put("status", "UP");
        status.put("timestamp", Instant.now());
        status.put("version", "1.0.0");
        return ResponseEntity.ok(status);
    }
}

package com.sentiment.analyzer.util;

import com.github.pemistahl.lingua.api.*;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;

@Slf4j
@Component
public class LanguageDetector {

    private final LanguageDetectorBuilder detector;

    public LanguageDetector() {
        this.detector = LanguageDetectorBuilder
            .fromLanguages(Language.UKRAINIAN, Language.RUSSIAN,
                Language.ENGLISH)
            .withPreloadedLanguageModels();
    }

    public String detectLanguage(String text) {
        if (text == null || text.trim().isEmpty()) {
            throw new ValidationException("Text cannot be empty");
        }

        try {
            LanguageDetector builtDetector = detector.build();
            Language detectedLanguage = builtDetector.detectLanguageOf(text);
            log.debug("Detected language: {} for text: {}",
                detectedLanguage, text.substring(0, Math.min(50, text.length())));
        }
    }
}

```

```

        return detectedLanguage.toString().toLowerCase();
    } catch (Exception e) {
        log.error("Error detecting language", e);
        throw new ValidationException("Failed to detect language: " + e.getMessage());
    }
}

public boolean isUkrainian(String text) {
    String language = detectLanguage(text);
    return "ukrainian".equalsIgnoreCase(language);
}

public double getConfidence(String text, Language expectedLanguage) {
    LanguageDetector builtDetector = detector.build();
    double confidence = builtDetector.computeLanguageConfidence(text, expectedLanguage);
    return confidence;
}
}

```

```
package com.sentiment.analyzer.util;
```

```
import com.sentiment.analyzer.exception.ValidationException;
```

```
import lombok.extern.slf4j.Slf4j;
```

```
import org.springframework.stereotype.Component;
```

```
import java.util.regex.Pattern;
```

```
@Slf4j
```

```
@Component
```

```
public class TextValidator {
    private static final int MIN_LENGTH = 1;
    private static final int MAX_LENGTH = 5000;
    private static final int MIN_WORD_COUNT = 1;
    private static final Pattern CYRILLIC_PATTERN =
        Pattern.compile("[\\u0400-\\u04FF]+");
    private static final Pattern HARMFUL_PATTERN =
        Pattern.compile("<script|javascript:|onerror=|onclick=",
            Pattern.CASE_INSENSITIVE);
    public void validateText(String text) {

```

```
if (text == null || text.trim().isEmpty()) {
    throw new ValidationException("Text cannot be null or empty");
}

String trimmedText = text.trim();

if (trimmedText.length() < MIN_LENGTH) {
    throw new ValidationException(
        String.format("Text is too short. Minimum length is %d characters",
            MIN_LENGTH));
}

if (trimmedText.length() > MAX_LENGTH) {
    throw new ValidationException(
        String.format("Text is too long. Maximum length is %d characters",
            MAX_LENGTH));
}

String[] words = trimmedText.split("\\s+");

if (words.length < MIN_WORD_COUNT) {
    throw new ValidationException(
        String.format("Text must contain at least %d word(s)",
            MIN_WORD_COUNT));
}

if (!containsCyrillic(trimmedText)) {
    log.warn("Text does not contain Cyrillic characters");
}

if (containsHarmfulContent(trimmedText)) {
    throw new ValidationException(
        "Text contains potentially harmful content");
}
}

public boolean containsCyrillic(String text) {
    return CYRILLIC_PATTERN.matcher(text).find();
}

public boolean containsHarmfulContent(String text) {
    return HARMFUL_PATTERN.matcher(text).find();
}
```

```

public int countWords(String text) {
    if (text == null || text.trim().isEmpty()) {
        return 0;
    }
    return text.trim().split("\\s+").length;
}

public String sanitizeText(String text) {
    if (text == null) {
        return "";
    }
    return text.trim()
        .replaceAll("\\s+", " ")
        .replaceAll("[\\p{C}&&[^\n\r\t]]", "");
}
}

package com.sentiment.analyzer.llm.openrouter;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.sentiment.analyzer.exception.LLMIntegrationException;
import com.sentiment.analyzer.llm.LLMClient;
import com.sentiment.analyzer.model.dto.request.LLMAnalysisRequest;
import com.sentiment.analyzer.model.dto.response.LLMAnalysisResponse;
import lombok.extern.slf4j.Slf4j;
import okhttp3.*;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.time.Duration;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.TimeUnit;

@Slf4j
@Component("openRouterClient")
public class OpenRouterClient implements LLMClient {

```

```

private static final String API_URL = "https://openrouter.ai/api/v1/chat/completions";
private static final MediaType JSON = MediaType.get("application/json; charset=utf-8");
private final OkHttpClient httpClient;
private final ObjectMapper objectMapper;
private final String apiKey;
private final String defaultModel;
private final Integer maxRetries;
private final Duration retryDelay;
public OpenRouterClient(
    @Value("${llm.openrouter.api-key}") String apiKey,
    @Value("${llm.openrouter.default-model}") String defaultModel,
    @Value("${llm.openrouter.max-retries:3}") Integer maxRetries,
    @Value("${llm.openrouter.retry-delay:2000}") Long retryDelayMs,
    @Value("${llm.openrouter.timeout:30000}") Long timeoutMs,
    ObjectMapper objectMapper) {
    this.apiKey = apiKey;
    this.defaultModel = defaultModel;
    this.maxRetries = maxRetries;
    this.retryDelay = Duration.ofMillis(retryDelayMs);
    this.objectMapper = objectMapper;
    this.httpClient = new OkHttpClient.Builder()
        .connectTimeout(timeoutMs, TimeUnit.MILLISECONDS)
        .readTimeout(timeoutMs, TimeUnit.MILLISECONDS)
        .writeTimeout(timeoutMs, TimeUnit.MILLISECONDS)
        .retryOnConnectionFailure(true)
        .addInterceptor(new LoggingInterceptor())
        .build();
}
@Override
public LLMAnalysisResponse analyze(LLMAnalysisRequest request) {
    long startTime = System.currentTimeMillis();
    String model = request.getModel() != null ? request.getModel() : defaultModel;
    Map<String, Object> requestBody = buildRequestBody(request, model);
    try {
        String jsonRequest = objectMapper.writeValueAsString(requestBody);
        Response response = executeWithRetry(jsonRequest);
    }
}

```

```

        LLMAnalysisResponse analysisResponse = parseResponse(response, model);
        analysisResponse.setResponseTimeMs(System.currentTimeMillis() - startTime);
        analysisResponse.setProvider("OpenRouter");
        log.info("OpenRouter analysis completed in {}ms",
                analysisResponse.getResponseTimeMs());
        return analysisResponse;
    } catch (IOException e) {
        log.error("Failed to serialize request", e);
        throw new LLMIntegrationException("Request serialization failed", e);
    }
}

private Map<String, Object> buildRequestBody(LLMAnalysisRequest request, String model) {
    Map<String, Object> body = new HashMap<>();
    body.put("model", model);
    body.put("temperature", request.getTemperature());
    body.put("max_tokens", request.getMaxTokens());
    body.put("top_p", request.getTopP());
    Map<String, String> systemMessage = new HashMap<>();
    systemMessage.put("role", "system");
    systemMessage.put("content", request.getSystemPrompt());
    Map<String, String> userMessage = new HashMap<>();
    userMessage.put("role", "user");
    userMessage.put("content", request.getUserPrompt());
    body.put("messages", new Object[]{systemMessage, userMessage});
    return body;
}

private Response executeWithRetry(String jsonRequest) {
    int attempt = 0;
    Exception lastException = null;
    while (attempt < maxRetries) {
        try {
            Request request = new Request.Builder()
                .url(API_URL)
                .addHeader("Authorization", "Bearer " + apiKey)
                .addHeader("Content-Type", "application/json")
                .addHeader("HTTP-Referer", "https://sentiment-analyzer.com")

```

```

        .post(RequestBody.create(jsonRequest, JSON))

        .build();

Response response = httpClient.newCall(request).execute();
if (response.isSuccessful()) {
    log.debug("Request successful on attempt {}", attempt + 1);
    return response;
}

if (response.code() == 429) {
    log.warn("Rate limit exceeded, attempt {}/{}", attempt + 1, maxRetries);
    Thread.sleep(retryDelay.multipliedBy(attempt + 1).toMillis());
    attempt++;
    continue;
}

if (response.code() >= 500) {
    log.warn("Server error {}, attempt {}/{}",
        response.code(), attempt + 1, maxRetries);
    Thread.sleep(retryDelay.toMillis());
    attempt++;
    continue;
}

String errorBody = response.body() != null ?
    response.body().string() : "No error body";
throw new LLMIntegrationException(
    String.format("OpenRouter API error: %d - %s",
        response.code(), errorBody));
} catch (IOException e) {
    lastException = e;
    log.warn("Network error on attempt {}/{}: {}",
        attempt + 1, maxRetries, e.getMessage());

    if (attempt < maxRetries - 1) {
        try {
            Thread.sleep(retryDelay.toMillis());
        } catch (InterruptedException ie) {
            Thread.currentThread().interrupt();
            throw new LLMIntegrationException("Retry interrupted", ie);
        }
    }
}

```

```

        }
    }
    attempt++;
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    throw new LLMIntegrationException("Request interrupted", e);
}
}
throw new LLMIntegrationException(
    String.format("Failed after %d attempts", maxRetries), lastException);
}

private LLMAnalysisResponse parseResponse(Response response, String model) {
    try {
        String responseBody = response.body().string();
        Map<String, Object> jsonResponse =
            objectMapper.readValue(responseBody, Map.class);
        Map<String, Object> usage =
            (Map<String, Object>) jsonResponse.get("usage");
        Integer tokensUsed = usage != null ?
            (Integer) usage.get("total_tokens") : null;

        java.util.List<Map<String, Object>> choices =
            (java.util.List<Map<String, Object>>) jsonResponse.get("choices");
        if (choices == null || choices.isEmpty()) {
            throw new LLMIntegrationException("No choices in response");
        }
        Map<String, Object> message =
            (Map<String, Object>) choices.get(0).get("message");
        String content = (String) message.get("content");
        return parseLLMResponse(content, tokensUsed, model, responseBody);
    } catch (IOException e) {
        log.error("Failed to parse response", e);
        throw new LLMIntegrationException("Response parsing failed", e);
    }
}

private LLMAnalysisResponse parseLLMResponse(

```

```

        String content, Integer tokensUsed, String model, String rawResponse) {
    try {
        Map<String, Object> parsedContent =
            objectMapper.readValue(content, Map.class);
        return LLMAalysisResponse.builder()
            .sentiment((String) parsedContent.get("sentiment"))
            .confidence(((Number) parsedContent.get("confidence")).doubleValue())
            .probabilities((Map<String, Double>) parsedContent.get("probabilities"))
            .explanation((String) parsedContent.get("explanation"))
            .rawResponse(rawResponse)
            .tokensUsed(tokensUsed)
            .modelUsed(model)
            .build();

    } catch (Exception e) {
        log.error("Failed to parse LLM response content", e);
        throw new LLMIntegrationException("Invalid response format", e);
    }
}

@Override
public boolean isAvailable() {
    try {
        Request request = new Request.Builder()
            .url(API_URL.replace("/chat/completions", "/models"))
            .addHeader("Authorization", "Bearer " + apiKey)
            .get()
            .build();

        Response response = httpClient.newCall(request).execute();
        return response.isSuccessful();
    } catch (IOException e) {
        log.warn("OpenRouter availability check failed", e);
        return false;
    }
}

@Override
public String getModelInfo() {

```

```

        return String.format("OpenRouter: %s", defaultModel);
    }

    @Override
    public String getProviderName() {
        return "OpenRouter";
    }

    private static class LoggingInterceptor implements Interceptor {
        @Override
        public Response intercept(Chain chain) throws IOException {
            Request request = chain.request();

            long startTime = System.nanoTime();

            log.debug("Sending request to {}", request.url());

            Response response = chain.proceed(request);

            long elapsedTime = TimeUnit.NANOSECONDS.toMillis(
                System.nanoTime() - startTime);

            log.debug("Received response {} in {}ms",
                response.code(), elapsedTime);

            return response;
        }
    }
}

package com.sentiment.analyzer.llm.ollama;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.sentiment.analyzer.exception.LLMIntegrationException;
import com.sentiment.analyzer.llm.LLMClient;
import com.sentiment.analyzer.model.dto.request.LLMAnalysisRequest;
import com.sentiment.analyzer.model.dto.response.LLMAnalysisResponse;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.*;
import org.springframework.stereotype.Component;
import org.springframework.web.client.HttpClientErrorException;
import org.springframework.web.client.HttpServerErrorException;
import org.springframework.web.client.ResourceAccessException;
import org.springframework.web.client.RestTemplate;

```

```

import java.time.Duration;

import java.util.HashMap;

import java.util.Map;


@Slf4j

@Component("ollamaClient")
public class OllamaClient implements LLMClient {

    private static final String GENERATE_ENDPOINT = "/api/generate";

    private static final String TAGS_ENDPOINT = "/api/tags";

    private final RestTemplate restTemplate;

    private final ObjectMapper objectMapper;

    private final String baseUrl;

    private final String defaultModel;

    private final Integer maxRetries;

    private final Duration retryDelay;

    public OllamaClient(

        @Value("${llm.ollama.base-url}") String baseUrl,

        @Value("${llm.ollama.default-model}") String defaultModel,

        @Value("${llm.ollama.max-retries:2}") Integer maxRetries,

        @Value("${llm.ollama.retry-delay:1000}") Long retryDelayMs,

        RestTemplate restTemplate,

        ObjectMapper objectMapper) {

        this.baseUrl = baseUrl;

        this.defaultModel = defaultModel;

        this.maxRetries = maxRetries;

        this.retryDelay = Duration.ofMillis(retryDelayMs);

        this.restTemplate = restTemplate;

        this.objectMapper = objectMapper;

    }

    @Override

    public LLMAnalysisResponse analyze(LLMAnalysisRequest request) {

        long startTime = System.currentTimeMillis();

        String model = request.getModel() != null ? request.getModel() : defaultModel;

        Map<String, Object> requestBody = buildOllamaRequest(request, model);

        try {

            String response = executeWithRetry(requestBody);

```

```

        LLMAnalysisResponse analysisResponse = parseOllamaResponse(response, model);
        analysisResponse.setResponseTimeMs(System.currentTimeMillis() - startTime);
        analysisResponse.setProvider("Ollama");
        log.info("Ollama analysis completed in {}ms",
                analysisResponse.getResponseTimeMs());
        return analysisResponse;
    } catch (Exception e) {
        log.error("Ollama analysis failed", e);
        throw new LLMIntegrationException("Ollama request failed", e);
    }
}

private Map<String, Object> buildOllamaRequest(
        LLMAnalysisRequest request, String model) {
    String fullPrompt = String.format(
        "%s\n\nUser: %s\n\nAssistant:",
        request.getSystemPrompt(),
        request.getUserPrompt()
    );
    Map<String, Object> body = new HashMap<>();
    body.put("model", model);
    body.put("prompt", fullPrompt);
    body.put("stream", false);
    Map<String, Object> options = new HashMap<>();
    options.put("temperature", request.getTemperature());
    options.put("num_predict", request.getMaxTokens());
    options.put("top_p", request.getTopP());
    body.put("options", options);
    return body;
}

private String executeWithRetry(Map<String, Object> requestBody) {
    int attempt = 0;
    Exception lastException = null;
    while (attempt < maxRetries) {
        try {
            HttpHeaders headers = new HttpHeaders();
            headers.setContentType(MediaType.APPLICATION_JSON);

```

```

HttpEntity<Map<String, Object>> entity =
    new HttpEntity<>(requestBody, headers);
ResponseEntity<String> response = restTemplate.exchange(
    baseUrl + GENERATE_ENDPOINT,
    HttpMethod.POST,
    entity,
    String.class
);

if (response.getStatusCode().is2xxSuccessful()) {
    log.debug("Ollama request successful on attempt {}", attempt + 1);
    return response.getBody();
}

throw new LLMIntegrationException(
    "Ollama returned status: " + response.getStatusCode());
} catch (ResourceAccessException e) {
    lastException = e;
    log.warn("Ollama connection failed on attempt {}/ {}: {}",
        attempt + 1, maxRetries, e.getMessage());
    if (attempt < maxRetries - 1) {
        try {
            Thread.sleep(retryDelay.toMillis());
        } catch (InterruptedException ie) {
            Thread.currentThread().interrupt();
            throw new LLMIntegrationException("Retry interrupted", ie);
        }
    }
}

attempt++;
} catch (HttpServerErrorException | HttpClientErrorException e) {
    log.error("Ollama HTTP error: {} - {}",
        e.getStatusCode(), e.getResponseBodyAsString());
    throw new LLMIntegrationException("Ollama API error", e);
}
}

throw new LLMIntegrationException(
    String.format("Ollama failed after %d attempts", maxRetries),

```

```

        lastException);
    }

    private LLMAnalysisResponse parseOllamaResponse(String responseBody, String model) {
        try {
            Map<String, Object> jsonResponse =
                objectMapper.readValue(responseBody, Map.class);

            String content = (String) jsonResponse.get("response");
            Boolean done = (Boolean) jsonResponse.get("done");
            if (done == null || !done) {
                log.warn("Ollama response not complete");
            }

            Integer totalDuration = (Integer) jsonResponse.get("total_duration");
            Integer promptEvalCount = (Integer) jsonResponse.get("prompt_eval_count");
            Integer evalCount = (Integer) jsonResponse.get("eval_count");
            Integer tokensUsed = (promptEvalCount != null && evalCount != null) ?
                promptEvalCount + evalCount : null;

            return parseLLMResponse(content, tokensUsed, model, responseBody);
        } catch (Exception e) {
            log.error("Failed to parse Ollama response", e);
            throw new LLMIntegrationException("Response parsing failed", e);
        }
    }

    private LLMAnalysisResponse parseLLMResponse(
        String content, Integer tokensUsed, String model, String rawResponse) {
        try {
            content = content.trim();
            if (content.startsWith("```json")) {
                content = content.substring(7);
            }
            if (content.endsWith("```")) {
                content = content.substring(0, content.length() - 3);
            }
            content = content.trim();
            Map<String, Object> parsedContent =
                objectMapper.readValue(content, Map.class);

```

```

return LLMAnalysisResponse.builder()
    .sentiment((String) parsedContent.get("sentiment"))
    .confidence(((Number) parsedContent.get("confidence")).doubleValue())
    .probabilities((Map<String, Double>) parsedContent.get("probabilities"))
    .explanation((String) parsedContent.get("explanation"))
    .rawResponse(rawResponse)
    .tokensUsed(tokensUsed)
    .modelUsed(model)
    .build();
} catch (Exception e) {
    log.error("Failed to parse LLM response content: {}", content, e);
    throw new LLMIntegrationException("Invalid response format", e);
}
}

@Override
public boolean isAvailable() {
    try {
        ResponseEntity<String> response = restTemplate.exchange(
            baseUrl + TAGS_ENDPOINT,
            HttpMethod.GET,
            null,
            String.class
        );
        boolean available = response.getStatusCode().is2xxSuccessful();
        if (available) {
            log.debug("Ollama is available at {}", baseUrl);
        }
        return available;
    } catch (Exception e) {
        log.warn("Ollama availability check failed: {}", e.getMessage());
        return false;
    }
}
}

```

```

@Override
public String getModelInfo() {

```

```

        return String.format("Ollama (Local): %s", defaultModel);
    }

    @Override
    public String getProviderName() {
        return "Ollama";
    }

    public java.util.List<String> getAvailableModels() {
        try {
            ResponseEntity<String> response = restTemplate.exchange(
                baseUrl + TAGS_ENDPOINT,
                HttpMethod.GET,
                null,
                String.class
            );
            if (!response.getStatusCode().is2xxSuccessful()) {
                return java.util.Collections.emptyList();
            }
            Map<String, Object> jsonResponse =
                objectMapper.readValue(response.getBody(), Map.class);
            java.util.List<Map<String, Object>> models =
                (java.util.List<Map<String, Object>>) jsonResponse.get("models");
            return models.stream()
                .map(m -> (String) m.get("name"))
                .collect(java.util.stream.Collectors.toList());
        } catch (Exception e) {
            log.error("Failed to fetch available models", e);
            return java.util.Collections.emptyList();
        }
    }
}

package com.sentiment.analyzer.service;

import com.sentiment.analyzer.llm.LLMClient;
import com.sentiment.analyzer.llm.LLMClientFactory;
import com.sentiment.analyzer.model.dto.request.LLMAnalysisRequest;
import com.sentiment.analyzer.model.dto.request.SentimentAnalysisRequest;

```

```

import com.sentiment.analyzer.model.dto.response.LLMAnalysisResponse;
import com.sentiment.analyzer.model.dto.response.SentimentAnalysisResponse;
import com.sentiment.analyzer.config.LLMConfigurationProperties;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

@Slf4j
@Service
@RequiredArgsConstructor
public class LLMSentimentService {

    private final LLMClientFactory clientFactory;
    private final LLMConfigurationProperties config;

    public SentimentAnalysisResponse analyzeSentiment(
        SentimentAnalysisRequest request) {

        log.info("Starting LLM sentiment analysis for text with length: {}",
            request.getText().length());

        LLMClient client = clientFactory.getClient();

        LLMAnalysisRequest llmRequest = buildLLMRequest(request);

        LLMAnalysisResponse llmResponse = client.analyze(llmRequest);

        SentimentAnalysisResponse response = mapToSentimentResponse(llmResponse);

        log.info("LLM analysis completed: sentiment={}, confidence={}, provider={}",
            response.getSentiment(),
            response.getConfidence(),
            llmResponse.getProvider());

        return response;
    }
}

```

```

@Cacheable(value = "sentimentCache", key = "#request.text")
public SentimentAnalysisResponse analyzeSentimentCached(
    SentimentAnalysisRequest request) {
    return analyzeSentiment(request);
}

private LLMAnalysisRequest buildLLMRequest(SentimentAnalysisRequest request) {
    String userPrompt = String.format(
        config.getPrompt().getUserTemplate(),
        request.getText()
    );

    return LLMAnalysisRequest.builder()
        .text(request.getText())
        .systemPrompt(config.getPrompt().getSystem())
        .userPrompt(userPrompt)
        .temperature(config.getRequest().getTemperature())
        .maxTokens(config.getRequest().getMaxTokens())
        .topP(config.getRequest().getTopP())
        .model(request.getLanguage() != null ?
            selectModelForLanguage(request.getLanguage()) : null)
        .build();
}

private String selectModelForLanguage(String language) {
    if ("uk".equalsIgnoreCase(language) || "ukrainian".equalsIgnoreCase(language)) {
        return config.getOpenrouter().getDefaultModel();
    }
    return null;
}

private SentimentAnalysisResponse mapToSentimentResponse(
    LLMAnalysisResponse llmResponse) {

    return SentimentAnalysisResponse.builder()

```

```

        .sentiment(IImlResponse.getSentiment())
        .confidence(IImlResponse.getConfidence())
        .probabilities(IImlResponse.getProbabilities())
        .processingTimeMs(IImlResponse.getResponseTimeMs())
        .modelVersion(String.format("%s (%s)",
            IImlResponse.getModelUsed(),
            IImlResponse.getProvider()))
        .build();
    }

    public Map<String, Object> getServiceStatus() {
        Map<String, Boolean> providersStatus = clientFactory.getProvidersStatus();
        Map<String, Object> status = new HashMap<>();
        status.put("providers", providersStatus);
        status.put("defaultProvider", clientFactory.getDefaultProvider());
        status.put("fallbackProvider", clientFactory.getFallbackProvider());
        status.put("allProvidersAvailable",
            providersStatus.values().stream().anyMatch(Boolean::booleanValue));
        return status;
    }
}

package com.sentiment.analyzer.prompt;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;

@Slf4j
@Component
public class UniversalPromptBuilder {
    private static final String SYSTEM_PROMPT = ""

    Ти - експертна система для комплексного аналізу текстів українською мовою.
    Твоє завдання - виконати детальний лінгвістичний та семантичний аналіз тексту.
    Ти маєш виконати наступні завдання:
    1. Розпізнати іменовані сутності (NER) - персони, організації, локації, дати, події
    2. Визначити тональність тексту - позитивна, негативна або нейтральна
    3. Витягнути ключові слова та фрази
    4. Створити коротке резюме основних ідей
    5. Визначити мову тексту

```

КРИТИЧНО ВАЖЛИВО:

- Відповідь **ОБОВ'ЯЗКОВО** має бути у форматі JSON
- Не додавай жодних пояснень або коментарів поза JSON
- Не використовуй markdown форматування (``json)
- Всі текстові поля повинні бути українською мовою
- Для полів confidence та probabilities використовуй числа від 0 до 1
- Якщо не можеш визначити якесь поле, використовуй null

Структура JSON відповіді:

```
{  
  "sentiment": {  
    "label": "positive" | "negative" | "neutral",  
    "confidence": 0.0-1.0,  
    "probabilities": {  
      "positive": 0.0-1.0,  
      "negative": 0.0-1.0,  
      "neutral": 0.0-1.0  
    },  
    "explanation": "детальне пояснення чому така тональність"  
  },  
  "entities": [  
    {  
      "text": "текст сутності",  
      "type": "PERSON" | "ORG" | "LOC" | "DATE" | "EVENT" | "MISC",  
      "confidence": 0.0-1.0  
    }  
  ],  
  "keywords": [  
    {  
      "word": "ключове слово",  
      "relevance": 0.0-1.0  
    }  
  ],  
  "summary": "короткий саммарі тексту (2-3 речення)",  
  "language": {
```

```

"code": "uk" | "en" | "ru" | "other",
"name": "Ukrainian" | "English" | "Russian" | "Other",
"confidence": 0.0-1.0
},
"metadata": {
  "text_length": 123,
  "word_count": 45,
  "sentence_count": 3
}
}
"";

```

```

public String buildSystemPrompt() {
    return SYSTEM_PROMPT;
}

```

```

public String buildUserPrompt(String text) {
    return buildUserPrompt(text, null);
}

```

```

public String buildUserPrompt(String text, PromptOptions options) {
    if (options == null) {
        options = PromptOptions.getDefault();
    }
}

```

```

StringBuilder prompt = new StringBuilder();
prompt.append("Виконай комплексний аналіз наступного тексту:\n\n");
prompt.append("\"\"\"\"").append(text).append("\"\"\"\"\\n\\n");
if (options.isIncludeExamples()) {
    prompt.append(buildFewShotExamples());
}

prompt.append("Тепер проаналізуй текст вище та поверни результат ТІЛЬКИ у форматі JSON без жодних додаткових коментарів.\n");

```

```

if (options.getAdditionalInstructions() != null &&
    !options.getAdditionalInstructions().isEmpty()) {

```

```

        prompt.append("\nДодаткові інструкції:\n");
        prompt.append(options.getAdditionalInstructions()).append("\n");
    }

    return prompt.toString();
}

private String buildFewShotExamples() {
    return ""

```

ПРИКЛАДИ ПРАВИЛЬНОГО АНАЛІЗУ:

Приклад 1:

Текст: "Президент України Володимир Зеленський виступив на саміті в Брюсселі.

Це була надзвичайно важлива подія для нашої країни."

Відповідь:

```

{
  "sentiment": {
    "label": "positive",
    "confidence": 0.85,
    "probabilities": {
      "positive": 0.85,
      "negative": 0.05,
      "neutral": 0.10
    },
    "explanation": "Текст містить позитивні формулювання ('надзвичайно важлива') та описує значущу подію"
  },
  "entities": [
    {"text": "Володимир Зеленський", "type": "PERSON", "confidence": 0.99},
    {"text": "Україна", "type": "LOC", "confidence": 0.98},
    {"text": "Брюссель", "type": "LOC", "confidence": 0.97},
    {"text": "саміт", "type": "EVENT", "confidence": 0.90}
  ],
  "keywords": [

```

```

{"word": "президент", "relevance": 0.92},
{"word": "саміт", "relevance": 0.88},
{"word": "виступ", "relevance": 0.75}
],
"summary": "Президент України виступив на важливому міжнародному саміті в Брюсселі",
"language": {
  "code": "uk",
  "name": "Ukrainian",
  "confidence": 0.99
},
"metadata": {
  "text_length": 156,
  "word_count": 18,
  "sentence_count": 2
}
}

```

Приклад 2:

Текст: "Вчора сталася жахлива аварія на проспекті Перемоги.

Троє людей постраждали, один у критичному стані."

Відповідь:

```

{
  "sentiment": {
    "label": "negative",
    "confidence": 0.92,
    "probabilities": {
      "positive": 0.02,
      "negative": 0.92,
      "neutral": 0.06
    },
    "explanation": "Текст описує трагічну подію з використанням негативної лексики ('жахлива', 'критичний стан')",
  },
  "entities": [
    {"text": "проспект Перемоги", "type": "LOC", "confidence": 0.95},

```

```

        {"text": "вчора", "type": "DATE", "confidence": 0.90},
        {"text": "аварія", "type": "EVENT", "confidence": 0.88}
    ],
    "keywords": [
        {"word": "аварія", "relevance": 0.95},
        {"word": "постраждали", "relevance": 0.87},
        {"word": "критичний стан", "relevance": 0.85}
    ],
    "summary": "На проспекті Перемоги сталася серйозна аварія з постраждалими",
    "language": {
        "code": "uk",
        "name": "Ukrainian",
        "confidence": 0.98
    },
    "metadata": {
        "text_length": 114,
        "word_count": 15,
        "sentence_count": 2
    }
}

""";
}
}

package com.sentiment.analyzer.service;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.sentiment.analyzer.exception.ResponseParsingException;
import com.sentiment.analyzer.model.dto.response.*;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;

import java.util.*;

```

```

import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.stream.Collectors;

@Slf4j
@Service
@RequiredArgsConstructor
public class LLMResponseParser {

    private final ObjectMapper objectMapper;

    private static final Pattern JSON_PATTERN =
        Pattern.compile("\\[[\\s\\S]*\\]", Pattern.MULTILINE);

    private static final Pattern MARKDOWN_PATTERN =
        Pattern.compile("```(?:json)?\\s*([\\s\\S]*?)```", Pattern.MULTILINE);

    public ComprehensiveAnalysisResponse parse(String rawResponse) {
        log.debug("Parsing LLM response of length: {}", rawResponse.length());
        String cleanedJson = extractAndCleanJson(rawResponse);
        try {
            JsonNode rootNode = objectMapper.readTree(cleanedJson);
            return ComprehensiveAnalysisResponse.builder()
                .sentiment(parseSentiment(rootNode.get("sentiment")))
                .entities(parseEntities(rootNode.get("entities")))
                .keywords(parseKeywords(rootNode.get("keywords")))
                .summary(parseString(rootNode.get("summary")))
                .language(parseLanguage(rootNode.get("language")))
                .metadata(parseMetadata(rootNode.get("metadata")))
                .build();
        } catch (JsonProcessingException e) {
            log.error("Failed to parse JSON response: {}", cleanedJson, e);
            throw new ResponseParsingException(
                "Invalid JSON format in LLM response", e);
        }
    }

    private String extractAndCleanJson(String rawResponse) {
        String cleaned = rawResponse.trim();

        Matcher markdownMatcher = MARKDOWN_PATTERN.matcher(cleaned);
    }

```

```

    if (markdownMatcher.find()) {
        cleaned = markdownMatcher.group(1).trim();
        log.debug("Extracted JSON from markdown block");
    }

    Matcher jsonMatcher = JSON_PATTERN.matcher(cleaned);
    if (jsonMatcher.find()) {
        cleaned = jsonMatcher.group(0);
        log.debug("Extracted JSON object from response");
    }

    cleaned = cleaned
        .replaceAll("^^[^{}]*", "") // Remove text before first {
        .replaceAll("[^}]*$", "") // Remove text after last }
        .replace("\\n", "\n") // Fix escaped newlines
        .replace("\\\"", "\"") // Fix escaped quotes
        .trim();
    return cleaned;
}

private SentimentAnalysis parseSentiment(JsonNode node) {
    if (node == null || node.isNull()) {
        throw new ResponseParsingException("Missing sentiment field");
    }

    String label = parseString(node.get("label"));
    Double confidence = parseDouble(node.get("confidence"));
    Map<String, Double> probabilities = new HashMap<>();
    JsonNode probNode = node.get("probabilities");
    if (probNode != null && !probNode.isNull()) {
        probNode.fields().forEachRemaining(entry ->
            probabilities.put(entry.getKey(), entry.getValue().asDouble()));
    }

    String explanation = parseString(node.get("explanation"));
    return SentimentAnalysis.builder()
        .label(normalizeSentimentLabel(label))
        .confidence(confidence)
        .probabilities(probabilities)
        .explanation(explanation)

```

```

        .build();
    }

    private List<NamedEntity> parseEntities(JsonNode node) {
        if (node == null || node.isNull() || !node.isArray()) {
            return Collections.emptyList();
        }

        List<NamedEntity> entities = new ArrayList<>();
        node.forEach(entityNode -> {
            try {
                NamedEntity entity = NamedEntity.builder()
                    .text(parseString(entityNode.get("text")))
                    .type(parseString(entityNode.get("type")))
                    .confidence(parseDouble(entityNode.get("confidence")))
                    .build();
                entities.add(entity);
            } catch (Exception e) {
                log.warn("Failed to parse entity: {}", entityNode, e);
            }
        });
        return entities;
    }

    private List<Keyword> parseKeywords(JsonNode node) {
        if (node == null || node.isNull() || !node.isArray()) {
            return Collections.emptyList();
        }

        List<Keyword> keywords = new ArrayList<>();
        node.forEach(keywordNode -> {
            try {
                Keyword keyword = Keyword.builder()
                    .word(parseString(keywordNode.get("word")))
                    .relevance(parseDouble(keywordNode.get("relevance")))
                    .build();
                keywords.add(keyword);
            } catch (Exception e) {
                log.warn("Failed to parse keyword: {}", keywordNode, e);
            }
        });
    }

```

```

    }
});

keywords.sort((k1, k2) ->
    Double.compare(k2.getRelevance(), k1.getRelevance()));

return keywords;
}

private LanguageInfo parseLanguage(JsonNode node) {
    if (node == null || node.isNull()) {
        return null;
    }

    return LanguageInfo.builder()
        .code(parseString(node.get("code")))
        .name(parseString(node.get("name")))
        .confidence(parseDouble(node.get("confidence")))
        .build();
}

private AnalysisMetadata parseMetadata(JsonNode node) {
    if (node == null || node.isNull()) {
        return null;
    }

    return AnalysisMetadata.builder()
        .textLength(parseInteger(node.get("text_length")))
        .wordCount(parseInteger(node.get("word_count")))
        .sentenceCount(parseInteger(node.get("sentence_count")))
        .build();
}

private String parseString(JsonNode node) {
    if (node == null || node.isNull()) {
        return null;
    }

    return node.asText();
}

private Double parseDouble(JsonNode node) {

```

```
    if (node == null || node.isNull()) {  
        return null;  
    }  
    return node.asDouble();  
}
```

```
private Integer parseInteger(JsonNode node) {  
    if (node == null || node.isNull()) {  
        return null;  
    }  
    return node.asInt();  
}
```

```
private String normalizeSentimentLabel(String label) {  
    if (label == null) {  
        return "neutral";  
    }  
    String normalized = label.toLowerCase().trim();  
    switch (normalized) {  
        case "positive":  
        case "pos":  
        case "ПОЗИТИВНА":  
            return "positive";  
        case "negative":  
        case "neg":  
        case "НЕГАТИВНА":  
            return "negative";  
        case "neutral":  
        case "neu":  
        case "НЕЙТРАЛЬНА":  
            return "neutral";  
        default:  
            log.warn("Unknown sentiment label: {}, defaulting to neutral", label);  
            return "neutral";  
    }  
}
```

```

public boolean isValidJson(String response) {
    try {
        objectMapper.readTree(response);
        return true;
    } catch (JsonProcessingException e) {
        return false;
    }
}

public ComprehensiveAnalysisResponse parseWithFallback(
    String rawResponse, String fallbackSentiment) {

    try {
        return parse(rawResponse);
    } catch (ResponseParsingException e) {
        log.error("Failed to parse response, using fallback", e);

        return ComprehensiveAnalysisResponse.builder()
            .sentiment(SentimentAnalysis.builder()
                .label(fallbackSentiment != null ? fallbackSentiment : "neutral")
                .confidence(0.0)
                .explanation("Failed to parse LLM response")
                .build())
            .entities(Collections.emptyList())
            .keywords(Collections.emptyList())
            .build();
    }
}

package com.sentiment.analyzer.service;

import com.sentiment.analyzer.llm.LLMClient;
import com.sentiment.analyzer.llm.LLMClientFactory;
import com.sentiment.analyzer.model.dto.request.LLMAnalysisRequest;
import com.sentiment.analyzer.model.dto.request.ComprehensiveAnalysisRequest;

```

```
import com.sentiment.analyzer.model.dto.response.ComprehensiveAnalysisResponse;
import com.sentiment.analyzer.model.dto.response.LLMAnalysisResponse;
import com.sentiment.analyzer.prompt.PromptOptions;
import com.sentiment.analyzer.prompt.UniversalPromptBuilder;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;
```

```
@Slf4j
```

```
@Service
```

```
@RequiredArgsConstructor
```

```
public class ComprehensiveAnalysisService {
```

```
    private final LLMClientFactory clientFactory;
```

```
    private final UniversalPromptBuilder promptBuilder;
```

```
    private final LLMResponseParser responseParser;
```

```
    private final TextPreprocessingService preprocessingService;
```

```
    public ComprehensiveAnalysisResponse analyze(
```

```
        ComprehensiveAnalysisRequest request) {
```

```
        long startTime = System.currentTimeMillis();
```

```
        log.info("Starting comprehensive analysis for text of length: {}",
```

```
            request.getText().length());
```

```
        String cleanedText = preprocessingService.clean(request.getText());
```

```
        PromptOptions options = buildPromptOptions(request);
```

```
        String systemPrompt = promptBuilder.buildSystemPrompt();
```

```
        String userPrompt = promptBuilder.buildUserPrompt(cleanedText, options);
```

```
        LLMAnalysisRequest llmRequest = LLMAnalysisRequest.builder()
```

```
            .text(cleanedText)
```

```
            .systemPrompt(systemPrompt)
```

```
            .userPrompt(userPrompt)
```

```
            .temperature(0.3)
```

```

        .maxTokens(1500)
        .topP(0.9)
        .build();

LLMClient client = clientFactory.getClient();
LLMAnalysisResponse llmResponse = client.analyze(llmRequest);

ComprehensiveAnalysisResponse analysis =
    responseParser.parse(llmResponse.getRawResponse());

long processingTime = System.currentTimeMillis() - startTime;

log.info("Comprehensive analysis completed in {}ms. Sentiment: {}, Entities: {}, Keywords: {}",
        processingTime,
        analysis.getSentiment().getLabel(),
        analysis.getEntities().size(),
        analysis.getKeywords().size());

return analysis;
}

private PromptOptions buildPromptOptions(ComprehensiveAnalysisRequest request) {
    PromptOptions.PromptOptionsBuilder builder = PromptOptions.builder();

    if (request.getIncludeExamples() != null) {
        builder.includeExamples(request.getIncludeExamples());
    }

    if (request.getMaxKeywords() != null) {
        builder.maxKeywords(request.getMaxKeywords());
    }

    if (request.getMaxEntities() != null) {
        builder.maxEntities(request.getMaxEntities());
    }
}

```

```
if (request.getDetailLevel() != null) {  
    switch (request.getDetailLevel()) {  
        case "minimal":  
            return PromptOptions.getMinimal();  
        case "detailed":  
            return PromptOptions.getDetailed();  
        default:  
            return PromptOptions.getDefault();  
    }  
}  
  
return builder.build();  
}
```

ПРЕЗЕНТАЦІЯ

Магістерська кваліфікаційна робота на тему

СИСТЕМА АВТОМАТИЗОВАНОГО ЛІНГВІСТИЧНОГО АНАЛІЗУ ТЕКСТІВ НА ОСНОВІ ЗАСТОСУВАННЯ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ

Виконав: Подгорбунський Веніамін Володимирович
Науковий керівник: Чічур Андрій Іванович

Державний університет інформаційно-комунікаційних технологій
Київ – 2024

АКТУАЛЬНІСТЬ ДОСЛІДЖЕННЯ

Експоненціальне зростання обсягів текстових даних

Потреба в автоматизованій обробці

Революція великих мовних моделей (LLM)

Якісно новий рівень розуміння природної мови

Виклики сучасних систем:

- Висока вартість комерційних API
- Проблеми приватності даних
- Складність інтеграції

Необхідність простого та гнучкого рішення:

REST API + можливість вибору LLM

МЕТА РОБОТИ

Розробка системи автоматизованого лінгвістичного аналізу з можливістю використання хмарних (OpenRouter) або локальних LLM (Ollama)

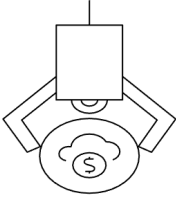
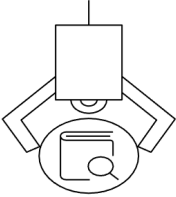
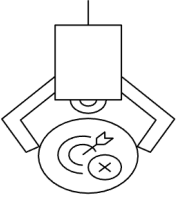
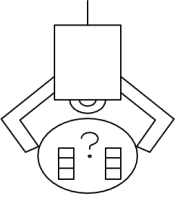
ЗАВДАННЯ ДОСЛІДЖЕННЯ

- Дослідити теоретичні основи NLP та архітектури LLM
- Проаналізувати існуючі рішення інтеграції з LLM
- Спроекувати REST API архітектуру системи
- Реалізувати систему на Java (Spring Boot)
- Розробити універсальний промпт для комплексного аналізу
- Провести тестування та порівняти OpenRouter vs Ollama

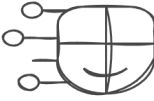
ПОРІВНЯЛЬНИЙ АНАЛІЗ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ

Модель	Параметри	Контекст	Якість	Вартість
GPT-4 Turbo	1.7T (est.)	128K	★★★★★	Висока
GPT-3.5	175B	16K	★★★★☆	Середня
Claude 3	Unknown	200K	★★★★★	Висока
LLaMA 2 70B	70B	4K	★★★★☆	Безкоштовно
Mistral 7B	7B	32K	★★★★☆	Безкоштовно

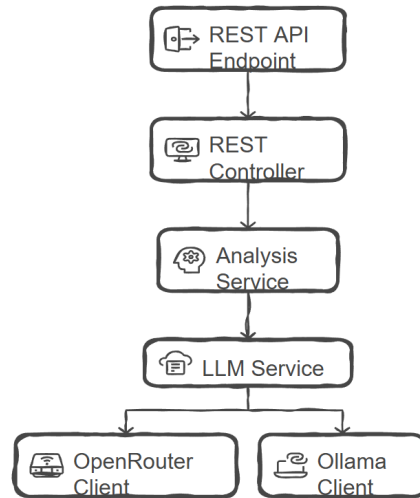
ПОРІВНЯННЯ ПІДХОДІВ ДО ЛІНГВІСТИЧНОГО АНАЛІЗУ

			
Комерційні API OpenAI API, Anthropic Claude, Google Cloud NLP, Azure Text Analytics, Amazon Comprehend. Висока якість, не потрібна інфраструктура, постійні оновлення.	Відкриті бібліотеки Hugging Face, spaCy, NLTK, Stanford CoreNLP. Безкоштовні, повний контроль, кастомізація.	Недоліки комерційних API Вартість, залежність від провайдера, приватність даних.	Недоліки відкритих бібліотек Складність налаштування, нижча якість, потрібна експертиза.

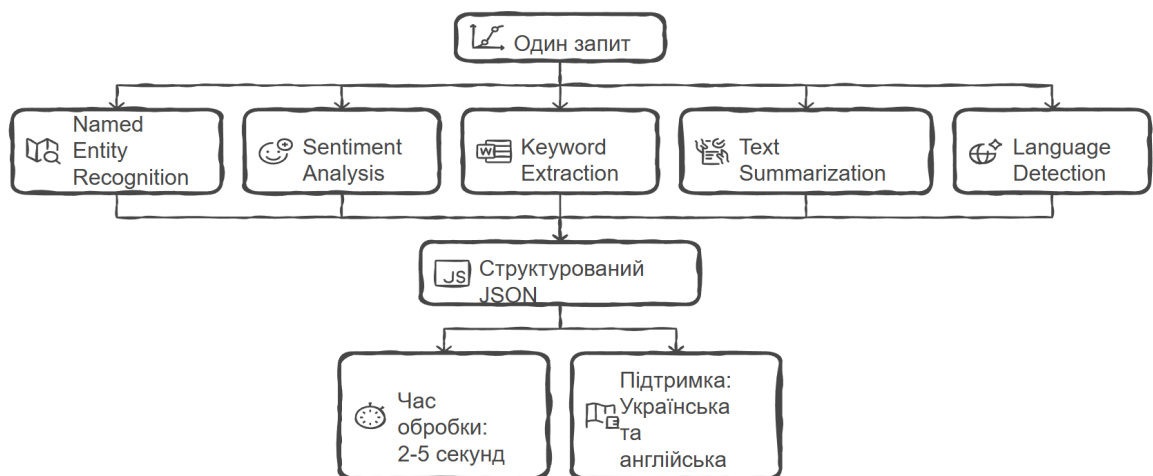
Стратегії інтеграції з LLM

	
OpenRouter Доступ до 100+ моделей через єдиний API. GPT-4, Claude, Mistral, LLaMA та інші. Простота: один endpoint для всіх. Прозоре ціноутворення. Automatic fallback між моделями.	Ollama Локальне розгортання LLM. LLaMA 2, Mistral, OpenChat. Повна приватність даних. Нульова вартість використання. Автономність (без інтернету).

Архітектура Розробленої Системи



Функції лінгвістичного аналізу



Універсальний промпт для аналізу

Характеристика	 Опис	 Формат
Інструкції	Чітке формулювання завдань	Специфікація формату виводу (JSON)
Few-Shot приклади	2-3 приклади для кожного завдання	Демонстрація бажаного формату
Текст для аналізу	Вхідний текст користувача	
Вимоги до виводу	Тільки JSON, без пояснень	Валідна структура

Точність аналізу тональності

	 OpenRouter	 Ollama	 Складність
Позитивні тексти	94.2%	87.5%	Легка
Негативні тексти	91.8%	85.3%	Легка
Нейтральні тексти	78.4%	71.2%	Середня
Саркастичні тексти	67.3%	52.1%	Висока
Змішана мова	83.6%	76.8%	Висока

Порівняння продуктивності OpenRouter та Ollama

	 OpenRouter (gpt-oss-20b)	 Ollama (Llama 3.2)
Середній час	2,847 мс	4,231 мс
Мінімум	1,523 мс	2,156 мс
Максимум	5,421 мс	8,734 мс
P50 (медіана)	2,698 мс	3,987 мс
P95	4,234 мс	7,123 мс
P99	4,987 мс	8,234 мс

ВИСНОВКИ

ДОСЯГНЕННЯ МЕТИ ТА ЗАВДАНЬ
Створено функціонуючу систему лінгвістичного аналізу з LLM
Реалізовано гібридний підхід: OpenRouter + Ollama
Розроблено REST API на Spring Boot з єдиним endpoint
Впроваджено комплексний аналіз за один запит

- МЕТРИКИ ЯКОСТІ
- Точність sentiment analysis: 89.7% (OpenRouter)
 - F1-score NER: 0.86
 - Час відгуку: 2.8 сек (хмара) / 4.2 сек (локально)

- НАУКОВА НОВИЗНА
- Гібридний підхід до лінгвістичного аналізу
 - Методика комплексного аналізу за один прохід
 - Уніфікований інтерфейс для різних LLM провайдерів

- ПРАКТИЧНА ЗНАЧУЩІСТЬ
- Ready-to-use рішення для інтеграції
 - Застосування: аналіз відгуків, моніторинг новин, обробка документації, лінгвістичні дослідження
 - Еталонна архітектура для подібних систем