НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В.о. завідувача кафедри
_____ Артем ВОЛОКИТА
(підпис)

“__” _____2026 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Система інтелектуального аналізу текстових даних із використанням методів обробки природної мови (NLP)

Виконав : студент __4__ курсу, групи _ІМ-21
(шифр групи)

_____Галай Андрій Вікторович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____ас., д-р філос. Трочун Є. В._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) ас., д-р філос. Коренко Д. В._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____доцент, к.т.н., доцент Шимкович В. М._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2026 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ ____Артем Волокита_
(підпис)

“__” _____2026 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Галая Андрія Вікторовича

1. Тема проєкту *Система інтелектуального аналізу текстових даних із використанням методів обробки природної мови (NLP)*
керівник проєкту асистент Трочун Євгеній Володимирович,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 3 червня 2026_ року _№2055-с_

2. Термін здачі студентом закінченого проєкту _2 червня 2026 р._____

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

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

Розділ 1. Огляд сучасних рішень.

Розділ 2. Огляд технологій та підходів у розробці системи.

Розділ 3. Реалізація головного функціоналу платформи.

Розділ 4. Огляд та тестування розробленої системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, функціональна схема (діаграма класів), принципова схема (блок-схема алгоритму).

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Коренко Д. В.		

7. Дата видачі завдання _____

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	17.11.2025-28.11.2025	
2.	*Вивчення та аналіз завдання*	01.12.2025-09.01.2026	
3.	*Розробка архітектури системи*	12.01.2026-23.01.2026	
4.	*Розробка структур окремих модулів системи*	02.03.2026-13.03.2026	
5.	*Програмна реалізація системи*	16.03.2026-17.04.2026	
6.	*Оформлення пояснювальної записки*	20.04.2026-22.05.2026	
7.	*Захист програмного продукту*	28.05.2026	
8.	*Передзахист*	29.05.2026	
9.	*Захист*	16.06.2026	

Студент-дипломник _____ Андрій ГАЛАЙ
(підпис)

Керівник проєкту _____ Євгеній ТРОЧУН
(підпис)

АНОТАЦІЯ

Дипломний проєкт присвячено розробці системи інтелектуального аналізу текстових даних із використанням методів обробки природної мови.

У роботі проаналізовано сучасні сервіси обробки природної мови та обґрунтовано потребу у відкритому рішенні, яке усуває ризики непрозорості обробки даних і залежності від постачальника послуг. Визначено поняття природної мови та наведено класифікацію текстових даних за їхньою структурою й обсягом. Описано процес розбиття тексту на токени та перетворення токенів на векторні представлення, що називаються ембедінгами, а також архітектури рекурентних нейронних мереж RNN та LSTM.

На підставі оглянутого матеріалу сформовано технологічний стек: бібліотека spaCy, фреймворк FastAPI, система керування базами даних PostgreSQL, сховище Redis та бібліотека React. Описано процес підготовки набору даних і генерації синтетичних даних для донавчання моделі класифікації спаму, а також тестування її роботи на незалежному наборі даних.

Реалізовано три модулі обробки природної мови: класифікатор спаму на основі донавченої моделі DistilBERT, модуль анонімізації персональних даних, що працює за допомогою регулярних виразів та розпізнавання іменованих сутностей, і модуль вилучення ключових слів. Серверну частину побудовано з розмежуванням двох категорій клієнтів за допомогою JWT-токенів та API-ключів, а клієнтський інтерфейс реалізовано як односторінковий застосунок, що побудований за допомогою бібліотеки React. Працездатність усіх компонентів підтверджено тестуванням, зокрема через реальні запити до запущеного сервісу. Текст програмного коду розміщено на платформі GitHub у відкритому доступі.

Ключові слова: обробка природної мови, класифікація спаму, анонімізація персональних даних, вилучення ключових слів, DistilBERT, FastAPI, API.

ANNOTATION

This bachelor's thesis addresses the development of a system for intelligent textual data analysis based on natural language processing techniques.

The work reviews existing natural language processing services and argues for an open alternative that mitigates the risks of opaque data handling and vendor lock-in. It defines the notion of natural language and classifies textual data according to its structure and volume. The fundamentals of text processing are then outlined, including tokenization, the transformation of tokens into vector representations known as embeddings, and the architectures of recurrent neural networks such as RNN and LSTM.

Drawing on this review, a technology stack was chosen, comprising the spaCy library, the FastAPI framework, the PostgreSQL database management system, Redis for in-memory storage, and the React library. The project further describes the preparation of the dataset and the generation of synthetic data for fine-tuning the spam classification model, as well as the evaluation of its performance on an independent dataset.

Three natural language processing modules were implemented: a spam classifier built on a fine-tuned DistilBERT model, a personal data anonymization module combining regular expressions with named-entity recognition, and a keyword extraction module. The back end distinguishes between two categories of clients through JWT tokens and API keys, while the front end is implemented as a single-page application developed with React. All components were validated through testing, including live requests to the deployed service. The source code is publicly available on GitHub.

Keywords: natural language processing, spam classification, personal data anonymization, keyword extraction, DistilBERT, FastAPI, API.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Система інтелектуального	4		
			аналізу текстових даних			
			із використанням методів			
			обробки природної мови			
			(NLP)			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Система інтелектуального	79		
			аналізу текстових даних			
			із використанням методів			
			обробки природної мови			
			(NLP)			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Структурна схема	1		
			системи			
	А4	***ІАЛЦ.467200.005 Д2***	Функціональна схема	1		
			(діаграма класів)			
	А4	***ІАЛЦ.467200.006 Д3***	Принципова схема	1		
			(блок-схема алгоритму)			
	А4	***ІАЛЦ.467200.007 Д4***	Текст програмного коду	1		

					ІАЛЦ.467200.001 ОА
Зм	*Лист*	*№ докум.*	*Підп*	*Дата*	
Розроб		Галай А.В.			*Система інтелектуального аналізу текстових даних із використанням методів обробки природної мови (NLP)* Опис альбому
Перев		Трочун Є. В.			

Літ.	Аркуш	Аркушів
	1	1

КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система інтелектуального аналізу текстових даних із використанням методів обробки природної мови (NLP)*»

Київ – 2026

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Це технічне завдання поширюється на розробку системи інтелектуального аналізу даних із використанням методів обробки природної мови, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування цієї системи є фільтрація спаму, знаходження ключових слів та анонімізація персональних даних у тексті, автоматизація обробки великої кількості неструктурованих текстових даних, інтеграція системи в сторонні сервіси через API.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою роботи є автоматизація типових задач обробки текстових даних, таких як фільтрація спаму, вилучення ключових слів та анонімізація персональних даних, шляхом створення програмної системи з відкритим вихідним кодом, придатної до розгортання на власних серверах, що забезпечує прозорість обробки даних.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки дипломного проекту є науково-технічна література, офіційні документації, публікації та статті в мережі Інтернет на дану тему.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Наявність функціоналу для визначення текстових даних, що є спамом, анонімізації персональних даних та вилучення ключових слів у текстових даних.
- Надати можливість реєстрації з метою отримання API ключа для використання системи.
- Обробляти вхідні запити асинхронно задля забезпечення стабільної роботи розробленої системи.
- Надати документацію щодо використання API системи.

5.2. Вимоги до програмного забезпечення

- ОС Linux, Mac чи Windows.
- Visual Studio Code версії 1.64 або вище.

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i3-2100T.
- ROM не менше ніж 64 ГБ.
- RAM не менше ніж 4 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	17.11.2025-28.11.2025
Вивчення та аналіз завдання	01.12.2025-09.01.2026
Розробка архітектури системи	12.01.2026-23.01.2026
Розробка структур окремих модулів системи	02.03.2026-13.03.2026
Програмна реалізація системи	16.03.2026-17.04.2026
Виправлення помилок	18.04.2026-19.05.2026
Оформлення пояснювальної записки	20.04.2026-22.05.2026

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система інтелектуального аналізу текстових даних із використанням методів обробки природної мови (NLP)*»

Київ – 2026

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

СКБД	Система керування базами даних
API	(Application Programming Interface) Прикладний програмний інтерфейс
DOM	(Document Object Model) Об'єктна модель документа
HTML	(HyperText Markup Language) Мова розмітки гіпертексту
HTTPS	(HyperText Transfer Protocol Secure) Захищений протокол передачі гіпертексту
IP	(Internet Protocol) Інтернет-протокол
JSON	(JavaScript Object Notation) Нотація об'єктів JavaScript
JWT	(JSON Web Token) Вебтокен JSON
LLM	(Large Language Model) Велика мовна модель
LSTM	(Long Short-Term Memory) Довга короткострокова пам'ять
NER	(Named Entity Recognition) Розпізнавання іменованих сутностей
ORM	(Object-Relational Mapping) Об'єктно-реляційне відображення
RNN	(Recurrent Neural Network) Рекурентна нейронна мережа
SHA	(Secure Hash Algorithm) Алгоритм безпечного хешування
SQL	(Structured Query Language) Мова структурованих запитів

ВСТУП

У сучасному світі кількість інформації, що генерується кожного дня, складно навіть уявити. Мільйони людей кожного дня пишуть електронні листи, відгуки, роблять фотографії, ставлять реакції у соціальних мережах тощо. Усе це – величезна кількість даних. За останні кілька років надзвичайно сильно розвинулися LLM, тому кількість текстових даних у світі почала рости ще в рази швидше. Оскільки текстових даних тепер генерується неймовірно багато, постала задача автоматизувати їхню обробку. Багато компаній та підприємств вже над цим працюють.

Розробляючи такі системи часто доводиться вирішувати подібні задачі. Наприклад, такими задачами може бути визначення чи є електронний лист спамом або ж вилучення ключових слів із тексту. Основна складність полягає не в розробці такої системи, а в її підтримці та модернізації, оскільки з’являються нові технології та методи роботи із текстовими даними. Саме тому було би зручно мати одну таку систему, яка би вміла виконувати потрібні задачі, а інші системи могли би з нею інтегруватися. Тоді буде потреба розробляти та підтримувати лише одну систему, яка добре виконуватиме потрібні задачі з обробки текстових даних.

Метою роботи є автоматизація типових задач обробки текстових даних, зокрема виявлення тексту, що є спамом, вилучення ключових слів із тексту, а також анонімізація персональних даних для унеможливлення ідентифікації особи чи осіб про яких йдеться у тексті.

Практичною цінністю даної роботи є розробка платформи зі зручним API та відкритим програмним кодом, що надасть можливість її інтеграції в сторонні сервіси для автоматизації обробки природної мови без необхідності займатися підтримкою подібної системи.

РОЗДІЛ 1

ОГЛЯД СУЧАСНИХ РІШЕНЬ

1.1 Визначення понять природної мови та текстових даних

Природна мова – це людська мова, що виникла й розвинулася природним шляхом у процесі спілкування, на відміну від штучних або формальних мов [1]. Людська мова може бути виражена у письмовій формі: написана від руки, друкована або існувати лише в електронному вигляді.

Будь-яка інформація, що представлена у вигляді послідовності символів є текстовими даними. Текстові дані у свою чергу поділяються на структуровані, напівструктуровані та неструктуровані текстові дані. Прикладом структурованих текстових даних є реляційні бази даних. Такі дані легше аналізувати та шукати в них закономірності, оскільки формат таких даних чітко описаний. До напівструктурованих текстових даних можна віднести файли у форматі JSON. У них зберігається певна структура даних, але водночас вона не є такою жорсткою як у баз даних, що дозволяє легко та швидко змінювати їхню схему видаляючи та додаючи нові поля чи теги. Неструктуровані текстові дані – це текстові дані, що не мають наперед визначеної структури. До неструктурованих текстових даних належить природна мова.

Текстові дані можна класифікувати за їхнім обсягом. Корпус тексту – це велика колекція текстів, що репрезентують певну мову. Корпус у свою чергу складається із документів, а документом може бути певна стаття, пост у соцмережі чи книга. Документи поділяються на абзаци та речення.

Аналіз неструктурованих текстових даних є найбільш складним. Він включає в себе багато етапів, а перший із них – розбиття корпусу на токени. Найпростіший технологічний підхід для токенізації текстових даних базується

на регулярних виразах. Для прикладу токенізація може бути розбиттям тексту на слова. У такому випадку токен – це слово. Це дуже передбачуваний та швидкий підхід, але він дуже жорсткий, оскільки йому бракує гнучкості. Із токенів формуються словники, а далі слова представляються у вигляді векторів для роботи з алгоритмами машинного навчання або для використання їх у глибокому навчанні.

На сьогодні популярним є використання глибокого навчання для аналізу неструктурованих текстових даних. Це можуть бути вузькоспеціалізовані моделі для виконання певних задач або велика мовна модель, де замість того щоб тренувати окрему модель для кожної задачі, тренується одна гігантська модель, що вміє виконувати безліч завдань.

1.2 Огляд сучасних рішень для вирішення задач аналізу природної мови

Швидка цифровізація та зростання обсягів текстових даних створюють попит на інструменти, що автоматизують їхню обробку. За останні роки було розроблено безліч платформ, що здатні ефективно вирішувати дуже різні задачі обробки природної мови та надають можливість інтеграції зі сторонніми сервісами.

Під час аналізу текстових даних досить часто потрібно використати комплексний підхід. Наприклад, системі модерації відгуків на маркетплейсі може бути потрібно фільтрувати коментарі із підозрілими посиланнями, рекламою конкурентів, замаскувати особисті дані, якщо користувач їх вказав, а також вилучити ключові слова, що потім будуть використовуватися для аналітики. Багато рішень орієнтовані лише на вирішення однієї проблеми, але також існує чимало платформ, що надають функціонал із вирішення багатьох

задач обробки природної мови таких як аналіз настроїв, аналіз іменованих сутностей, класифікації тексту тощо.

Незважаючи на велику кількість сервісів, вони всі мають закритий вихідний код та їх дуже рідко можна запускати на власних серверах. Неможливість переглянути вихідний код створює певні сумніви та ризики щодо того як дані користувачів сервісу можуть використовуватися.

У даному підрозділі проведено аналіз та порівняння сервісів, що вирішують задачі обробки природної мови. Також наведено їхні переваги та недоліки, що дозволять обґрунтувати доцільність створення платформи із відкритим вихідним кодом, що буде здатна виконувати різні задачі аналізу текстових даних.

1.2.1 Google Natural Language API

Google Natural Language API надає розробникам певний набір інструментів для обробки текстових даних. Дана система може виконувати аналіз настроїв, аналіз сутностей, аналіз настроїв сутностей, класифікацію контенту та синтаксичний аналіз [2]. Також існує детальна та зрозуміла документація із поясненнями як користуватися сервісом. На офіційному сайті є доступною можливість безкоштовно випробувати роботу API. На рисунку 1.1 зображено приклад того, як працює один із методів Google Natural Language API.

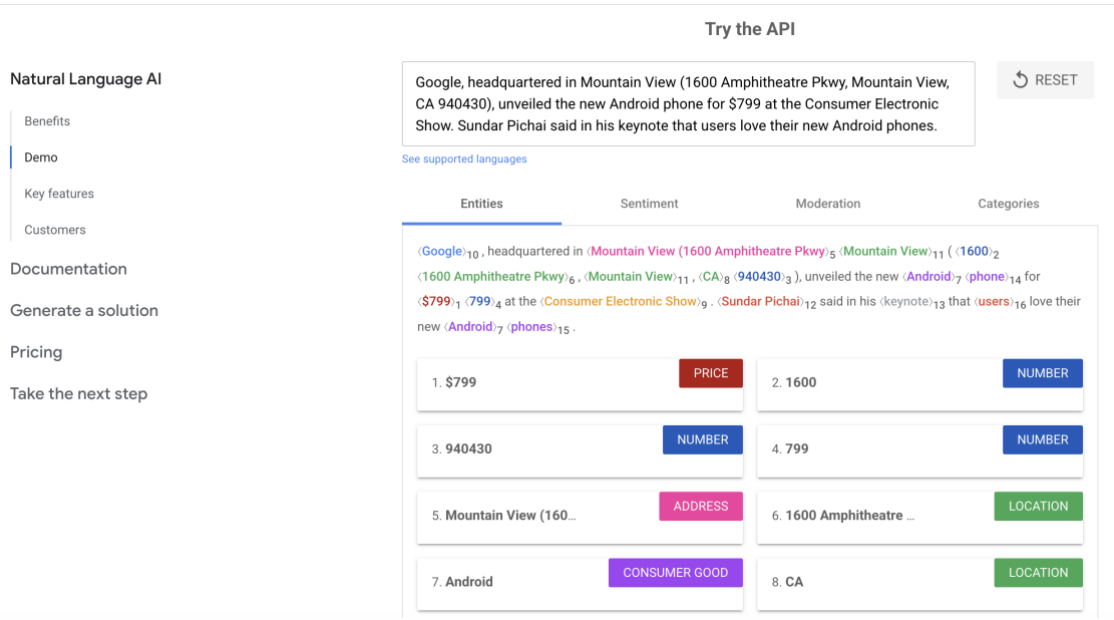
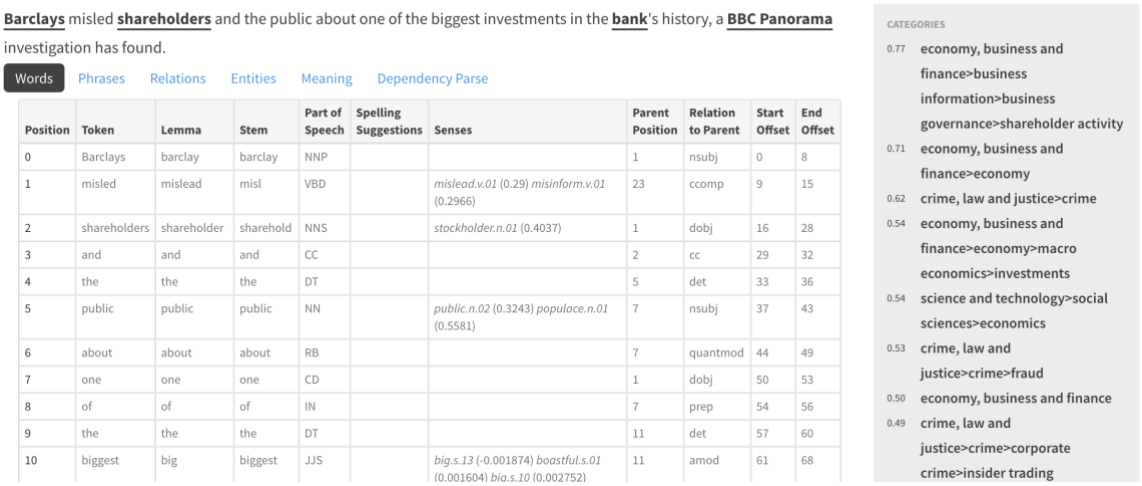


Рис. 1.1 – приклад результату роботи Google Natural Language API

Даний сервіс підтримує багато поширених мов таких як англійська, китайська, німецька тощо. Вихідний код знаходиться у закритому доступі, тому можливості побачити як відбувається процес обробки даних немає. Використовується модель оплати pay-as-you-go, що дає можливість платити лише за фактично використані ресурси. Доступний план, що дозволяє використовувати сервіс безкоштовно із певним лімітом токенів.

1.2.2 TextRazor

TextRazor – це ще один хмарний сервіс для роботи із неструктурованими текстовими даними. Основними його можливостями є знаходження сутностей та ключових фраз, автоматичне визначення теми, класифікація, аналіз структури речення та зв’язків між словами у реченні [3]. Дана платформа має один API метод для обробки текстових даних, а вказати, що потрібно розробнику можна через параметр запиту. Такий підхід може бути незручним, коли потрібно дуже детально проаналізувати текст, оскільки відповідь від сервера стає досить великою. Також при збільшенні обсягу тексту сервіс починає працювати помітно повільніше.



Barclays misled **shareholders** and the public about one of the biggest investments in the **bank**'s history, a **BBC Panorama** investigation has found.

Words | Phrases | Relations | Entities | Meaning | Dependency Parse

Position	Token	Lemma	Stem	Part of Speech	Spelling Suggestions	Senses	Parent Position	Relation to Parent	Start Offset	End Offset
0	Barclays	barclay	barclay	NNP			1	nsubj	0	8
1	misled	mislead	misl	VBD		*mislead.v.01* (0.29) *misinform.v.01* (0.2966)	23	ccomp	9	15
2	shareholders	shareholder	sharehold	NNS		*stockholder.n.01* (0.4037)	1	dobj	16	28
3	and	and	and	CC			2	cc	29	32
4	the	the	the	DT			5	det	33	36
5	public	public	public	NN		*public.n.02* (0.3243) *populace.n.01* (0.5581)	7	nsubj	37	43
6	about	about	about	RB			7	quantmod	44	49
7	one	one	one	CD			1	dobj	50	53
8	of	of	of	IN			7	prep	54	56
9	the	the	the	DT			11	det	57	60
10	biggest	big	biggest	JJS		*big.s.13* (-0.001874) *boastful.s.01* (0.001604) *bia.s.10* (0.002752)	11	amod	61	68

CATEGORIES

0.77 economy, business and finance>business information>business governance>shareholder activity
0.71 economy, business and finance>economy
0.62 crime, law and justice>crime
0.54 economy, business and finance>economy>macro economics>investments
0.54 science and technology>social sciences>economics
0.53 crime, law and justice>crime>fraud
0.50 economy, business and finance
0.49 crime, law and justice>crime>corporate crime>insider trading

Рис. 1.2 – використання TextRazor для аналізу слів у реченні

TextRazor підтримує 19 мов, серед них також є українська. Модель монетизації сервісу базується на фіксованій ціні й кількості запитів на місяць, що може не підійти тим, хто хоче користуватися сервісом не на всю потужність. Платформа має безкоштовний план, що дозволяє робити до 500 запитів на місяць. Вихідного коду немає у відкритому доступі, що також не дає побачити як саме обробляються дані.

1.2.3 Lexalytics

Ще одним прикладом схожої на попередні платформи є Lexalytics. Ця система здатна виконувати детальний аналіз настроїв розбиваючи текст на сутності та даючи оцінку настрою кожній із них, класифікувати текст, визначати тему, робити сумаризацію [4]. Lexalytics добре справляється із навантаженням та швидко дає відповіді на запити, має підтримку кількох десятків мов. Платформа має зручний API та добре справляється із великими наборами даних. Модель оплати залишається невідомою, оскільки на офіційному вебсайті про це немає даних. Єдиний спосіб дізнатися ціну за використання сервісу – звернутися до їхнього відділу продажів, що не є зручно та займає час. Оскільки для початку використання сервісу потрібно зв’язатися

із відділом продажів, стартапи та бізнеси, що рухаються в швидкому темпі, можуть навіть не розглядати Lexalytics через це.

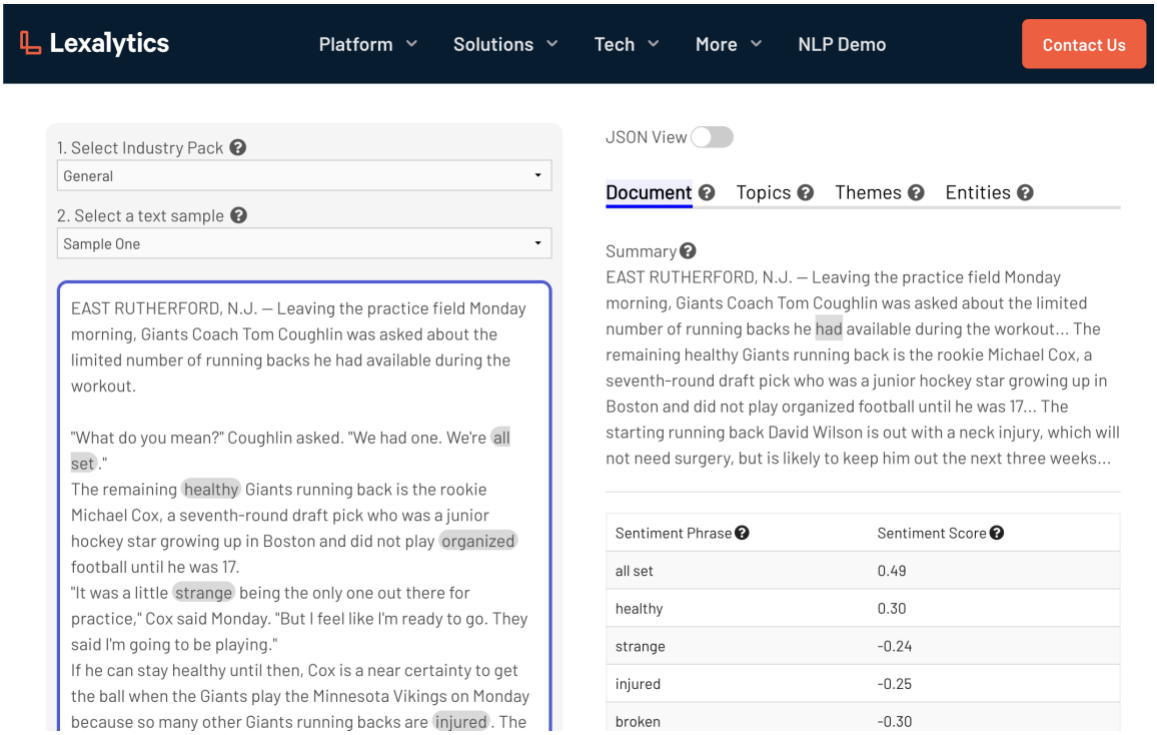


Рис. 1.3 – використання Lexalytics для аналізу текстових даних

На вебсайті присутня детальна документація із достатньою кількістю прикладів, що полегшує роботу із API. Вихідного коду даної платформи також немає у відкритому доступі, тому як і в попередніх випадках побачити, що саме відбувається із даними та як вони обробляються немає.

1.3 Порівняння

Для порівняння раніше згаданих сервісів було складено порівняльну таблицю. Зіставлення проведено за критеріями, що є найбільш суттєвими для потенційного користувача: наявність безкоштовного плану, модель оплати, якість документації, цільова аудиторія та відкритість вихідного коду. Такий набір критеріїв дозволяє оцінити сервіси не лише за технічними можливостями, а й за зручністю впровадження та супроводу.

Таблиця 1.1 – порівняльна характеристика сервісів, що використовуються для аналізу неструктурованих текстових даних

Критерій	Google Natural Language API	TextRazor	Lexalytics
Наявність безкоштовного плану	Так	Так	Ні
Модель оплати	Базується на кількості використаних токенів	Щомісячна підписка	Вирішується під час розмови із відділом продажів
Документація	Чітко описана та має багато прикладів	Лаконічна, мінімум теорії, достатньо прикладів	Детальна, має пояснення основних понять та того, як вирішуються задачі обробки текстових даних
Цільова аудиторія	Масовий ринок, розробники вебсайтів та мобільних застосунків	Розробники та аналітики даних	Великі корпорації
Вихідний код у відкритому доступі	Ні	Ні	Ні

Із таблиці видно, що всі сервіси, про які йшла мова, мають закритий вихідний код. Це означає, що побачити як вони обробляють текстові дані не можна, що додає певних сумнівів щодо конфіденційності даних, оскільки існує

ймовірність того, що дані платформи можуть збирати дані користувачів для подальшого навчання своїх моделей. Lexalytics має on-premises рішення, яке можна розгортати на власних серверах, але ціна на його використання може бути досить високою.

Найбільш оптимальним рішенням є Google Natural Language API. Даний сервіс має безкоштовний план, прозору модель оплати, що дозволяє не переплачувати, хорошу документацію та зручний API. Проте як і у випадку із іншими платформами, вихідний код є у закритому доступі, тому компанії для яких конфіденційність даних є критичною можуть мати сумніви щодо використання цього сервісу.

Отже, існує потреба в розробці сучасного безкоштовного рішення із відкритим вихідним кодом, що можна модифікувати під свої потреби та розгортати на власних серверах, щоб забезпечити максимальний захист для конфіденційної інформації, усунути ризик Vendor Lock-in і знизити витрати.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі було розглянуто базові поняття, що стосуються природної мови та текстових даних, а також проведено огляд сучасних рішень, які використовуються для їхньої автоматизованої обробки. Природну мову складно аналізувати, це зумовлено відсутністю наперед визначеної структури, варіативністю формулювань, наявністю синонімів та залежності значення слів від контексту. Як наслідок, обробка природної мови потребує послідовного виконання певних кроків – від токенізації та формування словників до побудови векторних представлень слів і застосування алгоритмів машинного й глибокого навчання.

Аналіз наявних платформ показав, що ринок інструментів для обробки природної мови є насиченим, але далеко не ідеальним. Кожне з розглянутих рішень має власні сильні сторони: різноманітні моделі оплати, якісна документація, підтримка великої кількості мов, якісні результати аналізу даних. Водночас усі без винятку розглянуті сервіси мають певні недоліки.

Ключовим недоліком наявних на ринку рішень є закритість вихідного коду. Користувач не має змоги побачити, як саме оброблюються його дані, які алгоритми застосовуються та чи не використовується надісланий текст для подальшого тренування внутрішніх моделей постачальника послуг. Для організацій, що працюють із чутливою інформацією – персональними даними клієнтів, медичними записами, комерційною таємницею чи внутрішніми документами – подібна непрозорість є серйозною перешкодою для впровадження таких сервісів.

Не менш важливою проблемою залишається залежність від постачальника, відома як Vendor Lock-in. Побудувавши свою інфраструктуру навколо конкретного API, компанія стикається з ризиком підвищення цін, зміни умов використання чи навіть припинення роботи сервісу. Перехід на іншу

платформу у такій ситуації потребує значних витрат часу та ресурсів на переписування інтеграцій.

Питання вартості використання залишається відкритим для багатьох категорій користувачів. Моделі оплати, побудовані на місячній підписці з фіксованою кількістю запитів, є невигідними для тих, хто має нерівномірне навантаження. Підхід Lexalytics, за якого ціна формується індивідуально після спілкування з відділом продажів, фактично відсікає невеликі компанії, стартапи та індивідуальних розробників, які не готові витрачати тижні на узгодження умов.

Сукупність наведених недоліків формує чітко окреслену потребу у створенні альтернативного рішення. Така система повинна мати відкритий вихідний код, що забезпечить прозорість обробки даних і можливість незалежного аудиту безпеки. Вона має бути придатною до розгортання на власній інфраструктурі, що гарантує конфіденційність текстової інформації та усуває ризик передачі даних третім сторонам. Архітектура повинна дозволяти модифікацію під потреби конкретного проєкту – від додавання нових моделей машинного навчання до зміни логіки обробки. Зрештою, така платформа може бути безкоштовною.

Таким чином, проведений огляд підтверджує актуальність теми роботи та обґрунтовує необхідність розробки власної системи інтелектуального аналізу текстових даних. Подальші розділи присвячено вибору технологічного стеку, проєктуванню архітектури та реалізації такої системи.

РОЗДІЛ 2

ОГЛЯД ТЕХНОЛОГІЙ ТА ПІДХОДІВ У РОЗРОБЦІ СИСТЕМИ

2.1 Теоретичні основи обробки природної мови

2.1.1 Токенізація тексту

Комп'ютери та нейромережі добре працюють із числами, але не вміють напряму опрацьовувати текст у тому вигляді, у якому його сприймає людина. Щоб модель могла проаналізувати текст, його спершу треба перетворити на числові послідовності, а перший крок на цьому шляху – токенізація. Токенізація – це процес розбиття корпусу тексту на менші одиниці, що називаються токенами. Це початковий етап обробки будь-яких текстових даних, і саме від нього значною мірою залежить якість усіх наступних кроків аналізу.

Існує чимало способів поділити текст на токени. Насамперед варто зазначити, що той, хто виконує токенізацію, може сам визначити правило, за яким текст ділиться на одиниці. Наприклад, токеном може бути ціле речення – тоді корпус розбивається на речення. Так само за токен можна взяти слово, частину слова або окремий символ і відповідно до цього ділити текст.

Під час токенізації кожному токену присвоюється унікальний ідентифікатор, а множина всіх унікальних токенів разом із їхніми ідентифікаторами утворює словник. До словника можна додавати й власноруч визначені токени. Наприклад, можна створити окремий токен для позначення IP-адреси, який використовуватиметься щоразу, коли в тексті трапляється IP-адреса. Надалі токени зі словника використовуються для побудови ембедінгів.

Підходи до токенізації зручно розрізняти за тим, як саме формується словник і на яких одиницях працює модель. За цією ознакою виділяють токенізацію на основі правил (rule-based) та статистичну.

Rule-based токенізація розбиває текст на токени за наперед заданими шаблонами, які зазвичай описують регулярними виразами. Її головна перевага – простота й передбачуваність: за однакових вхідних даних результат завжди однаковий, а сам процес не потребує попереднього навчання на корпусі. Такий метод добре підходить для нескладної токенізації, проте зі зростанням кількості правил його складність швидко збільшується. У межах цього підходу іноді окремо розглядають токенізацію за цілими словами, але вона має суттєві недоліки: словник виходить надмірно великим, а якщо модель зустріне слово, якого не було серед відомих, вона не зможе його опрацювати й втратить частину контексту.

Статистична токенізація, на відміну від попередньої, не покладається на чіткі правила. Замість цього вона враховує, як часто в корпусі трапляються окремі символи та їхні комбінації, і додає до словника лише ті послідовності, що зустрічаються найчастіше. Завдяки цьому словник формується автоматично на основі аналізу великого корпусу, а не задається вручну. Такий метод значно гнучкіший, але теж вимагає підготовки великих обсягів тексту для побудови словника. Варто враховувати, що ці алгоритми працюють зі словами, тож текст спершу проходить нормалізацію та попередню токенізацію.

Нормалізація може включати приведення символів до одного регістру, видалення HTML-тегів, зайвих пробілів тощо. Попередня токенізація – це розбиття тексту на одиниці більші за окремі символи, найчастіше на слова за пробілами та знаками пунктуації.

Найпоширенішим прикладом статистичного підходу є Byte Pair Encoding. Спочатку кожне слово розбивається на окремі символи, після чого алгоритм багаторазово знаходить найчастішу пару сусідніх токенів і об'єднує її в один новий токен. Цей процес триває, доки словник не досягне наперед заданого

розміру [5]. Завдяки такому підходу часто вживані слова з часом стають окремими токенами, тоді як рідкісні та складні слова залишаються розбитими на менші частини. Це дає важливу перевагу над токенізацією за словами: навіть якщо модель зустріне незнайоме слово, вона зможе скласти його з уже відомих частин і не втратить контекст повністю, а розмір словника при цьому лишається помірним і контрольованим. Серед інших відомих алгоритмів цього напрямку варто згадати WordPiece та Unigram – вони працюють за схожою ідеєю, але використовують різні критерії для відбору токенів до словника. Саме підслівна токенізація на сьогодні є стандартом для більшості сучасних мовних моделей [6].

2.1.2 Векторні представлення слів

Після того, як текст розбито на токени та кожному з них присвоєно унікальний числовий ідентифікатор, виникає нова проблема. Звичайні ідентифікатори не несуть у собі жодного змісту. Якщо, наприклад, словам «король» і «королева» присвоєно номери 15 і 213, то різниця між цими числами жодним чином не відображає їхню смислову близькість. Нейромережі ж необхідно розуміти семантику слів, їхній контекст та взаємозв'язки. Саме для вирішення цієї задачі застосовуються ембедінги.

Ембедінг – це перетворення токена у вектор дійсних чисел фіксованої довжини. Замість одного ізольованого числа, токен описується масивом значень у багатовимірному просторі. Кількість таких вимірів може становити від кількох десятків до кількох тисяч, що безпосередньо залежить від обраної моделі. Кожен вимір відповідає за певну приховану характеристику слова, хоча під час роботи алгоритму ці ознаки зазвичай неможливо чітко інтерпретувати людською мовою.

Головна перевага ембедінгів полягає в їхній здатності враховувати семантичне значення слова. У згаданому багатовимірному просторі токени, які мають схожий зміст або часто вживаються в однаковому контексті, будуть розташовані під невеликим кутом один до одного. У такому випадку схожість між векторами можна виміряти за допомогою косинусної подібності [7]. Ця метрика дає уявлення про те, наскільки схожі за значенням є токени. Відповідно, косинусна подібність між векторами слів «король» та «королева» буде значно більшою, ніж між векторами «кіт» та «автомобіль». Це дає змогу моделям не просто розрізняти вектори, а й вимірювати ступінь їхньої спорідненості. Косинусна подібність між двома векторами обчислюється за формулою:

$$cosine\_sim(A, B) = \cos(\theta) = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Де A та B – вектори, θ – кут між векторами, n – розмірність векторів [8].

Під час роботи з векторними представленнями також варто розрізняти статичні та контекстуальні ембедінги. У класичних підходах кожному слову зі словника завжди відповідає один і той самий незмінний вектор, незалежно від речення, у якому воно знаходиться. Проте в природній мові слова можуть бути багатозначними. Тому в сучасних архітектурах генеруються контекстуальні ембедінги. Під час їх формування враховується не лише сам токен, але і його сусідні слова у тексті. Завдяки цьому слово «замок» у контексті середньовічної фортеці та у контексті дверного механізму буде описане абсолютно різними числовими векторами, що є критично важливим для точного аналізу тексту.

Відомим методом для перетворення слів на статичні вектори, що використовує нейронну мережу є Word2Vec [9]. У цього методу є дві архітектури:

- Continuous bag-of-words
- Skip-Gram

Ідеєю першої архітектури є те, що модель на вхід отримує слова, що формують контекст, а на виході модель повинна визначити слово, що було пропущено в середині тексту.

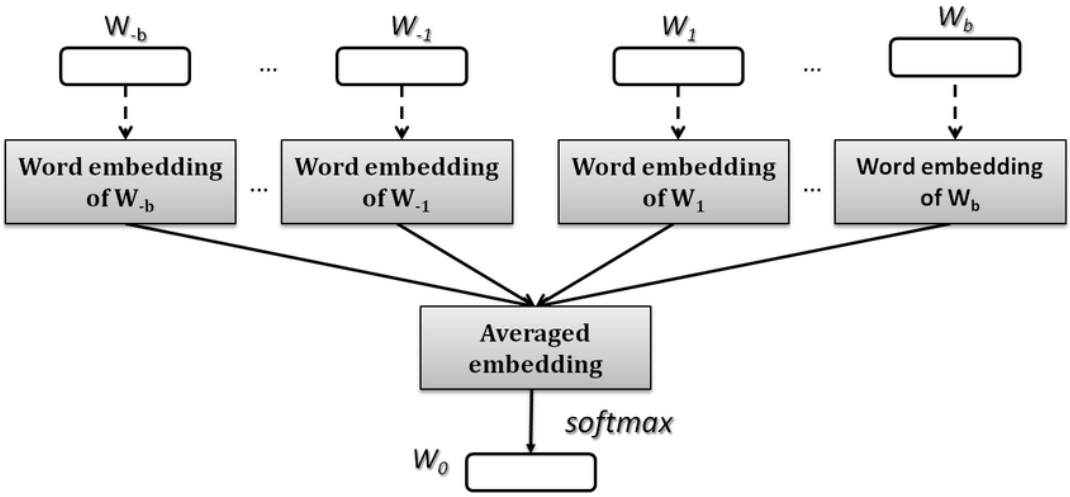
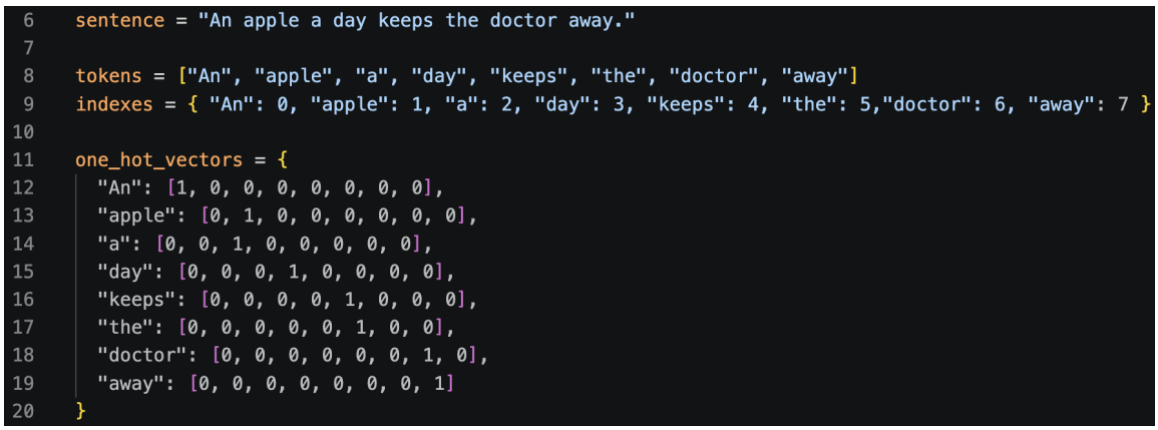


Рис. 2.1 – Огляд моделі Continuous bag-of-words [10]

На рисунку спрощено зображено принцип роботи першої архітектури. Спочатку обирається слово, яке модель має передбачити. На вхід моделі передаються слова, які оточують цільове слово у тексті. На рисунку W означає слово, b – індекс цільового слова у тексті. Тому W_{-1} та W_1 є словами в тексті, що йдуть одразу перед та після слова, яке модель намагається передбачити. Варто зазначити, що на вхід слова передаються не як текст, а у вигляді one-hot векторів. Для того щоб отримати one-hot вектор потрібно надати кожному токену чисельний індекс, а потім для кожного токену зробити вектор, розмір якого дорівнюватиме словнику, отриманому під час токенізації. Далі для кожного слова потрібно заповнити вектор нулями за всіма індексами, окрім того індекса, що дорівнює індексу слова, що був наданий йому раніше. Значення вектору за індексом який дорівнює індексу слова, потрібно заповнити одиницею. На рисунку 2.2 зображено приклад перетворення токенів у реченні на one-hot вектори.



```
sentence = "An apple a day keeps the doctor away."

tokens = ["An", "apple", "a", "day", "keeps", "the", "doctor", "away"]
indexes = { "An": 0, "apple": 1, "a": 2, "day": 3, "keeps": 4, "the": 5,"doctor": 6, "away": 7 }

one_hot_vectors = {
  "An": [1, 0, 0, 0, 0, 0, 0, 0],
  "apple": [0, 1, 0, 0, 0, 0, 0, 0],
  "a": [0, 0, 1, 0, 0, 0, 0, 0],
  "day": [0, 0, 0, 1, 0, 0, 0, 0],
  "keeps": [0, 0, 0, 0, 1, 0, 0, 0],
  "the": [0, 0, 0, 0, 0, 1, 0, 0],
  "doctor": [0, 0, 0, 0, 0, 0, 1, 0],
  "away": [0, 0, 0, 0, 0, 0, 0, 1]
}
```

Рис. 2.2 – приклад представлення токенів за допомогою one-hot векторів

Як видно з рисунку, спочатку речення розбивається на токени, кожному токену дається індекс, а потім для кожного токену формується вектор, де за індексом токену стоїть значення «1», решта значень вектора заповнена нулями.

Наступним кроком one-hot вектори множаться на матрицю ваг, формуючи ембедінги. Далі сформовані вектори потрапляють до прихованого шару, де вони усереднюються та формується узагальнений вектор. Потім цей вектор передається до вихідного шару, де за допомогою активаційної функції обчислюється ймовірності слів, які можуть стояти на місці пропущеного слова.

Таким чином після кількох епох, мінімізації функції втрат та корегування матриці ваг отримуються найбільш точні ембедінги, що потім використовуються нейромережею для аналізу текстових даних. Skip-Gram працює за зворотнім принципом: на вхід дається певне слово, а на виході модель має передбачити слова, що стоять поруч із вхідним словом [11].

Як було сказано раніше, недоліком статичних ембедінгів є те, що вони будуть однакові для слів незалежно від того, яке саме значення має слово у реченні. Це є проблемою, оскільки в природній мові часто трапляються випадки, коли одне й те саме слово може мати різне значення залежно від контексту.

2.1.3 Рекурентні нейронні мережі

Для врахування контексту певний час використовувалися рекурентні нейронні мережі. Вони відрізняються від звичайних нейронних мереж тим, що мають прихований стан, який працює як короткострокова пам’ять. Рекурентна мережа читає текст зліва на право та на кожному кроці отримує вектор поточного слова та вектор «пам’яті» із попередніх кроків. Базуючись на цих двох значеннях мережа обчислює поточне значення, яке буде також враховано при обчислюванні значення на наступному кроці. Стану поточного прихованого шару рахується за формулою:

$$\boldsymbol{h_t = \sigma(U \cdot X_t + W \cdot h_{t-1} + B)}$$

Де h_t – новий прихований стан (вектор пам'яті) на поточному кроці t, σ - функція активації (найчастіше це гіперболічний тангенс $tanh$ або $ReLU$), яка додає системі нелінійності, U - матриця ваг для вхідних даних, X_t – вхідний вектор (ембедінг поточного слова) на кроці t, W – матриця ваг для попереднього прихованого стану, h_{t-1} – прихований стан з попереднього кроку $t-1$, B – баяс [12].

Такий підхід дозволив моделям «пам'ятати» попередні слова у реченні, що стало значним кроком уперед порівняно зі статичними ембедінгами, які повністю ігнорували порядок слів. Завдяки рекурентним зв'язкам мережа здатна генерувати базові контекстуальні вектори, де значення слова залежить від його попередників.

Проте, класичні рекурентні нейронні мережі мають суттєвий недолік при роботі з довгими текстами – проблему затухання та вибуху градієнта (vanishing and exploding gradient problem). Під час навчання мережі за допомогою алгоритму зворотного поширення помилки в часі, градієнти, що проходять через багато часових кроків, постійно перемножуються. Це призводить до того,

що вони можуть ставати або експоненційно великими (вибух), або наближатися до нуля (затухання).

На практиці затухання градієнта означає, що ваги для перших слів у довгому реченні майже не оновлюються. Через це класична RNN втрачає здатність встановлювати довгострокові залежності – вона добре пам'ятає останні кілька слів, але "забуває" контекст початку довгого абзацу чи речення. Для вирішення цієї проблеми були розроблені більш складні модифікації рекурентних мереж, такі як **LSTM**.

2.1.4 LSTM, encoder-decoder та transformer моделі

Для розв'язання цієї проблеми затухання та вибуху градієнта в RNN було запропоновано архітектуру LSTM (мережі довгої короткострокової пам'яті). Головною інновацією LSTM стала радикальна зміна внутрішньої структури рекурентного вузла (комірки). Замість простого застосування однієї функції активації до вхідних даних, комірка LSTM використовує внутрішній стан осередку (cell state) та систему спеціальних вентилів (gates), які динамічно керують потоками інформації.

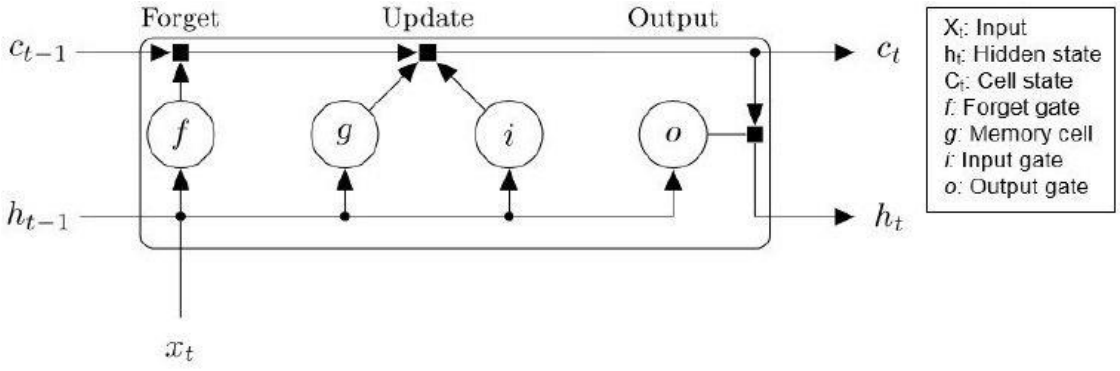


Рис. 2.3 – Архітектура вузла LSTM [13]

Стан осередку діє як магістральний конвеєр пам'яті, що проходить крізь увесь ланцюжок обробки тексту. Завдяки лінійним операціям над ним, градієнти під час зворотного поширення помилки можуть вільно протікати

через багато кроків назад, не затухаючи. Додавання, модифікація або видалення інформації з цієї пам'яті регулюється трьома типами вентилів, кожен з яких є шаром нейромережі з сигмоїдальною функцією активації:

1. Вентиль забування (Forget Gate): Аналізує поточне вхідне слово та попередній стан мережі, після чого приймає рішення, яку частину накопиченої раніше інформації слід стерти з пам'яті як нерелевантну.
2. Вентиль введення (Input Gate): Визначає, які нові семантичні ознаки з поточного слова є достатньо важливими, щоб їх варто було записати до стану осередку (внутрішньої пам'яті).
3. Вентиль виведення (Output Gate): Фільтрує оновлену інформацію з пам'яті осередку та формує поточний прихований стан (вихідний контекстуальний вектор слова), який передається на наступний крок обробки та на вищі шари мережі.

Завдяки такій структурі моделі на базі LSTM демонструють високу ефективність у задачах інтелектуального аналізу тексту, оскільки вони здатні успішно вловлювати складні довгострокові залежності у реченнях та абзацах.

У складніших завданнях обробки природної мови, де вхідна та вихідна текстові послідовності мають різну довжину (наприклад, у системах генерації відповідей або машинного перекладу), класичного використання LSTM виявилося недостатньо. Це зумовило появу архітектурного шаблону кодувальника-декодувальника (моделей Sequence-to-Sequence).

У межах цієї парадигми одна рекурентна мережа (кодувальник) послідовно стискає весь вхідний текст у єдиний вектор контексту фіксованого розміру. Друга мережа (декодувальник) приймає цей єдиний вектор і крок за кроком генерує вихідну послідовність слів. Основним обмеженням такого підходу стала проблема того, що спроба вмістити зміст великого речення чи абзацу в один статичний вектор неминуче призводила до втрати дрібних семантичних деталей тексту [14].

Для усунення цієї проблеми кодувальників-декодувальників було розроблено механізм уваги (Attention Mechanism), який згодом еволюціонував у повноцінну архітектуру Transformer. Ключовою відмінністю Трансформерів стала повна відмова від рекурентної обробки даних на користь механізму внутрішньої уваги (Self-Attention).

Замість послідовного аналізу слова за словом, Трансформер обробляє весь текст паралельно, дозволяючи кожному слову в реченні миттєво взаємодіяти з усіма іншими словами для обчислення ступеня їхньої взаємопов'язаності. Це забезпечує дві головні переваги: формування максимально точних двонаправлених контекстуальних ембедінгів та можливість ефективного розпаралелювання обчислень на сучасних графічних процесорах. Саме паралельна обробка та архітектура Transformer стали технологічним фундаментом для створення сучасних великих мовних моделей, що демонструють найвищу точність у задачах інтелектуального аналізу текстових даних [15].

2.1.5 Претреновані моделі та донавчання (fine-tuning)

Тренування моделі з нуля для вирішення певної прикладної задачі є досить складним та довгим процесом. У світі існує чимало архітектур нейронних мереж для вирішення задач обробки тексту, зображень тощо. Тому дуже зручно взяти використати готову архітектуру для розв’язання проблеми. Але також можна не обмежуватися використанням вже створеної архітектури, а взяти модель, що була попередньо навчена. Для задач обробки текстових даних це можна порівняти із тим, що замість того щоб вчити людину читати з нуля, працювати починають із тим, хто вже вміє читати.

У випадку з нейронними мережами попередньо навчена модель є моделлю, що була вже натренована для виконання певної задачі. Але замість

того, щоб відразу користуватися нею, виконується її донавчання для виконання іншої задачі. Технічно це реалізовано так, що останній шар попередньо навченої моделі заміняється на інший шар, що потрібен для її донавчання виконувати нову задачу. Існує два основних способи донавчання моделі:

- Ваги базових шарів моделі не змінюються, а навчання полягає в коригування ваг нового останнього шару;
- Ваги базових шарів змінюються, але не так швидко як ваги останнього шару.

Перевагами донавчання є:

- Економія часу та ресурсів на тренування, адже модель не доводиться навчати з нуля – базові знання вона вже отримала на попередньому етапі;
- Здатність показувати хороші результати навіть тоді, коли набір даних для конкретної задачі є невеликим, оскільки модель спирається на досвід, накопичений під час початкового навчання;
- Висока якість роботи у вузькій предметній галузі – у межах свого домену донавчена модель зазвичай перевершує універсальні моделі загального призначення.

Водночас варто врахувати проблеми, що можуть виникнути:

- Перенавчання. Якщо набір даних для донавчання є надто малим, модель ризикує запам'ятати самі приклади замість того, щоб виявити загальні закономірності. У такому разі вона показуватиме високу точність на тестових даних, але погано працюватиме з новими, раніше небаченими прикладами;
- Втрата раніше засвоєних знань. Якщо під час донавчання змінювати ваги базових шарів зі завеликою швидкістю навчання, модель може поступово «забути» те, що вивчила на попередньому етапі.

2.2 Огляд технологій та підходів у розробці системи

2.2.1 Бібліотека обробки тексту spaCy

Однією із найвідоміших бібліотек для роботи із неструктурованими текстовими даними є spaCy. Особливістю бібліотеки є те, що створювалася вона для розв’язання прикладних задач, а не проведення наукових досліджень. За головну мету розробники бібліотеки мали створення зручного, швидко та надійного інструменту для розробки реальних застосунків.

Центральні структури даних та більшість складних обчислень у цій бібліотеці були написані за допомогою мови Cython. Завдяки цьому бібліотека працює швидше та добре підходить для обробки великої кількості інформації.

Ключовою можливостю бібліотеки є здатність виконувати наступні задачі обробки текстових даних:

- Токенізація;
- Розпізнавання іменованих сутностей;
- Визначення частин мови;
- Лематизація;
- Знаходження синтаксичної залежності між словами.

Весь процес обробки тексту в spaCy побудований за принципом конвеєра. Коли необроблений текст потрапляє в систему, він послідовно проходить через різні етапи. Спочатку текст токенізується, потім визначаються частини мови, розбираються синтаксичні зв'язки, і на фінальних етапах застосовуються складніші алгоритми, такі як NER [16].

Зручність такого підходу полягає в тому, що розробник може самостійно керувати цим конвеєром. Зайві кроки можна просто вимкнути, щоб зекономити обчислювальні ресурси сервера, або ж навпаки – додати власні правила чи компоненти у загальний потік виконання.

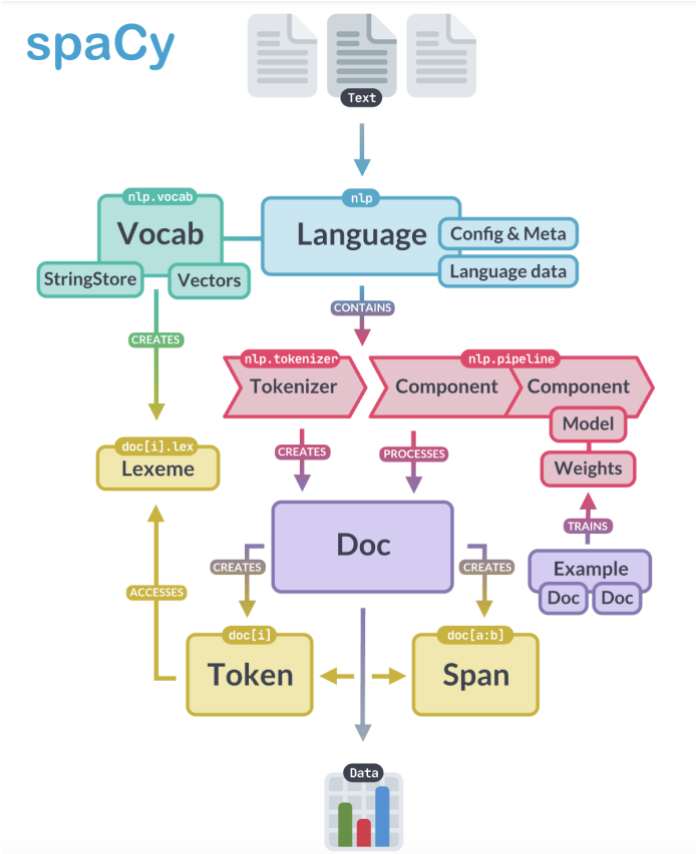


Рис. 2.4 – Архітектура та основні компоненти бібліотеки spaCy [17]

Також значною перевагою бібліотеки є наявність уже готових натренованих моделей для багатьох мов. Замість того, щоб збирати великі корпуси тексту і навчати алгоритми з нуля для базового синтаксичного аналізу, можна завантажити готову модель. Вона одразу після ініціалізації здатна якісно аналізувати текстові дані, готуючи їх до подальшого використання у складніших задачах, таких як класифікація.

2.2.2 Огляд вебфреймворків для розробки API

Серверна частина системи має приймати запити, передавати текст у відповідні модулі аналізу, повертати користувачу результат, а також контролювати доступ до системи. Від вибору вебфреймворку залежить, наскільки зручно буде реалізувати цю логіку та інтегрувати її з бібліотеками обробки природної мови. Для мови Python існує декілька зрілих рішень, тому

перед початком розробки доцільно розглянути два найпоширеніші з них – FastAPI та Django.

FastAPI – це сучасний вебфреймворк, основним призначенням якого є побудова програмних інтерфейсів. Він працює поверх двох бібліотек: Starlette, що відповідає за асинхронну обробку мережевих запитів, та Pydantic, яка виконує валідацію й серіалізацію даних. Взаємодія клієнта із сервером реалізована за принципами архітектурного стилю REST через стандартні методи HTTP — GET, POST, PUT та DELETE. Кожному маршруту відповідає окрема функція-обробник, оголошена за допомогою декоратора, тож із самого опису маршруту видно, який метод використовується, які параметри очікуються і яку структуру матиме відповідь. До сильних сторін FastAPI належать висока продуктивність, нативна підтримка асинхронності, автоматична валідація вхідних даних та автоматична генерація інтерактивної документації [18].

Окремої уваги заслуговує підхід FastAPI до роботи з даними. Фреймворк побудований навколо стандартних анотацій типів Python: розробник описує очікувану структуру запиту у вигляді звичайного класу, а бібліотека Pydantic автоматично перевіряє кожен вхідний запит на відповідність цій структурі [19]. Якщо дані некоректні, клієнт одразу отримує зрозуміле повідомлення про помилку, і запит навіть не доходить до логіки обробки. Цей самий опис типів використовується і для автоматичної генерації інтерактивної документації, яка завжди залишається синхронізованою з фактичним кодом. Ще однією важливою рисою є асинхронність: оскільки аналіз великих текстів може тривати помітний час, здатність обробляти запити асинхронно дозволяє серверу не блокуватися й продовжувати приймати нові звернення, поки виконуються попередні. У підсумку взаємодія між шарами системи виглядає так: клієнт надсилає дані у тілі HTTP-запиту, FastAPI валідує їх через Pydantic-модель, передає у відповідний модуль аналізу тексту й повертає результат у форматі JSON.

Django, своєю чергою, є одним із найвідоміших і найзріліших рішень для розробки вебзастосунків на Python. На відміну від мінімалістичного FastAPI, він є повнофункціональним фреймворком і містить усі компоненти для побудови повноцінного вебзастосунку: власну ORM, систему міграцій, шаблонізатор для генерації HTML-сторінок, готову адміністративну панель, вбудовану автентифікацію, механізми кешування та роботи з формами. Серед його переваг – зрілість і стабільність, потужна ORM, розвинена система безпеки та узгодженість усіх компонентів між собою. Завдяки великій спільноті, обширній документації та значній кількості сторонніх пакетів Django добре підходить для супроводу великих проєктів. Водночас така повнота має й зворотний бік: значна частина вбудованих компонентів у проєкті, орієнтованому суто на API, залишається незатребуваною, що лише збільшує обсяг кодової бази та час старту застосунку [20].

Зіставлення двох фреймворків показує, що вони розраховані на різні сценарії. Django є оптимальним для класичних вебзастосунків зі складною бізнес-логікою, адміністративною частиною та сторінками, що генеруються на сервері. Розроблювана ж система може не мати ні графічного інтерфейсу, ні складної серверної логіки відображення – її єдиним завданням є надання API до моделей обробки природної мови. У такому сценарії сильні сторони Django здебільшого не використовуються, тоді як його монолітність стає радше тягарем. FastAPI, навпаки, від початку спроєктований саме під побудову інтерфейсів: він забезпечує високу продуктивність, асинхронну обробку запитів, автоматичну валідацію та документацію без зайвих надбудов.

2.2.3 Обмеження частоти запитів користувачів

Система матиме демонстраційну сторінку, на якій будь-хто може випробувати її можливості без реєстрації та отримання API-ключа. Це зручно

для ознайомлення, але водночас створює проблему: оскільки сторінка відкрита й не вимагає автентифікації, нею можна користуватися як безкоштовним доступом до функцій системи в обхід реєстрації. Окремий користувач здатен надсилати запити безперервно, навантажуючи сервер і фактично перетворюючи демонстрацію на повноцінний робочий інструмент, яким вона не має бути.

Щоб демонстрація лишалася відкритою, але не зловживалася, застосовано механізм обмеження частоти запитів. Його суть проста: для кожного клієнта ведеться підрахунок звернень за певний проміжок часу, і щойно їх кількість перевищує дозволений поріг, сервер тимчасово відмовляє в обробці нових запитів. Через деякий час лічильник скидається, і клієнт знову може користуватися сторінкою. Таким чином поодинокі звернення для ознайомлення проходять вільно, а спроби систематичного навантаження відсікаються.

Для реалізації цього механізму використано Redis – сховище структур даних, що тримає інформацію в оперативній пам'яті. Такий вибір зумовлений характером задачі: лічильники запитів потрібно дуже швидко збільшувати й перевіряти на кожне звернення, а також автоматично скидати після завершення відведеного проміжку. Redis добре підходить для цього, оскільки виконує операції в пам'яті майже миттєво [21].

Кожному клієнту відповідає окремий лічильник у Redis, прив'язаний до його адреси. Щоб не зберігати самі адреси у відкритому вигляді, замість адреси використовується її хеш. На кожен запит лічильник збільшується на одиницю, і для нього встановлюється обмежений час життя. Якщо значення лічильника перевищує заданий поріг, сервер повертає відповідь із кодом, що означає завелику кількість запитів, і додає заголовок із часом, через який можна повторити спробу.

```python
async def rate_limit(request: Request) -> None:
    ip_hash = hashlib.sha256(_client_ip(request).encode()).hexdigest()
    key = f"ratelimit:{ip_hash}"

    pipe = redis.pipeline()
    pipe.incr(key)
    pipe.expire(key, settings.rate_limit_window_seconds, nx=True)
    count, _ = await pipe.execute()

    if count > settings.rate_limit_requests:
        ttl = await redis.ttl(key)
        retry_after = ttl if ttl > 0 else settings.rate_limit_window_seconds
        raise HTTPException(
            status_code=429,
            detail="Rate limit exceeded",
            headers={"Retry-After": str(retry_after)},
        )
```

Рис. 2.5 – функція обмеження частоти запитів

Обраний підхід дозволяє захистити демонстраційну сторінку без ускладнення архітектури системи. Поріг і тривалість проміжку винесено в налаштування, тож їх можна змінювати без втручання в код, підлаштовуючи під реальне навантаження. Для зареєстрованих користувачів, що працюватимуть через API-ключі, передбачено окремий контроль доступу, описаний пізніше.

2.2.4 Система керування базою даних

Окрім вебфреймворку, ще однією критично важливою складовою системи є сховище даних. Системі необхідно зберігати облікові записи користувачів та токени автентифікації. Від правильно обраної бази даних безпосередньо залежить надійність, продуктивність та можливість подальшого розвитку системи.

На сьогодні всі сучасні рішення для зберігання інформації прийнято поділяти на дві великі групи – реляційні (SQL) та нереляційні (NoSQL) бази даних. Кожен із цих підходів має власну філософію, сильні та слабкі сторони,

тому остаточний вибір залежить від характеру даних, з якими доведеться працювати конкретному програмному комплексу.

Уся інформація в реляційних базах даних представлена у вигляді таблиць, що складаються із рядків і стовпців, де кожен стовпець має суворо визначений тип, а між таблицями встановлюються логічні зв'язки через зовнішні ключі. Робота з даними здійснюється за допомогою декларативної мови запитів SQL, що дозволяє описувати, який саме результат потрібно отримати, не вдаючись у деталі того, як саме сховище буде його збирати. Найвідомішими реляційними базами даних є PostgreSQL, MySQL, Oracle Database, Microsoft SQL Server.

Нереляційні бази даних з'явилися як відповідь на потреби сучасних вебсервісів, що оперують величезними обсягами слабо структурованих даних. На відміну від класичних SQL-рішень, NoSQL не вимагає попереднього визначення жорсткої схеми, а сам термін охоплює одразу декілька принципово різних родин сховищ:

- Документоорієнтовані, що зберігають дані у вигляді JSON-подібних документів довільної структури;
- Ключ-значення, орієнтовані на максимально швидкий доступ за ключем;
- Стовпчикові, оптимізовані для аналітики над величезними наборами даних;
- Графові, призначені для обробки складно пов'язаних між собою сутностей.

Сильною стороною NoSQL баз даних є гнучкість структури, горизонтальне масштабування та висока швидкодія на простих операціях. Водночас більшість NoSQL-сховищ свідомо відмовляються від частини принципів ACID на користь доступності й продуктивності, а складні аналітичні запити доводиться реалізовувати на рівні застосунку, що збільшує обсяг коду та ймовірність помилок.

Окремої уваги заслуговує питання цілісності даних. Оскільки система передбачає механізм автентифікації та обмеження доступу, неприпустимою є ситуація, коли API ключ існує без власника, або задача обробки даних залишається прив'язаною до видаленого користувача. Реляційні бази даних розв'язують подібні проблеми через обмеження зовнішніх ключів та транзакції. У світі NoSQL подібні гарантії доводиться реалізовувати у коді застосунку, що додає складності та збільшує ризик некоректних станів даних [22].

Ще одним аргументом на користь реляційного підходу є характер запитів, які виникають у системі. Адміністратору можуть знадобитися статистичні зведення: скільки документів було оброблено за певний період, які користувачі формують найбільше навантаження, який середній час обробки тексту тощо. Подібні запити елегантно виражаються засобами SQL за рахунок операцій з'єднання, агрегації та групування. У документоорієнтованій базі для отримання аналогічного результату довелося б писати громіздкі агрегаційні конвеєри або частково дублювати дані між колекціями.

З урахуванням наведених вище міркувань для зберігання даних розроблюваної системи було обрано PostgreSQL – одну з найпотужніших реляційних СКБД із відкритим вихідним кодом. Ця система давно вже вийшла за межі класичної SQL-бази й по суті є гібридним об'єктно-реляційним сховищем із широкими можливостями розширення [23].

2.2.5 Безпека та контроль доступу до системи

Окремої уваги під час проєктування будь-якої системи заслуговує питання безпеки. Оскільки розроблений застосунок надає доступ до обчислювально витратних моделей обробки природної мови через мережу, він автоматично стає привабливою ціллю для зловмисників. Неконтрольований доступ до API може призвести не лише до витоку конфіденційних даних, що

передаються користувачами на обробку, а й до повного вичерпання ресурсів сервера через надмірну кількість сторонніх запитів. Саме тому захист API є не другорядним питанням, а однією з ключових вимог до архітектури системи.

Існує декілька поширених підходів до автентифікації клієнтів у вебзастосунках: базова автентифікація за логіном і паролем, сесійні cookie, токени JWT, а також механізм API-ключів. Кожен із цих методів має свою сферу застосування. Сесії на основі cookie добре підходять для класичних вебсайтів, де клієнтом виступає браузер. JWT часто застосовують у системах із розподіленою архітектурою, де потрібна автентифікація без збереження стану сесії на сервері [24].

Для доступу до модулів обробки природної мови було обрано саме механізм API-ключів. Такий вибір зумовлений характером застосунку: основним клієнтом виступає не людина перед браузером, а інша програма чи скрипт, що звертається до системи для пакетної обробки текстів. У подібному сценарії використання логіна і пароля при кожному запиті є незручним. API-ключ, своєю чергою, становить собою унікальний довгий рядок випадкових символів, який клієнт передає у заголовку HTTP-запиту, і який однозначно ідентифікує власника та надані йому права.

Принцип роботи даного механізму є таким, що після реєстрації користувач отримує можливість згенерувати один або кілька ключів через особистий кабінет. Кожен такий ключ прив'язується до облікового запису в базі даних PostgreSQL та має набір додаткових атрибутів: дата створення, дата останнього використання, статус активності, а також за потреби термін дії. Під час обробки кожного вхідного запиту вебфреймворк перехоплює значення ключа із заголовка, шукає відповідний запис у базі та перевіряє, чи дійсно цей ключ є активним і чи має його власник право звертатися до конкретного ресурсу. У разі відсутності або недійсності ключа сервер одразу повертає помилку, не передаючи запит далі до черги обробки.

Важливим аспектом є спосіб зберігання ключів. Зберігати їх у відкритому вигляді у базі даних – це погана практика з точки зору безпеки, адже у випадку компрометації сховища усі ключі стають доступними для зловмисника. Тому ключі зберігаються у вигляді хешів. Користувач бачить повне значення ключа лише один раз – у момент його генерації, після чого система зберігає лише його хеш. Під час перевірки вхідного запиту відбувається хешування переданого ключа і порівняння отриманого результату зі збереженим у базі. Такий підхід виключає можливість відновлення оригінальних ключів навіть за наявності прямого доступу до таблиць.

Окрім самої автентифікації, механізм API-ключів дозволяє органічно вирішити ще декілька завдань безпеки. Передусім це обмеження частоти запитів. Прив'язавши лічильник звернень до конкретного ключа, можна не допустити ситуації, коли один клієнт ненавмисно чи свідомо генерує таку кількість задач, що блокує роботу системи для всіх інших користувачів. Реалізувати таке обмеження зручно за допомогою того ж Redis: для кожного ключа створюється відповідний лічильник із обмеженим часом життя, який інкрементується при кожному запиті. Якщо значення лічильника перевищує заздалегідь визначений поріг, сервер відповідає кодом, що означає завелику кількість запитів і не передає завдання у чергу.

Не менш важливим є відстеження дій кожного ключа. Завдяки тому, що всі задачі обробки тексту прив'язуються до конкретного ключа, у будь-який момент можна сформувати журнал активності та виявити підозрілу поведінку: різке зростання кількості запитів, звернення з нетипових IP-адрес чи спроби доступу до ресурсів, на які власник ключа не має прав. У разі підозри на компрометацію ключ можна миттєво деактивувати без необхідності змінювати пароль чи блокувати весь обліковий запис.

Додатковий рівень захисту забезпечується самим вебфреймворком FastAPI, який автоматично виконує валідацію вхідних даних за допомогою

бібліотеки Pydantic. Це усуває цілу низку класичних загроз, пов'язаних із некоректним або зловмисно сформованим вмістом запиту.

Використання API-ключів є оптимальним саме для систем, орієнтованих на програмних клієнтів. Він поєднує простоту реалізації та інтеграції з достатньо високим рівнем захисту, дозволяє гнучко керувати правами доступу, обмежувати навантаження та оперативно реагувати на інциденти. Хешування ключів та валідація вхідних даних механізм створює надійний захист, який відповідає сучасним вимогам до вебзастосунків.

2.2.6 Огляд фреймворків для розробки користувацького інтерфейсу

Окрім серверної частини, система передбачає вебінтерфейс, через який користувач реєструється, входить до особистого кабінету та керує своїми API-ключами. Хоча основним способом взаємодії із сервісом є програмний доступ через API, наявність зручного інтерфейсу спрощує ознайомлення із системою та керування обліковим записом. Тому перед розробкою клієнтської частини доцільно розглянути найпоширеніші фреймворки для побудови інтерфейсів – React, Angular та Vue.

React – це бібліотека для побудови інтерфейсів, розроблена компанією Meta. В її основі лежить компонентний підхід: інтерфейс складається з невеликих самодостатніх компонентів, кожен з яких відповідає за окрему частину сторінки та описує, як він має виглядати залежно від свого стану. React не нав'язує жорсткої структури проєкту й залишає розробнику свободу у виборі додаткових бібліотек для маршрутизації, керування станом тощо. Завдяки великій спільноті та широкій екосистемі React добре підходить для проєктів різного масштабу [25].

Angular – це повнофункціональний фреймворк від компанії Google, побудований на мові TypeScript. На відміну від React, він є цілісною

платформою й одразу містить усі компоненти для побудови застосунку: маршрутизацію, керування формами, механізм впровадження залежностей та засоби взаємодії із сервером [26]. Такий підхід забезпечує узгодженість великих проєктів, але водночас має вищий поріг входження та надлишкову складність для невеликих застосунків.

Vue – це прогресивний фреймворк, що позиціонується як компроміс між гнучкістю React та цілісністю Angular. Він також побудований на компонентах, має просту й зрозумілу структуру та невисокий поріг входження, що робить його зручним для швидкого старту [27]. Водночас його екосистема та спільнота менші за дві попередні.

Для розроблюваної системи обрано React. Цей вибір зумовлений тим, що клієнтська частина є відносно простою – вона зводиться до кількох екранів та форм, тож повнофункціональність Angular тут була б надлишковою. Водночас гнучкість React, велика екосистема готових рішень та компонентний підхід дозволяють зручно зібрати потрібний інтерфейс без зайвих надбудов. Крім того, React має найбільшу спільноту серед розглянутих фреймворків, що спрощує пошук рішень під час розробки.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі розглянуто технології та підходи, на яких ґрунтується розробка системи, і за результатами цього аналізу сформовано технологічний стек майбутньої системи.

Спершу опрацьовано теоретичні основи обробки природної мови. Токенізацію визначено як перший і базовий етап, від якого залежить якість усіх подальших кроків. З розглянутих підходів, а саме токенізації на основі правил та статистичної, для практичних задач системи найдоцільнішою є статистична токенізація: вона розв'язує проблему невідомих слів, утримує розмір словника в розумних межах і наразі застосовується у більшості сучасних мовних моделей. Далі розглянуто перетворення токенів у векторні представлення. Розглянуто й засоби врахування контексту: рекурентні мережі, обмежені проблемою затухання градієнта, та архітектуру LSTM; коротко згадано механізм уваги й архітектуру Transformer як подальший напрям розвитку. Проаналізовано концепцію претренованих моделей: донавчання вже готової моделі дозволяє економити ресурси та отримувати якісні результати навіть на невеликих наборах даних.

На основі огляду інструментів сформовано стек системи. Бібліотекою обробки тексту обрано spaCy за конвеєрну архітектуру та наявність готових натренованих моделей. Для серверної частини порівняно FastAPI та Django; перевагу віддано FastAPI через високу продуктивність, нативну асинхронність і автоматичну валідацію даних, тоді як сильні сторони Django у проєкті, орієнтованому здебільшого на API, лишаються незадіяними. Для зберігання облікових записів користувачів та API-ключів обрано реляційну СКБД PostgreSQL, оскільки вона забезпечує цілісність даних через зовнішні ключі й транзакції та зручно обслуговує характерні для системи запити. Розглянуто потребу захистити відкриту демонстраційну сторінку від зловживань: для

обмеження частоти запитів обрано Redis, який тримає лічильники звернень у оперативній пам’яті та виконує операції над ними практично миттєво.

Окрему увагу приділено питанню безпеки. Серед підходів до автентифікації обрано механізм API-ключів як найбільш відповідний сценарію, де клієнтом системи виступає програма, а не людина. Визначено, що ключі мають зберігатися у вигляді хешів. Для побудови вебінтерфейсу серед React, Angular та Vue обрано React, оскільки клієнтська частина є відносно простою, а гнучкість бібліотеки та її велика екосистема дозволяють зібрати потрібний інтерфейс без зайвих надбудов.

РОЗДІЛ 3

РЕАЛІЗАЦІЯ ГОЛОВНОГО ФУНКЦІОНАЛУ ПЛАТФОРМИ

3.1 Реалізація модуля виявлення спаму

3.1.1 Підготовка навчального набору даних

Якість моделі, що донавчається, безпосередньо залежить від якості та різноманітності навчальних даних. Для задачі класифікації листів існує кілька відкритих корпусів, проте жоден із них не покриває потреби системи повністю. Тому навчальний набір було сформовано як комбінацію реальних даних із відкритого джерела та синтетично згенерованих листів, які доповнюють реальний корпус у тих категоріях, де його виявилося недостатньо.

Основою набору даних став корпус CEAS_08 – відкрита колекція електронних листів, зібрана в межах боротьби зі спамом і фішингом [28]. Кожен лист у цьому корпусі вже має розмітку на дві категорії: спам і звичайний лист. Проте навіть цей готовий розмічений корпус не варто використовувати напряму, оскільки він містить значну кількість шуму, що шкодить тренуванню моделі. Тому перед тим, як включити CEAS_08 у фінальний датасет, було виконано кілька етапів очищення. Спочатку відкидаються рядки, у яких відсутнє тіло листа або мітка класу. Далі видаляються дублікати: один і той самий лист, що повторюється в корпусі багато разів, спотворить пропорції класів і призведе до того, що модель надмірно запам'ятає саме його, а не загальні закономірності. Після очищення з корпусу робиться вибірка

фіксованого розміру, що зберігає природне співвідношення спаму й звичайних листів у вихідних даних.

Під час подальшого аналізу було з'ясовано, що набір даних CEAS_08 добре покриває різноманітні різновиди спаму такі як класичний фішинг, однак має недостатньо прикладів транзакційних листів. Йдеться про автоматичні листи на кшталт підтверджень оплати, чеків з онлайн-магазинів, нотифікацій банку про переказ коштів, підтверджень бронювань. Такі листи мають важливу особливість: вони поверхово схожі на спам, оскільки містять суми грошей, назви компаній, посилання та заклики до дії, але насправді є критично важливою інформацією для користувача. Якщо модель не побачить достатньо таких прикладів під час навчання, вона з високою ймовірністю позначатиме їх як спам.

З огляду на це було ухвалено рішення доповнити навчальний набір даних двома типами синтетично згенерованих листів. Перший тип – це додаткові спам-листи, які повторюють найпоширеніші схеми реального спаму: фальшиві сповіщення про блокування акаунта, фішинг від імені відомих компаній, обіцянки виграшів та інші шахрайські схеми. Другим і не менш важливим типом є транзакційні листи, які якраз і компенсують прогалину CEAS_08.

Для генерації було розроблено окремі скрипти, що будуть генерувати навчальні дані. У кожному з них визначено набір текстових шаблонів типових листів і можливих значень для підстановки: назви компаній, грошові суми, найменування товарів, посилання, імена адресатів. Під час генерації значення з пулів випадково підставляються у шаблони, утворюючи на виході різноманітні, але правдоподібні листи. Такий підхід дозволяє за помірних зусиль отримати велику кількість прикладів, що покривають типові варіації в межах кожної категорії. Також варто сказати, що якщо просто заповнити шаблони чистим, граматично правильним текстом, отримана модель буде розпізнавати лише такий «ідеальний» спам або транзакційні листи, а на реальних листах із певними граматичними помилками, нерівним регістром і дивним форматуванням

покаже значно гіршу якість. Тому в генератор було свідомо закладено внесення шуму: випадкові одруківки, нестандартне використання великих літер у середині слів, надмірно офіційний або, навпаки, неприродно емоційний стиль викладу. Такі дрібні дефекти роблять синтетичні приклади ближчими до реальних і допомагають моделі навчитися шукати ширші ознаки спаму, а не його поверхневу форму. Оскільки синтетичні дані можуть привести до того, що модель запам’ятає лише ці шаблони та не зможе працювати із новими даними, кількість синтетичних даних у всьому наборі даних є невеликою – близько десяти відсотків.

Після того, як усі дані були готові, вони об'єднуються в єдиний набір даних. На цьому етапі повторно виконується видалення дублікатів, оскільки шаблонна генерація могла випадково створити лист, схожий на той, що вже є в CEAS_08, тому такі збіги відсіюються. Завершальним кроком є випадкове перемішування рядків, після якого набір даних готовий для навчання моделі.

3.1.2 Донавчання моделі DistilBERT

Маючи готовий навчальний датасет, наступним кроком є вибір способу побудови класифікатора. Для цієї задачі було обрано підхід донавчання попередньо натренованої моделі. Він дозволяє скористатися знаннями про мову, які модель уже здобула на великому корпусі загального тексту, та адаптувати їх до конкретної задачі за допомогою порівняно невеликої спеціалізованої вибірки. Тренування моделі з нуля вимагало би значно більших обсягів даних, обчислювальних ресурсів і часу, а очікувана якість на задачі бінарної класифікації листів навряд чи виправдала би ці витрати [29].

За базову модель було обрано DistilBERT – компактнішу версію відомої моделі BERT. DistilBERT зберігає близько дев'яноста семи відсотків якості оригінальної моделі на стандартних мовних задачах, водночас будучи

приблизно вдвічі швидшим та значно меншим [30]. Для сервісу, який має обробляти запити користувачів у реальному часі, така характеристика є вагомою. Класифікатор повинен відповідати швидко, інакше час обробки одного листа стане відчутним для користувача. Використання важчих варіантів моделей сімейства BERT дало б у кращому разі незначний приріст якості, але суттєво збільшило би навантаження на сервер та витрати на інфраструктуру.

Перед подачею в модель кожен лист із датасету проходить додаткову підготовку. Корисна інформація для класифікації листа міститься як у його темі, так і в тілі, причому ці дві частини доповнюють одна одну. Тема часто буває коротшою, але більш концентрованою за змістом: саме в темі спам-листів зосереджені характерні слова-маркери на кшталт «терміново», «виграш», «підтвердіть акаунт». Тіло, своєю чергою, містить детальніший контекст, посилання та заклики до дії. Тому тема й тіло об'єднуються в один текст, після чого передаються токенізатору моделі.

```
train_df, temp_df = train_test_split(
    df_sample, test_size=0.20, stratify=df_sample["label"], random_state=SEED
)
val_df, test_df = train_test_split(
    temp_df, test_size=0.50, stratify=temp_df["label"], random_state=SEED
)
```

Рис. 3.1 – поділ набору даних на тренувальний, валідаційний та тестовий

Підготовлений датасет розбито на тренувальну, валідаційну та тестову частини у пропорціях вісімдесят, десять та десять відсотків відповідно. Кожна з трьох вибірок зберігає те саме співвідношення спаму й звичайних листів, що й вихідний датасет. Це принципово важливо для коректного оцінювання: якщо тестова вибірка випадково отримає викривлене співвідношення класів, метрики стануть нерепрезентативними. Тренувальна частина набору даних використовується для оновлення ваг моделі під час донавчання, валідаційна – для відстеження якості в процесі тренування та своєчасного виявлення перенавчання, а тестова жодним чином не бере участі в навчанні й слугує лише

для фінального оцінювання моделі на даних, яких вона раніше не бачила. Наступним кроком було виконано тренування моделі.

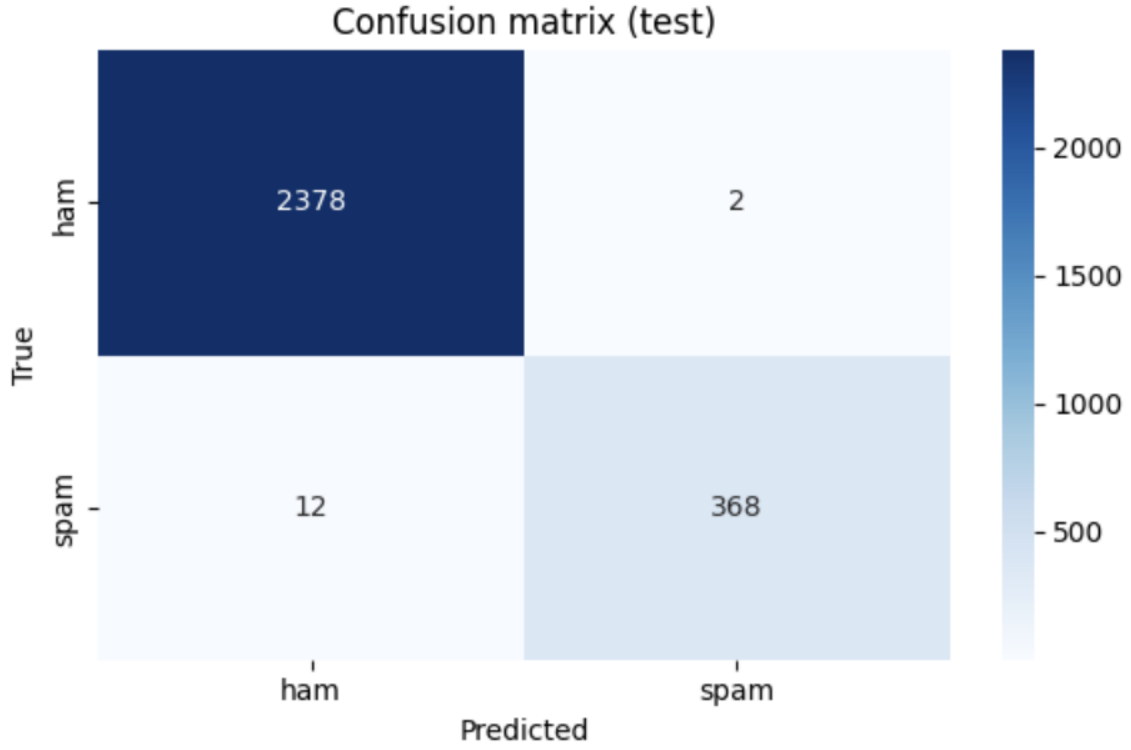


Рис. 3.2 – матриця помилок для тестового набору даних

Варто зазначити, що в контексті класифікації спаму термін «ham» означає текст, що не є спамом. Із матриці видно, що модель після тренування показала точність близько 99%. Така точність моделі може бути підозрілою, оскільки вона занадто висока та може вказувати на перенавчання. Але в даному випадку дублікати було видалено із тестового набору даних і до того ж задача бінарної класифікації є досить простою для моделей-трансформерів, тому такий результат є очікуваним. У цьому випадку тестова вибірка була складена із того ж набору даних, що і тренувальна та валідаційна. Тому наступним кроком було взято інший набір даних (spam-detection-dataset-splits) [31] для тестування моделі.

	precision	recall	f1-score	support
ham	0.84	0.99	0.90	680
spam	0.98	0.81	0.89	682
accuracy			0.90	1362
macro avg	0.91	0.90	0.90	1362
weighted avg	0.91	0.90	0.90	1362

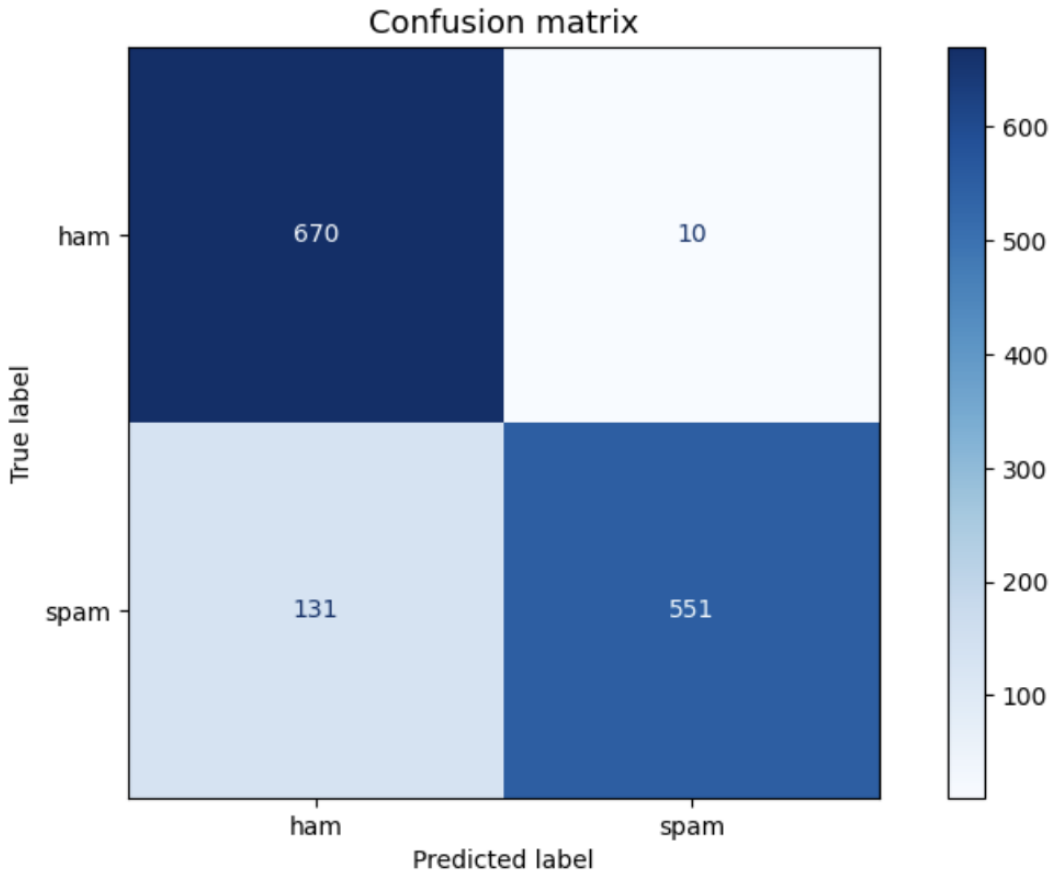


Рис. 3.3 - метрики та матриця помилок класифікатора для нового набору даних

Матриця вище демонструє, що модель правильно класифікувала 670 з 680 ham-повідомлень та 551 з 682 спам-повідомлень. Пропущений 131 спам-лист пояснюється явищем domain shift [32] – відмінністю у стилі, мові та структурі спаму між тренувальним набором та тестовим. Незважаючи на це, модель

досягає точності 90% та F1-score 89% на повністю невідомих даних, що підтверджує її здатність до узагальнення за межами тренувального розподілу.

3.2 Реалізація модулів анонімізації персональних даних та вилучення ключових слів у тексті

3.2.1 Розробка модуля анонімізації персональних даних

Одним із ключових модулів для обробки природної мови, реалізованих у системі, є модуль анонімізації персональних даних. Його завдання – знайти у вхідному тексті фрагменти, що дозволяють ідентифікувати особу, та замінити їх позначками так, щоб зміст тексту залишався зрозумілим, а сам суб'єкт ставав невпізнаваним. Складність цієї задачі полягає в тому, що персональні дані бувають двох різних типів. Частина з них має чітку та передбачувану структуру – адреси електронної пошти, номери телефонів, банківських карток чи IP-адреси завжди підпорядковуються певному формату. Інша ж частина, передусім імена людей, назви організацій та географічні назви, жодного формального шаблону не має й розпізнається лише за контекстом. Через це в основу модуля покладено комбінований підхід, що поєднує два незалежні методи виявлення персональних даних.

Перший метод спирається на регулярні вирази й відповідає за структуровані дані. Для кожного типу таких даних складено окремий шаблон: для електронних адрес, посилань, адрес IPv4 та IPv6, MAC-адрес, номерів банківських карток, ідентифікаційних номерів та банківських рахунків у форматі IBAN. Перевагою цього підходу є його передбачуваність та висока швидкодія, оскільки пошук за шаблоном не потребує попереднього навчання й виконується за один прохід тексту. Другий метод відповідає за неструктуровані дані й використовує модель розпізнавання іменованих сутностей із бібліотеки

spaCy. Модель аналізує текст із урахуванням контексту та повертає знайдені сутності разом із їхнім типом, після чого мітки spaCy відображаються на власні позначки системи: особа стає позначкою для імені, географічна назва та споруда – позначкою місцезнаходження, організація та національність – відповідними їм мітками. Таке відображення дозволяє звести різні типи сутностей до невеликого набору узагальнених категорій, зрозумілих кінцевому користувачеві.

Поєднання двох методів породжує проблему перекриття знайдених фрагментів, коли один і той самий відрізок тексту може бути позначений одразу кількома детекторами. Наприклад, послідовність цифр може одночасно нагадувати і номер телефону, і номер картки, а доменна частина електронної адреси – фрагмент посилання. Щоб уникнути некоректних замін, кожному типу даних присвоєно пріоритет, який відображає однозначність відповідного шаблону. Найвищий пріоритет надано чітко структурованим ідентифікаторам на кшталт електронної пошти чи IP-адреси, найнижчий – результатам розпізнавання іменованих сутностей, оскільки вони мають імовірнісну природу й можуть бути неточними. Усі знайдені фрагменти збираються в єдиний перелік, після чого впорядковуються за пріоритетом, а потім за довжиною. Далі модуль послідовно переглядає цей перелік і додає фрагмент до результату лише тоді, коли він не перетинається з жодним уже прийнятим. Завдяки такому порядку остаточний результат не залежить від того, у якій послідовності спрацювали окремі детектори, а серед конкурентних варіантів завжди перемагає найбільш надійний.

Окремої уваги потребувало розпізнавання номерів банківських карток. Простий шаблон, що шукає послідовність із тринадцяти–дев'ятнадцяти цифр, дає велику кількість хибних спрацювань, оскільки під нього підпадають будь-які довгі числові рядки, наприклад ідентифікатори чи номери документів. Щоб відсіяти такі випадки, кожен фрагмент, який претендує на роль номера картки, додатково перевіряється за контрольною сумою, якій підпорядковуються

номери реальних платіжних карток. Фрагмент позначається як номер картки лише тоді, коли він успішно проходить цю перевірку, що дозволяє різко зменшити кількість помилкових замін без ускладнення самого шаблону.

Важливою вимогою до модуля стала послідовність заміни. Якщо одне й те саме ім'я або номер трапляється в тексті кілька разів, усі його входження мають бути замінені однаковою позначкою, інакше зв'язність тексту втрачається. Для цього модуль зберігає відповідність між уже обробленими значеннями та виданими позначками, нормалізуючи значення без урахування регістру та зайвих пробілів. Завдяки цьому повторні згадки тієї самої сутності отримують ту саму мітку, а текст залишається придатним для подальшого читання й аналізу. Результатом роботи модуля є не лише текст із прихованими персональними даними, а й перелік виконаних замін, у якому кожній позначці відповідають виявлені під нею оригінальні значення. Оскільки сам цей перелік містить персональні дані, за замовчуванням оригінали в ньому маскуються – зберігається лише невеликий фрагмент значення, достатній для контролю якості, але недостатній для ідентифікації особи.

Загальний порядок роботи модуля виглядає так, що спершу до тексту застосовуються всі регулярні вирази. Потім той самий текст обробляється моделлю розпізнавання іменованих сутностей. Усі знайдені фрагменти об'єднуються та проходять відбір за пріоритетом. На завершальному кроці модуль проходить текст зліва направо, замінює прийняті фрагменти відповідними позначками з урахуванням уже виданих міток і формує підсумковий перелік замін. Такий підхід дозволяє надійно знеособлювати як структуровані, так і неструктуровані персональні дані, зберігаючи при цьому читабельність тексту, та узгоджується із загальною метою системи – унеможливити ідентифікацію суб'єкта в тексті.

3.2.2 Розробка модуля пошуку ключових слів

Ще один модуль обробки природної мови в системі – це модуль вилучення ключових слів. Його завданням є знаходження у вхідному тексті слів й словосполучень, що найкраще передають його зміст. Такий функціонал може використовуватися для групування документів за темами та пошуку за змістом, а ще дозволяє швидко зрозуміти, про що йдеться у великому тексті, не читаючи його повністю.

Головна складність полягає в тому, що важливість слова не зводиться до того, як часто воно зустрічається. Часто повторюване слово цілком може виявитися службовим, тоді як справжній ключовий термін іноді згадується лише раз чи двічі. Для пошуку ключових слів обрано алгоритм TextRank. Він працює в межах одного документа й не потребує ні навчання, ні зовнішнього корпусу. Текст у ньому подається як граф: вузлами є слова, а зв'язки виникають між тими словами, що стоять поруч. Чим більше вагомих слів пов'язано з певним словом, тим важливішим воно вважається [33]. Завдяки такому підходу алгоритм оцінює слова за їхньою роллю в тексті, а не за позицією чи простою частотою, тож головні терміни виявляються незалежно від того, у якій частині документа вони згадані.

Сам алгоритм видобуває чимало фраз-кандидатів, серед яких трапляються вкладені одна в одну – наприклад, коротша фраза цілком міститься в довшій. Якщо лишити їх усі, перелік ключових слів засмічується повторами того самого поняття. Тому після видобутку виконується додатковий крок: серед фраз, де одна є частиною іншої, у результаті лишається тільки та, що має вищу оцінку. Так перелік стає чистішим і не дублює одну й ту саму думку кількома записами.

На виході формується перелік ключових слів разом із їхньою релевантністю, впорядкований від найважливіших до найменш важливих, що

зручно як для людини, так і для програмних клієнтів. Код функції, що виконує цей процес, наведено на рис. 3.4.

```python
def extract_keywords(text: str, top_n: int = 10) -> list[dict]:
    text = (text or "").strip()
    if not text:
        return []

    nlp = _get_nlp(_detect_lang(text))
    doc = nlp(text)

    phrases = [(p.text, float(p.rank)) for p in doc._.phrases if p.rank > 0]
    if not phrases:
        return []

    phrases.sort(key=lambda x: x[1], reverse=True)
    deduped = _dedupe_nested(phrases)[:top_n]
    if not deduped:
        return []

    max_weight = max(w for _, w in deduped)
    return [
        {"phrase": phrase, "relevance": round(weight / max_weight, 2)}
        for phrase, weight in deduped
    ]
```

Рис. 3.4 – функція вилучення ключових слів

Модуль реалізовано у вигляді кількох функцій, які послідовно перевіряють вхідний текст, застосовують алгоритм, прибирають вкладені дублікати й нормують оцінки. Він не потребує важких нейронних моделей, швидко працює і не висуває значних вимог до ресурсів сервера.

3.3 Програмна реалізація серверної частини платформи

Одним із ключових компонентів системи є сервер. За допомогою нього здійснюється контроль доступу до особистого кабінету зареєстрованих користувачів, створення та видалення токенів для доступу до модулів обробки природної мови, взаємодія із іншими компонентами системи для її роботи.

3.3.1 Аутентифікація через JWT

Для розмежування доступу зареєстрованих користувачів до приватної частини системи застосовано механізм автентифікації на основі JSON Web Token. Цей підхід дозволяє серверу впізнавати користувача за кожним запитом, не зберігаючи при цьому стану сесії на своєму боці. Уся необхідна інформація про автентифікованого користувача міститься безпосередньо у токені, а цілісність цієї інформації гарантується криптографічним підписом.

Сам токен складається з трьох частин: заголовка з описом використаного алгоритму, корисного навантаження з даними про користувача та підпису. До корисного навантаження записується ідентифікатор користувача, момент створення токена та момент завершення його дії. Підпис формується хеш-функцією SHA-256 та секретним ключем, відомим лише серверу. Якщо хоча б один символ у токені змінити після створення, підпис перестане збігатися, і сервер відхилить такий токен як недійсний. Завдяки симетричній природі обраного алгоритму перевірка підпису є швидкою операцією і не створює помітного навантаження навіть при великій кількості запитів.

Час життя токена свідомо обмежено, щоб мінімізувати наслідки його можливого витоку. Якщо зловмисник якимось чином заволодів дійсним токеном, час, протягом якого він зможе ним скористатися, є обмеженим, після чого токен автоматично втрачає чинність і потребує повторного входу користувача. Створення токена реалізовано окремою функцією, фрагмент коду якої наведено на рисунку 3.5.

```python
def create_access_token(user_id: UUID | str) -> str:
    now = datetime.now(timezone.utc)
    payload = {
        "sub": str(user_id),
        "iat": int(now.timestamp()),
        "exp": int((now + timedelta(minutes=settings.jwt_expires_minutes)).timestamp()),
    }
    return jwt.encode(payload, settings.jwt_secret, algorithm=settings.jwt_algorithm)
```

Рис. 3.5 – функція створення JWT токену

Отриманий токен клієнт прикріплює до кожного наступного запиту у заголовку Authorization. Сервер перевіряє підпис і термін дії токена, після чого витягує з нього ідентифікатор користувача та використовує його для виконання запитуваної операції. У разі простроченого або підробленого токена сервер повертає відповідь із кодом 401, не передаючи запит далі до бізнес-логіки.

3.3.2 Використання API-ключів для доступу до модулів обробки природної мови

Оскільки система обслуговує дві принципово різні категорії клієнтів – людину, що взаємодіє через вебінтерфейс, і програмні застосунки, що звертаються до API – використання єдиного механізму автентифікації для обох випадків було б неоптимальним. Короткоживучий JWT-токен зручний для роботи через браузер, але незручний для тривалих програмних інтеграцій, які потребують стабільного доступу без періодичного повторного входу. Натомість довготривалий API-ключ погано підходить для роботи людини, оскільки зберігання такого ключа у браузері створює зайві ризики безпеки. Тому в системі реалізовано два паралельні механізми.

Згенероване значення API-ключа являє собою випадковий рядок із тридцяти двох байтів. Така довжина робить визначення ключа методом перебору надзвичайно складним. Проте сам по собі довгий випадковий рядок ще не гарантує безпеки: якщо зберігати ключі у базі даних у відкритому вигляді, компрометація сховища одразу відкриває зловмиснику доступ до всіх ключів. Щоб уникнути цього, після генерації значення ключа одразу хешується за допомогою функції SHA-256, і у базу записується лише отриманий хеш. Оригінал повертається користувачу один раз – у відповіді на запит створення. Після цього відновити його неможливо навіть для адміністратора системи. Функції для генерації та хешування ключа наведено на рисунку 3.6.

```python
def generate_api_token() -> tuple[str, str]:
    raw = secrets.token_urlsafe(32)
    return raw, hash_api_token(raw)


def hash_api_token(raw: str) -> str:
    return hashlib.sha256(raw.encode("utf-8")).hexdigest()
```

Рис. 3.6 – функції генерації та хешування API-ключа

Під час перевірки вхідного запиту сервер бере переданий клієнтом ключ, хешує його тією ж функцією і порівнює отриманий результат із збереженим у базі. Збіг хешів підтверджує, що клієнт володіє оригінальним ключем, при цьому самого оригіналу серверу для перевірки знати не потрібно. Такий підхід відповідає тим самим принципам, за якими у системі зберігаються паролі користувачів.

Користувач не обмежений одним ключем – система дозволяє створити їх довільну кількість. Кожному ключу можна задати зрозумілу назву та термін дії, що дає змогу розмежовувати ключі за призначенням: окремий для продуктивного середовища, окремий для тестового, окремий для конкретної інтеграції чи скрипта. У разі підозри на компрометацію будь-який ключ можна негайно відкликати, не зачіпаючи інших і не блокуючи весь обліковий запис. Відкликання реалізовано через позначку у базі даних, а не через фізичне видалення запису, що дозволяє зберігати журнал колись виданих ключів для подальшого аудиту безпеки. Така гнучкість керування є істотною перевагою порівняно з єдиним статичним обліковим записом і відповідає сучасним практикам побудови захищених програмних інтерфейсів.

3.4 Розробка вебінтерфейсу

Клієнтська частина оформлена як односторінковий застосунок, що повністю працює в браузері й взаємодіє із сервером лише через API. Усі переходи між сторінками відбуваються без перезавантаження, що дає відчуття плавної роботи, типове для сучасних вебсервісів. Сервер, у свою чергу, нічого не знає про конкретні сторінки й шаблони відображення – він обмежується наданням даних і виконанням операцій над ними, що добре узгоджується з архітектурою, описаною у попередніх підрозділах.

Структуру проєкту побудовано модульно. Окремо виділено каталог із готовими екранами, що відповідають маршрутам у браузері, окремо розташовані допоміжні елементи функціоналу на кшталт модального вікна входу, які можуть перевикористовуватися між екранами, і ще окремо шар взаємодії з програмним інтерфейсом сервера. Стилі для кожного екрана зберігаються поруч із його кодом і ізольовані одне від одного, що унеможливлює випадковий вплив стилів одного екрана на вигляд іншого. Такий розподіл відповідальностей робить кодову базу зрозумілою для людини, що бачить її вперше, а також спрощує внесення змін: щоб змінити одну сторінку, не потрібно перейматися наслідками для решти системи. Маршрутизація між екранами реалізована на стороні клієнта засобами бібліотеки react-router-dom.

Автентифікація на боці клієнта побудована довкола JSON Web Token, який сервер видає у відповідь на успішний вхід або реєстрацію. Отриманий токен клієнт зберігає у локальному сховищі браузера. Цей вибір зумовлений тим, що сховище переживає закриття вкладки й перезапуск браузера, а отже повторно входити при кожному відвідуванні сайту не потрібно. Поки токен лишається дійсним, він автоматично додається до кожного запиту, що проходить через базовий клієнт, описаний у попередньому підрозділі.

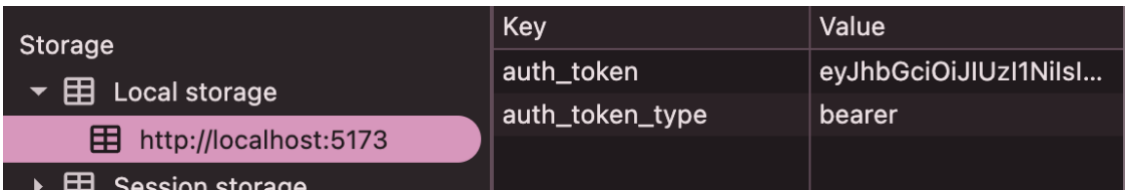


Рис. 3.7 – збережений токен у локальному сховищі браузера

Заголовок сайту показує різний набір кнопок залежно від того, увійшов користувач чи ні. Деякі сторінки взагалі недоступні, якщо не ввійти в систему. Усі ці місця мають реагувати на вхід і вихід однаково й миттєво, без перезавантаження сторінки. В основі клієнтського застосунку лежить компонентний підхід, який пропонує бібліотека React. Інтерфейс розглядається не як один великий шаблон, а як композиція невеликих самодостатніх компонентів, кожен з яких відповідає за окремий фрагмент сторінки – кнопку, форму, модальне вікно, особистий кабінет тощо. Кожен компонент описує, як він має виглядати залежно від свого внутрішнього стану та переданих йому даних, а React сам бере на себе обов'язок оновлювати потрібні частини сторінки при зміні цього стану. На практиці це означає, що розробник зосереджується на тому, яким має бути інтерфейс у кожен момент часу, а не на тому, як саме змінити його після події – стандартна проблема ручної маніпуляції DOM-деревом тут зникає. Компонентна модель добре поєднується з модульною структурою коду. Складніші екрани збираються з простіших компонентів, а ті, що використовуються в кількох місцях, виносяться окремо й перевикористовуються – наприклад, модальне вікно входу відкривається як зі стартової сторінки, так і з інших сторінок, де користувача треба попросити увійти. Це дозволяє уникнути дублювання й гарантує, що один і той самий елемент інтерфейсу скрізь поводиться однаково.

ВИСНОВОК ДО РОЗДІЛУ 3

У цьому розділі описано практичну реалізацію основних компонентів платформи: моделі класифікації листів, модулів обробки природної мови, серверної частини та клієнтського вебінтерфейсу.

Для задачі виявлення спаму було підготовлено комбінований набір даних на основі відкритого корпусу CEAS_08, доповнений синтетично згенерованими прикладами там, де реальних бракувало – насамперед транзакційними листами. На цьому датасеті донавчено модель DistilBERT, обрану через її компактність і високу швидкість. На тестовій вибірці з того ж розподілу модель досягла точності близько 99% (14 хибних передбачень), а на незалежному наборі даних модель досягла точності 90% та F1-міри 89%. Зниження точності на сторонніх даних пояснюється зміною домену, проте загальні показники підтверджують здатність моделі до узагальнення за межами тренувального розподілу.

Модуль анонімізації персональних даних побудовано на комбінації двох незалежних методів – пошуку структурованих даних за регулярними виразами та розпізнавання іменованих сутностей, що важко знайти регулярними виразами, за допомогою бібліотеки spaCy. Модуль вилучення ключових слів реалізовано на базі TextRank, що не потребує ні навчання, ні зовнішнього корпусу, ні мовних моделей, що зробило його придатним для роботи з текстами різних тематик без додаткової підготовки.

Серверна частина платформи реалізована з урахуванням двох різних категорій клієнтів – людей, що працюють через браузер, і програмних застосунків, що звертаються до API напряму. Для першої категорії використано короткоживучі JWT-токени, для другої – постійні API-ключі, які зберігаються у вигляді хешів і можуть бути окремо створені, названі та видалені користувачем. Такий розподіл дозволив адаптувати механізми безпеки під

реальні сценарії використання, а не нав'язувати єдиний підхід обом типам клієнтів.

Клієнтська частина побудована як односторінковий застосунок із використанням бібліотеки React і взаємодіє з сервером виключно через API. Компонентний підхід та модульна організація коду дозволили отримати кодову базу, у якій кожна сторінка є самодостатньою, а зміни в одній частині інтерфейсу не зачіпають інших. Стан автентифікації синхронізується між усіма частинами інтерфейсу та навіть між відкритими вкладками браузера, що забезпечує очікувану для сучасних вебсервісів поведінку.

У сукупності всі описані компоненти утворюють цілісну систему, готову до розгортання та подальшого розширення.

РОЗДІЛ 4

ОГЛЯД ТА ТЕСТУВАННЯ РОЗРОБЛЕНОЇ СИСТЕМИ

4.1 Огляд функціональних можливостей системи

4.1.1 Огляд сторінки Demo

Демонстраційна сторінка є початковою точкою взаємодії користувача із системою та призначена для того, щоб без реєстрації показати можливості платформи в дії. На ній розміщено єдине поле введення тексту та кнопку запуску аналізу, а результати роботи всіх модулів обробки природної мови відображаються одразу під формою. Такий підхід дозволяє потенційному користувачеві швидко оцінити, що саме вміє система, перш ніж створювати обліковий запис та отримувати API-ключ.

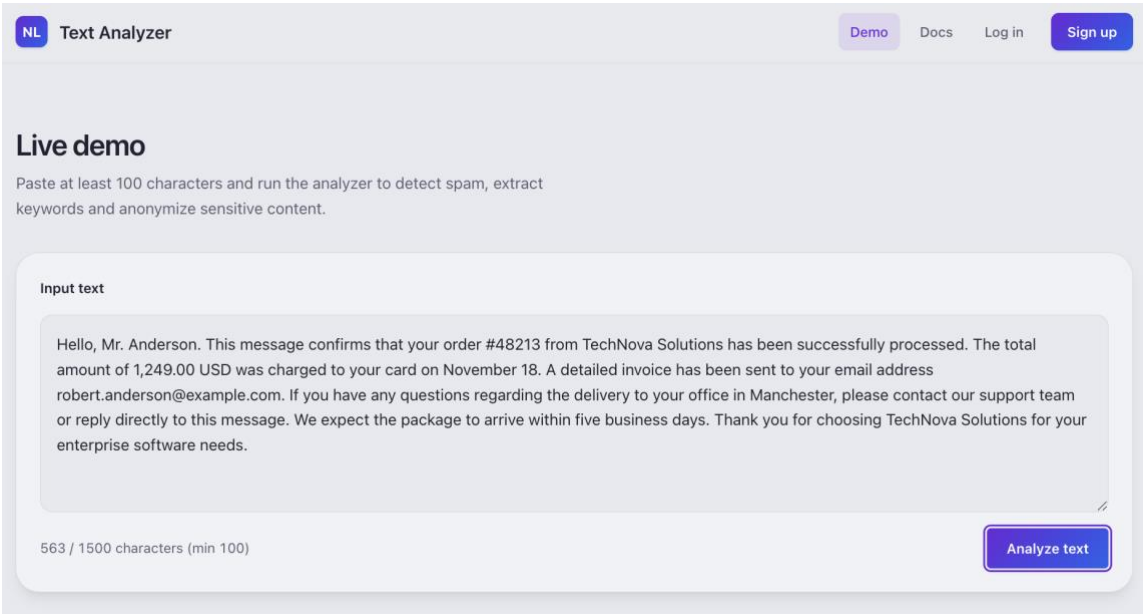


Рис. 4.1 – сторінка для перегляду функціоналу сервісу

Запустити аналіз можна лише тоді, коли в полі назбирається хоча б сто символів. Якщо робити аналіз для лише кількох слів, дуже ймовірно результати будуть не точними, оскільки даних замало.

Коли користувач тисне «Analyze text», текст відправляється на сервер, там відпрацьовують усі три модулі, і назад повертається спільний результат. На сторінці він розкладений на три картки – по одній на кожну функцію системи.

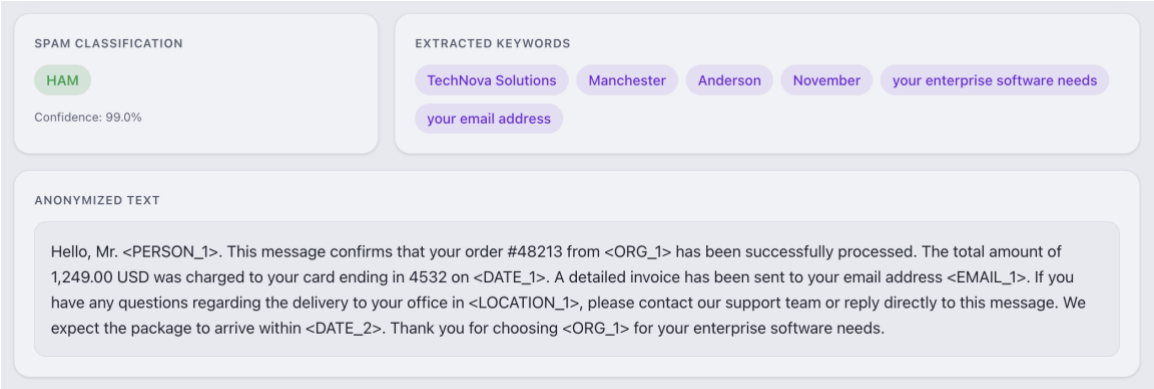


Рис. 4.2 – картки із результатами аналізу тексту

Перша картка, «Spam classification», показує вердикт моделі: спам чи звичайний лист, і поряд наскільки модель у цьому впевнена. На рисунку видно транзакційний лист – підтвердження оплати замовлення. Поверхово він схожий на спам: тут і сума, і назва компанії, і заклик звертатися в підтримку. Попри це модель упевнено відносить його до звичайних листів з оцінкою 99%. Саме такого результату й хотілося досягти, коли навчальний набір спеціально доповнювали транзакційними листами, оскільки без них подібні повідомлення регулярно потрапляли б у спам.

Друга картка, «Extracted keywords», містить ключові слова й фрази. Кожна фраза показана окремою позначкою, тож основні теми листа видно з першого погляду, не вчитуючись.

Третя картка, «Anonymized text», демонструє роботу модуля знеособлення. Тут добре видно, що система впоралася з обома типами персональних даних. Ім'я отримало мітку <PERSON_1>, назву компанії замінено на <ORG_1>, місто – на <LOCATION_1>, дати – на <DATE_1> і

<DATE_2>, а електронну адресу – на <EMAIL_1>. Водночас сума «1,249.00 USD» лишилася на місці: вона нікого не ідентифікує, тож і ховати її немає сенсу – інакше текст лише втратив би зміст без жодної користі для приватності.

Оскільки сторінка Demo відкрита без входу в систему, є очевидний ризик: хтось може почати використовувати сервіс за допомогою неї, щоб уникнути реєстрації й не створювати API-ключ. Щоб цього не сталося, на сторінці стоїть обмеження частоти запитів поверх Redis. На кожного клієнта заводиться лічильник із коротким часом життя, який збільшується з кожним зверненням; щойно він перевищує заданий поріг, сервер на якийсь час перестає приймати нові запити. Так сторінка залишається доступною, аби спокійно роздивитися можливості системи, але як заміна повноцінному доступу через API вона не є.

4.1.2 Огляд сторінок реєстрації та автентифікації

Реєстрація є першим кроком для користувача, який хоче отримати повноцінний доступ до системи та можливість генерувати API-ключі. Форма реєстрації містить три поля: ім'я користувача, електронну пошту та пароль (рис. 4.3). Після заповнення полів і натискання кнопки «Create account» дані надсилаються на сервер, який створює новий обліковий запис. Важливо, що пароль ніколи не зберігається у відкритому вигляді. На сервері він проходить хешування після чого до бази даних потрапляє лише отриманий хеш. Завдяки цьому навіть у разі компрометації сховища оригінальні паролі користувачів залишаються недоступними, оскільки відновити їх із хешу неможливо. Цей підхід узгоджується із загальним принципом безпеки системи, за яким чутливі дані такі як паролі, так і API-ключі зберігаються виключно у хешованому вигляді.

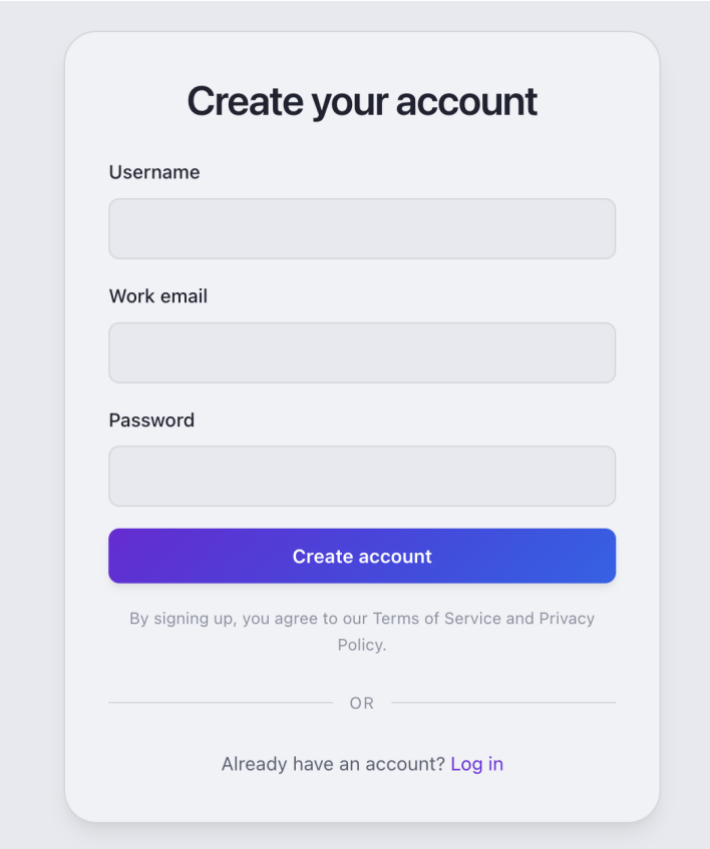


Рис. 4.3 – форма реєстрації

Перш ніж запит дійде до сервера, поля форми проходять базову перевірку безпосередньо у браузері. Зокрема, поле електронної пошти позначене як таке, що має відповідати формату адреси, тому при спробі надіслати форму з некоректним значенням браузер блокує відправлення й показує підказку. Така перевірка на боці клієнта дає користувачеві миттєвий зворотний зв'язок і відсіює очевидно хибні дані ще до звернення до сервера. Водночас вона не замінює серверну валідацію, а лише доповнює її: остаточну перевірку коректності та унікальності даних виконує сервер, оскільки клієнтську перевірку легко обійти. Після успішної реєстрації користувач автоматично переходить до особистого кабінету, де може керувати своїми API-ключами.

Для повернення до системи зареєстрований користувач використовує форму входу, що містить поля електронної пошти та пароля (рис. 4.4).

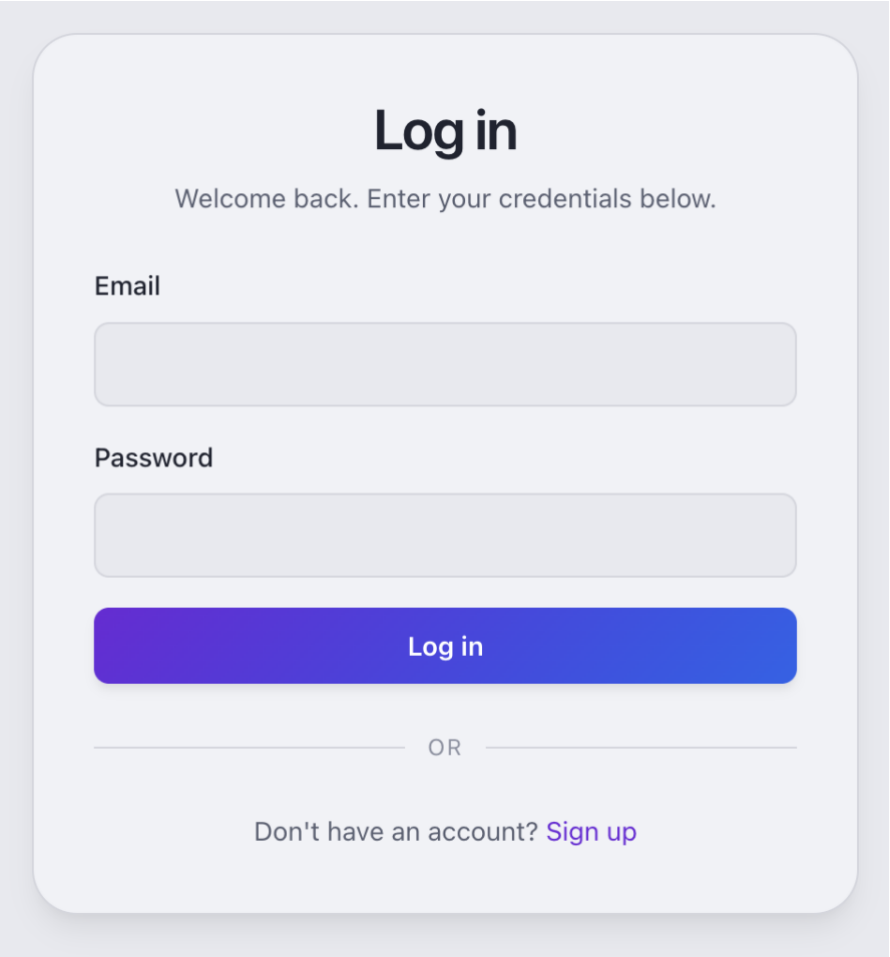


Рис. 4.4 – форма входу в систему

Після надсилання коректних даних сервер знаходить користувача за електронною поштою, хешує пароль та перевіряє чи він є правильним порівнюючи із хешем, що збережений у базі даних. Збіг підтверджує, що пароль правильний. У відповідь на успішний вхід сервер видає JWT-токен, який клієнт зберігає в локальному сховищі браузера й надалі додає до кожного запиту. Якщо ж облікові дані хибні, сервер повертає помилку, не розкриваючи, що саме було введено.

Обидві форми пов'язані між собою: зі сторінки входу можна перейти до реєстрації за посиланням «Sign up», а зі сторінки реєстрації повернутися до входу за посиланням «Log in». Це дозволяє користувачеві швидко перемкнутися на потрібну дію, не шукаючи відповідну сторінку вручну.

4.1.3 Огляд особистого кабінету користувача

Особистий кабінет – це закрита частина системи, доступна лише після входу. Саме тут користувач керує своїми API-ключами, через які програмні клієнти звертаються до модулів обробки тексту. Сторінка поділена на два логічні блоки: відомості про профіль та керування токенами.

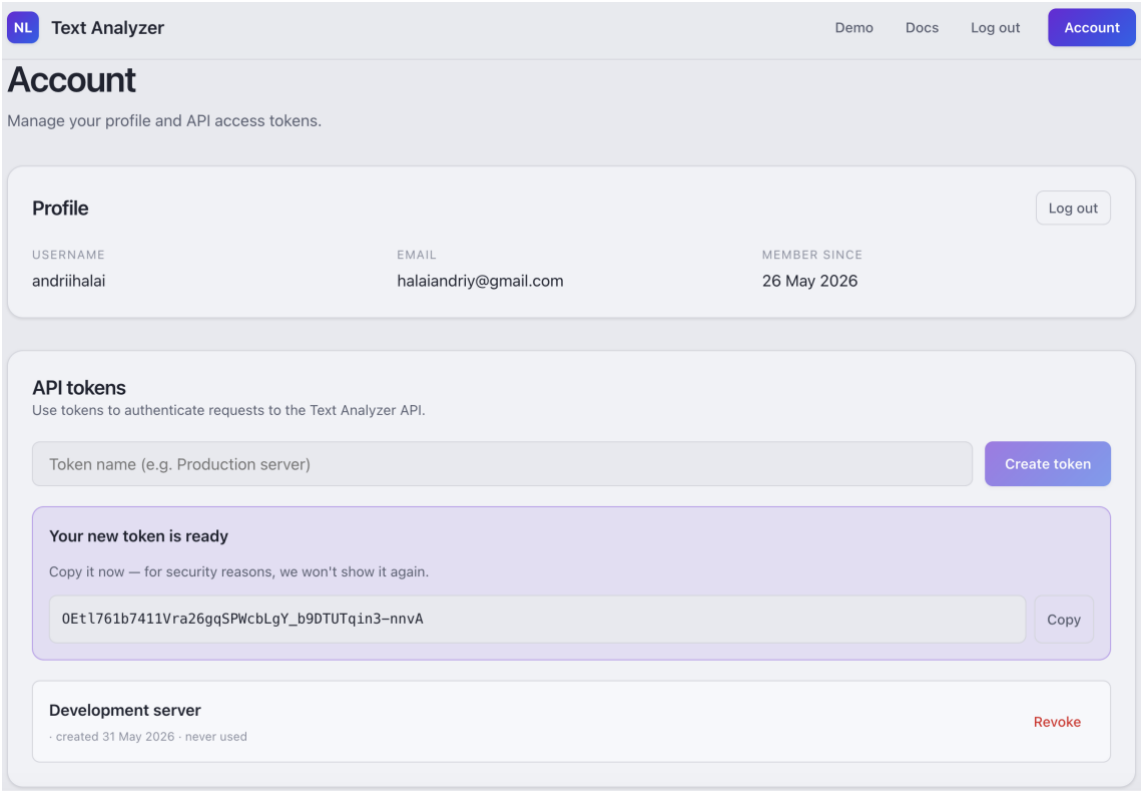


Рис. 4.5 – особистий кабінет користувача

Блок «Profile» показує основну інформацію про обліковий запис: ім'я користувача, електронну пошту та дату реєстрації. Тут же розміщено кнопку виходу із системи, що дублює відповідний пункт у верхній навігації.

Основну ж частину сторінки займає блок «API tokens». У його верхній частині є поле для назви та кнопка створення нового токена. Назва потрібна для того, щоб користувач міг розрізняти ключі за призначенням – наприклад, окремий токен для робочого середовища, окремий для тестового чи для конкретної інтеграції. Після натискання кнопки «Create token» сервер генерує новий ключ і одразу повертає його користувачу.

Згенерований ключ показується рівно один раз, про що прямо попереджає підказка в інтерфейсі: скопіювати його потрібно негайно, бо вдруге система його не покаже. Така поведінка не випадкова: на сервері зберігається не сам ключ, а лише його хеш, тому відновити оригінальне значення згодом неможливо навіть адміністратору. Це той самий принцип, за яким у системі зберігаються паролі. Для зручності поряд із ключем є кнопка копіювання.

Нижче розташований перелік уже створених токенів. Для кожного відображається його назва, дата створення та позначка про використання. Наприклад, «never used», якщо ключем ще жодного разу не зверталися до API. Навпроти кожного токена є кнопка «Revoke», яка дозволяє відкликати ключ. Важливо, що відкликання реалізовано не через видалення запису з бази, а через позначку про недійсність: завдяки цьому в системі зберігається історія колись виданих ключів для подальшого аудиту. Це також означає, що в разі підозри на компрометацію конкретний ключ можна вимкнути миттєво, не зачіпаючи інші токени й не блокуючи весь обліковий запис.

4.1.4 Огляд сторінки документації

Окрему сторінку відведено під документацію API. Її призначенням є надання розробнику, який інтегрує систему у свій застосунок, повний опис доступних можливостей сервісу. Наявність такої документації є однією з вимог до системи, оскільки основним клієнтом виступає не людина, а програма, і без чіткого опису інтерфейсу інтеграція стає невиправдано складною.

Сторінку побудовано за класичною для технічної документації схемою: ліворуч розміщено навігаційне меню з розділами, а праворуч – основний вміст обраного розділу. Меню розділено на розділи: спочатку йде опис сервісу, розділ автентифікації, керування обліковим записом і, нарешті, робота з API-ключами.

Такий поділ дозволяє швидко знайти потрібний розділ, не гортаючи всю сторінку.

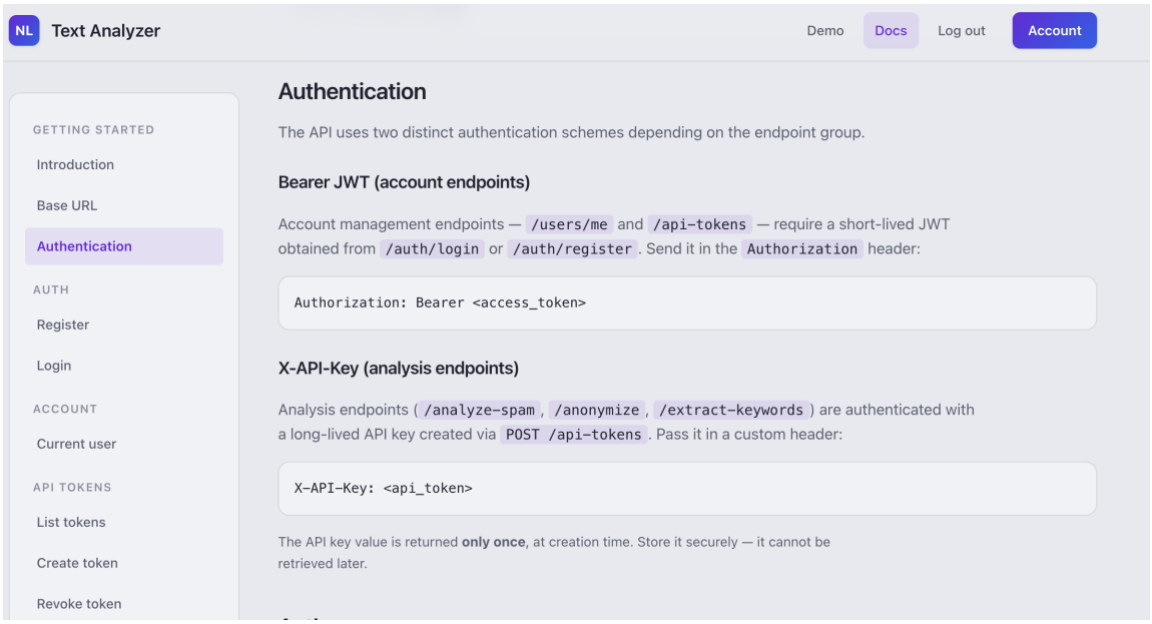


Рис. 4.6 – сторінка документації системи

Центральним для розуміння роботи з API є розділ «Authentication». У ньому пояснено, що система використовує дві різні схеми автентифікації залежно від групи методів сервера. Запити до методів керування обліковим записом потребують короткоживучого JWT-токена, отриманого під час входу чи реєстрації, який передається у заголовку Authorization у форматі Bearer. Натомість запити до методів аналізу тексту використовують довготриваліший API-ключ, який передається в окремому заголовку X-API-Key. Розмежування двох механізмів за різними заголовками робить логіку перевірки однозначною та відповідає різним сценаріям використання. Тут же наголошено, що значення API-ключа повертається лише один раз під час створення й надалі не може бути отримане повторно.

Для кожного метода в документації наведено його призначення, метод і шлях, перелік очікуваних параметрів та приклад відповіді. Завдяки цьому розробник отримує всю інформацію, потрібну для інтеграції, в одному місці, що відповідає закладеній у систему вимозі – надати зрозумілу документацію щодо використання API.

4.2 Тестування модулів обробки природної мови

4.2.1 Тестування модуля класифікації спаму

Окрім оцінювання на відкладеній вибірці, описаного в третьому розділі, модуль класифікації перевірявся вже у складі зібраного сервісу – через реальні HTTP-запити до локально запущеного API. Якщо тестування на наборі даних показує якість самої моделі, то перевірка через запити підтверджує, що модуль коректно інтегровано в систему: сервер приймає запит, передає текст у класифікатор і повертає відповідь у потрібному форматі. Для надсилання запитів використано клієнт Postman, що дає змогу зручно формувати тіло запиту та переглядати відповідь сервера.

Текст для класифікації передається у тілі запиту у форматі JSON, а у відповідь сервер повертає передбачений клас (ham або spam) та оцінку впевненості моделі.

Для перевірки роботи класифікатора в обидва боки використано два приклади. Перший є звичайним робочим листом із нагадуванням про зустріч, без жодних ознак спаму (рис. 4.7). Модель упевнено віднесла його до класу ham з оцінкою близько 0.999, що відповідає очікуваному результату.

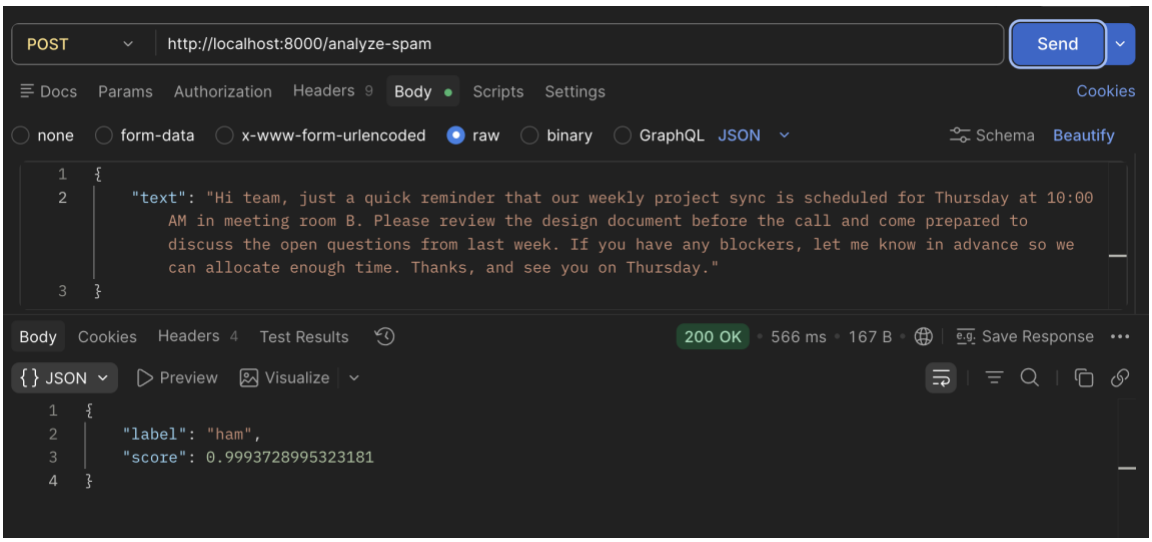


Рис. 4.7 – класифікація звичайного листа як ham

Другий приклад - це типове шахрайське повідомлення з обіцянкою грошового виграшу, штучною терміновістю та закликом перейти за підозрілим посиланням і ввести банківські дані (рис. 4.8). Цей текст модель класифікувала як spam, також з оцінкою близько 0.999.

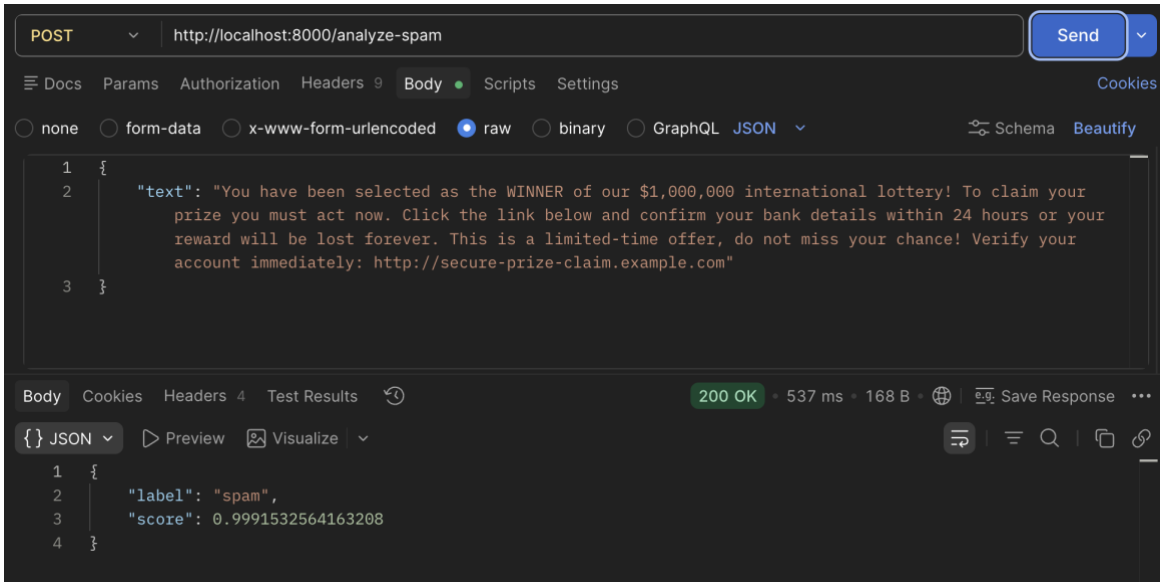


Рис. 4.8 – класифікація спам-листа

Результати перевірки через API узгоджуються з показниками, отриманими під час оцінювання моделі в третьому розділі, і підтверджують, що модуль класифікації правильно вбудовано в сервіс і він коректно класифікував обидва приклади.

4.2.2 Тестування модуля вилучення ключових слів

Перевірку цього модуля проведено через Postman за допомогою запитів до сервісу, запущеного локально. На вхід подається текст, а у відповідь повертається список знайдених ключових слів і фраз, де біля кожної вказано оцінку її важливості. Фрази впорядковано так, що найважливіші стоять першими.

Як приклад було взято текст про відновлювану енергетику. Його дібрано спеціально: головні поняття в ньому розкидані по всьому тексту, а не зосереджені в першому реченні. Завдяки цьому одразу видно, чи здатен модуль знаходити справді важливі терміни, а не просто хапати те, що трапилось на початку.

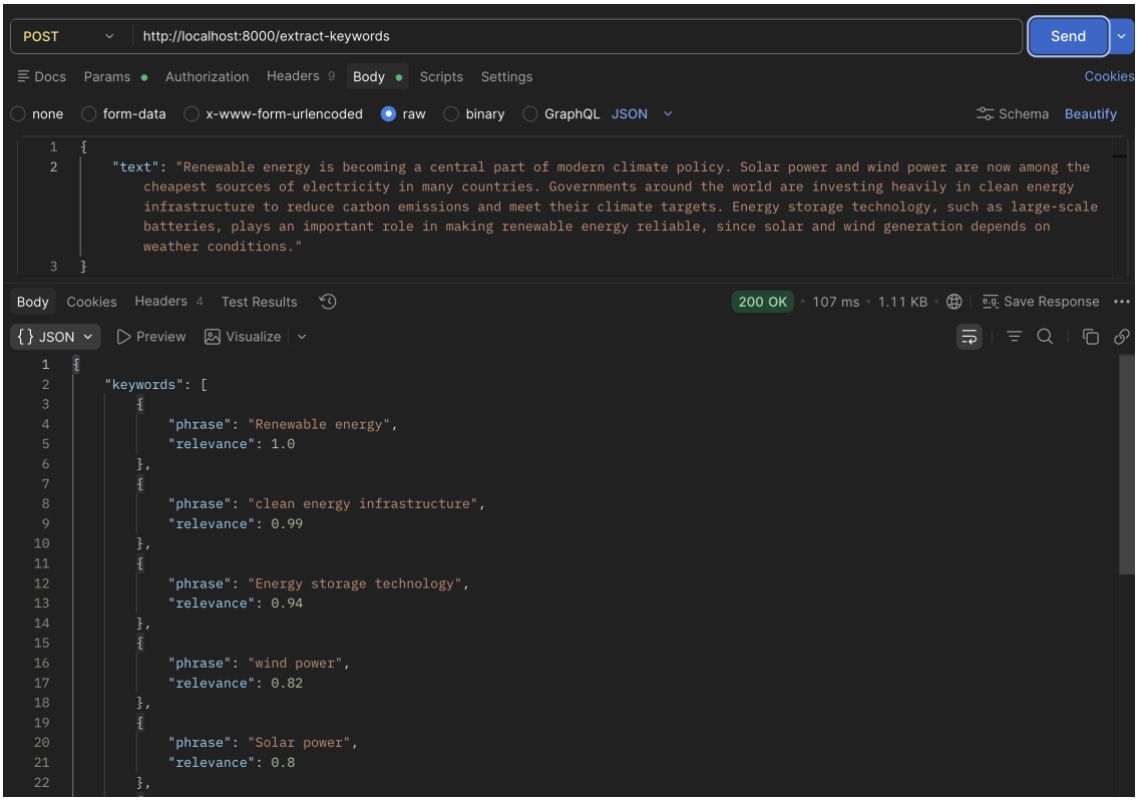


Рис. 4.9 – вилучення ключових слів

Отриманий результат показано на рис. 4.9. Нагору піднялася фраза «renewable energy», і це закономірно, адже саме навколо неї будується весь текст. Поряд із нею до переліку потрапили «clean energy infrastructure» та

«energy storage technology» – також змістовні терміни, що добре передають суть документа. Випадкових службових зворотів у результаті немає, як немає й повторів, коли та сама фраза різної довжини потрапляє в список кількома рядками. Сервер відповів кодом 200, тобто запит опрацьовано без помилок.

Підсумовуючи, перевірка показала, що модуль коректно працює у складі сервісу й правильно вловлює основний зміст тексту. При цьому він лишається легким: йому не потрібні ні навчання, ні зовнішній корпус, ні великі мовні моделі, тож застосовувати його можна до текстів різних тематик.

4.2.3 Тестування модуля анонімізації персональних даних

Модуль анонімізації перевірявся так само, через запити до локально запущеного сервісу за допомогою Postman. У відповідь на запит сервіс повертає дві частини: текст із заміненими даними і перелік виконаних замін.

Для тесту складено лист, що навмисно поєднує обидва типи персональних даних, які має розпізнавати модуль. Перша група – це структуровані дані з передбачуваним форматом, як-от електронна адреса, банківський рахунок у форматі IBAN та IP-адреса. Друга група охоплює неструктуровані дані, що розпізнаються лише за контекстом: імена людей, назву організації, географічну назву та дату. Завдяки цьому один запит дозволяє перевірити роботу обох методів виявлення, закладених у модуль.

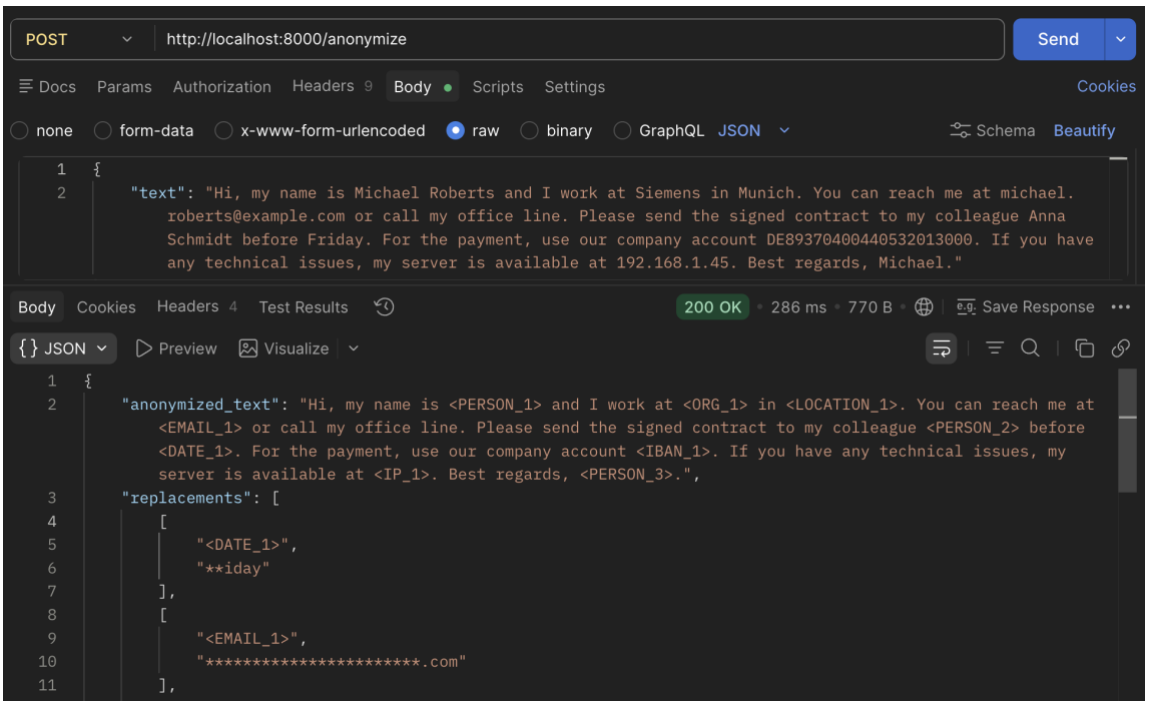


Рис. 4.10 – анонімізація персональних даних у тексті

Результат обробки наведено на рис. 4.10. Видно, що модуль коректно опрацював усі типи даних: структуровані дані було замінено через відповідні шаблони, а імена, організацію й місто через розпізнавання іменованих сутностей. Ім'я, що згадується в листі двічі, на початку та в підписі, отримало однакову позначку, завдяки чому зв'язність тексту збереглася. Кожному типу даних присвоєно власну категорію позначки, тож текст лишається читабельним і зрозумілим.

Друга частина відповіді, перелік замін, показує, якій позначці відповідало яке знайдене значення. При цьому оригінальні значення в ньому відображаються не повністю, а лише невеликим фрагментом із замаскованою рештою символів. Таке маскування дозволяє за потреби звірити коректність роботи модуля, не розкриваючи самих персональних даних, що узгоджується із загальним призначенням модуля, а саме унеможливити ідентифікацію особи в тексті.

Перевірка підтвердила, що модуль анонімізації коректно вбудовано в сервіс і він надійно знеособлює як структуровані, так і неструктуровані персональні дані, зберігаючи читабельність вихідного тексту.

ВИСНОВОК ДО РОЗДІЛУ 4

У четвертому розділі продемонстровано готову систему в роботі та перевірено, що всі її частини функціонують коректно.

Спершу розглянуто інтерфейс із боку користувача. Показано демонстраційну сторінку, де можна випробувати всі три модулі обробки тексту без реєстрації, сторінку з документацією для тих, хто інтегруватиме сервіс у свої застосунки, а також форми входу й реєстрації та особистий кабінет, через який користувач керує своїми API-ключами. Таким чином пройдено весь шлях користувача: від першого знайомства із сервісом до отримання ключа для доступу до API.

Далі перевірено роботу модулів обробки природної мови через реальні запити до локально запущеного сервісу. Класифікатор упевнено відрізнив звичайний лист від спаму, модуль анонімізації знеособив у тексті як структуровані дані, так і імена та назви, а модуль вилучення ключових слів коректно визначив основні теми документа. Усі запити опрацьовано без помилок, що підтверджує правильну інтеграцію модулів у сервіс.

ВИСНОВКИ

У ході виконання дипломного проєкту було розроблено систему інтелектуального аналізу текстових даних, що використовує методи обробки природної мови та надає програмний інтерфейс для виконання типових задач роботи з текстом.

У першому розділі розглянуто базові поняття, пов'язані з природною мовою та текстовими даними, а також проведено огляд наявних на ринку сервісів для їхньої автоматизованої обробки. На основі цього порівняння обґрунтовано доцільність створення рішення з відкритим вихідним кодом, де можна бачити як обробляються дані користувачів, та придатного до розгортання на власній інфраструктурі організації.

У другому розділі опрацьовано теоретичні основи обробки природної мови та обрано технологічний стек майбутньої системи. За результатами порівняння інструментів бібліотекою обробки тексту обрано spaCy, серверним фреймворком – FastAPI, системою керування базами даних – PostgreSQL, а для контролю кількості запитів користувачів також додано Redis.

У третьому розділі описано практичну реалізацію основних компонентів платформи. Для виявлення спаму підготовлено комбінований набір даних на основі корпусу CEAS_08, що був доповнений синтетичними прикладами, і на ньому донавчено модель DistilBERT. На незалежному наборі даних модель досягла точності 90% та F1-міри 89%; зміна домену набору даних не завадила моделі зберегти здатність класифікувати спам за межами тренувального набору даних. Реалізовано модуль анонімізації персональних даних за допомогою поєднання регулярних виразів та розпізнавання іменованих сутностей, а також модуль вилучення ключових слів. Серверну частину побудовано з урахуванням двох категорій клієнтів із застосуванням JWT-токенів та API-ключів, а клієнтський інтерфейс реалізовано як односторінковий вебсайт.

У четвертому розділі продемонстровано роботу готової системи та перевірено коректність її функціонування. Розглянуто інтерфейс із боку користувача, зокрема демонстраційну сторінку, форми реєстрації й входу, особистий кабінет та сторінку документації. Роботу всіх трьох модулів обробки природної мови перевірено через реальні запити до локально запущеного сервісу.

У результаті виконання роботи створено цілісну систему інтелектуального аналізу текстових даних з відкритим вихідним кодом, що вирішує задачі виявлення спаму, анонімізації персональних даних та вилучення ключових слів. Розроблене рішення надає зручний програмний інтерфейс, відкритий вихідний код та готове до розгортання на власній інфраструктурі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Природна мова — тлумачення. *Горох — онлайн-бібліотека словників для всебічного та зручного дослідження української мови*. URL: https://goroh.pp.ua/Фразеологія/природна%20мова (дата звернення: 16.05.2026).
2. Cloud Natural Language. *Google Cloud*. URL: https://cloud.google.com/natural-language?hl=uk (дата звернення: 16.05.2026).
3. TextRazor — Extract Meaning from your Text. *TextRazor*. URL: https://www.textrazor.com/ (дата звернення: 16.05.2026).
4. Supported Languages and Features. *Lexalytics*. URL: https://www.lexalytics.com/technology/features/ (дата звернення: 16.05.2026).
5. Byte-Pair Encoding tokenization. *Hugging Face — LLM Course*. URL: https://huggingface.co/learn/llm-course/chapter6/5 (дата звернення: 16.05.2026).
6. Summary of the tokenizers. *Hugging Face — Transformers documentation*. URL: https://huggingface.co/docs/transformers/tokenizer_summary (дата звернення: 16.05.2026).
7. Krantz T., Jonker A. What Is Cosine Similarity?. *IBM*. URL: https://www.ibm.com/think/topics/cosine-similarity (дата звернення: 16.05.2026).
8. Cosine similarity. *Engati*. URL: https://www.engati.ai/glossary/cosine-similarity (дата звернення: 16.05.2026).
9. From Words to Vectors: Understanding Word2Vec in NLP. *Zilliz Vector Lakebase for Enterprise AI, Powered by Milvus*. URL: https://zilliz.com/glossary/word2vec (дата звернення: 16.05.2026).

10. Othman N., Faiz R., Smaïli K. Enhancing Question Retrieval in Community Question Answering Using Word Embeddings. *Proceedings of the 23rd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES2019)* : матеріали Міжнар. наук. конф., м. Будапешт, 4–6 вересня 2019. С. 4. URL: https://www.researchgate.net/publication/333041167_Enhancing_Question_Retrieval_in_Community_Question_Answering_Using_Word_Embeddings (дата звернення: 17.05.2026).
11. GeeksforGeeks. Word Embedding using Word2Vec. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/python/python-word-embedding-using-word2vec/ (дата звернення: 17.05.2026).
12. Recurrent neural networks. *Dive into Deep Learning*. URL: https://d2l.ai/chapter_recurrent-neural-networks/rnn.html (дата звернення: 17.05.2026).
13. Long Short-Term Memory (LSTM). *MathWorks*. URL: https://www.mathworks.com/discovery/lstm.html (дата звернення: 17.05.2026).
14. Sequence-to-sequence модель. *Keras — Documentation*. URL: https://keras.io/examples/nlp/lstm_seq2seq/ (дата звернення: 17.05.2026).
15. The Transformer model family. *Hugging Face — Transformers documentation*. URL: https://huggingface.co/docs/transformers/model_summary (дата звернення: 17.05.2026).
16. SpaCy: Everything you need to know. *spaCy — Usage*. URL: https://spacy.io/usage/spacy-101 (дата звернення: 17.05.2026).
17. SpaCy library architecture. *spaCy — API documentation*. URL: https://spacy.io/api (дата звернення: 17.05.2026).
18. FastAPI. *FastAPI Documentation*. URL: https://fastapi.tiangolo.com/ (дата звернення: 17.05.2026).

19. Pydantic. *Pydantic Documentation*. URL: https://docs.pydantic.dev/latest/ (дата звернення: 17.05.2026).
20. Django documentation. *Django*. URL: https://docs.djangoproject.com/en/stable/ (дата звернення: 18.05.2026).
21. Redis documentation. *Redis*. URL: https://redis.io/docs/latest/ (дата звернення: 18.05.2026).
22. Anderson B., Nicholson B. SQL vs. NoSQL Databases: What's the Difference? *IBM*. URL: https://www.ibm.com/think/topics/sql-vs-nosql (дата звернення: 18.05.2026).
23. PostgreSQL Documentation. *PostgreSQL*. URL: https://www.postgresql.org/docs/ (дата звернення: 18.05.2026).
24. Introduction to JSON Web Tokens. *JWT.io*. URL: https://jwt.io/introduction (дата звернення: 18.05.2026).
25. React documentation. *React*. URL: https://react.dev/ (дата звернення: 18.05.2026).
26. Angular documentation. *Angular*. URL: https://angular.dev/ (дата звернення: 18.05.2026).
27. Vue.js documentation. *Vue.js*. URL: https://vuejs.org/ (дата звернення: 18.05.2026).
28. Датасет CEAS-08. *Kaggle*. URL: https://www.kaggle.com/datasets/doryanay/ceas-08 (дата звернення: 18.05.2026).
29. DistilBERT. *Hugging Face – Transformers documentation*. URL: https://huggingface.co/docs/transformers/model_doc/distilbert (дата звернення: 19.05.2026).
30. Large Language Models: DistilBERT – Smaller, Faster, Cheaper and Lighter. *Towards Data Science*. URL: https://towardsdatascience.com/distilbert-11c8810d29fc/ (дата звернення: 19.05.2026).

31. tanquangduong/spam-detection-dataset-splits. *Hugging Face — Datasets*. URL: https://huggingface.co/datasets/tanquangduong/spam-detection-dataset-splits (дата звернення: 19.05.2026).
32. Domain shift. *Statlect*. URL: https://www.statlect.com/machine-learning/domain-shift (дата звернення: 19.05.2026).
33. Mihalcea R., Tarau P. TextRank: Bringing Order into Text. *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing (EMNLP 2004)*, Barcelona, Spain, July 2004. P. 404–411. URL: https://aclanthology.org/W04-3252/ (дата звернення: 19.05.2026).

ДОДАТОК А

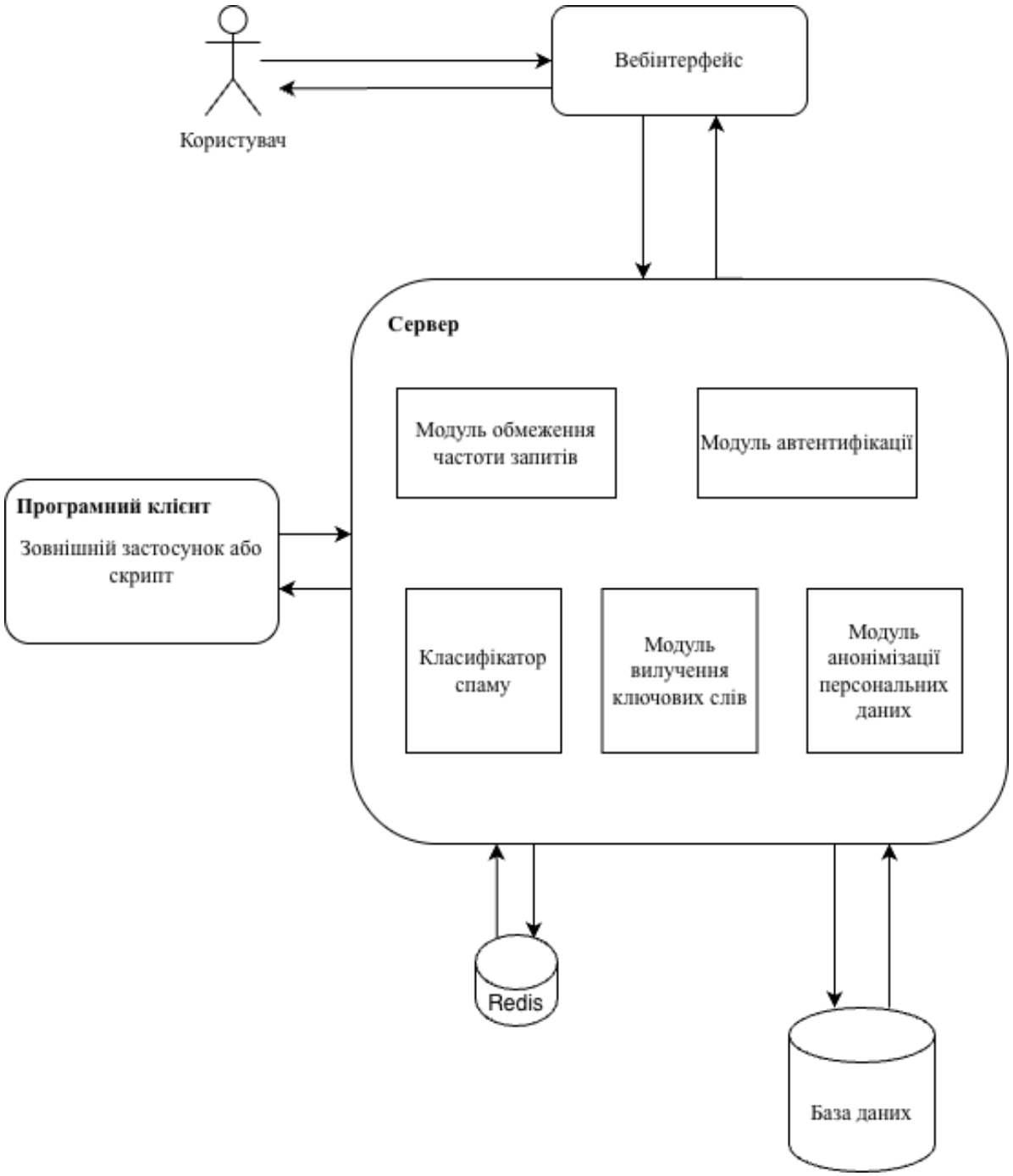
Система інтелектуального аналізу текстових даних із використанням методів обробки природної мови (NLP)

Структурна схема системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ – 2026



					ІАЛЦ.467200.004 Д1	
		№ докум.	Підпис	Дата		
Розробив		Галай А. В.			**Система інтелектуального аналізу текстових даних із використанням методів обробки природної мови (NLP)** Структурна схема системи	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Трочун Є. В.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21
Реценз.		Шимкович В. М.				
Н. Контр.		Коренко Д. В.				
Затвердив		Волокита А. М.				

ДОДАТОК Б

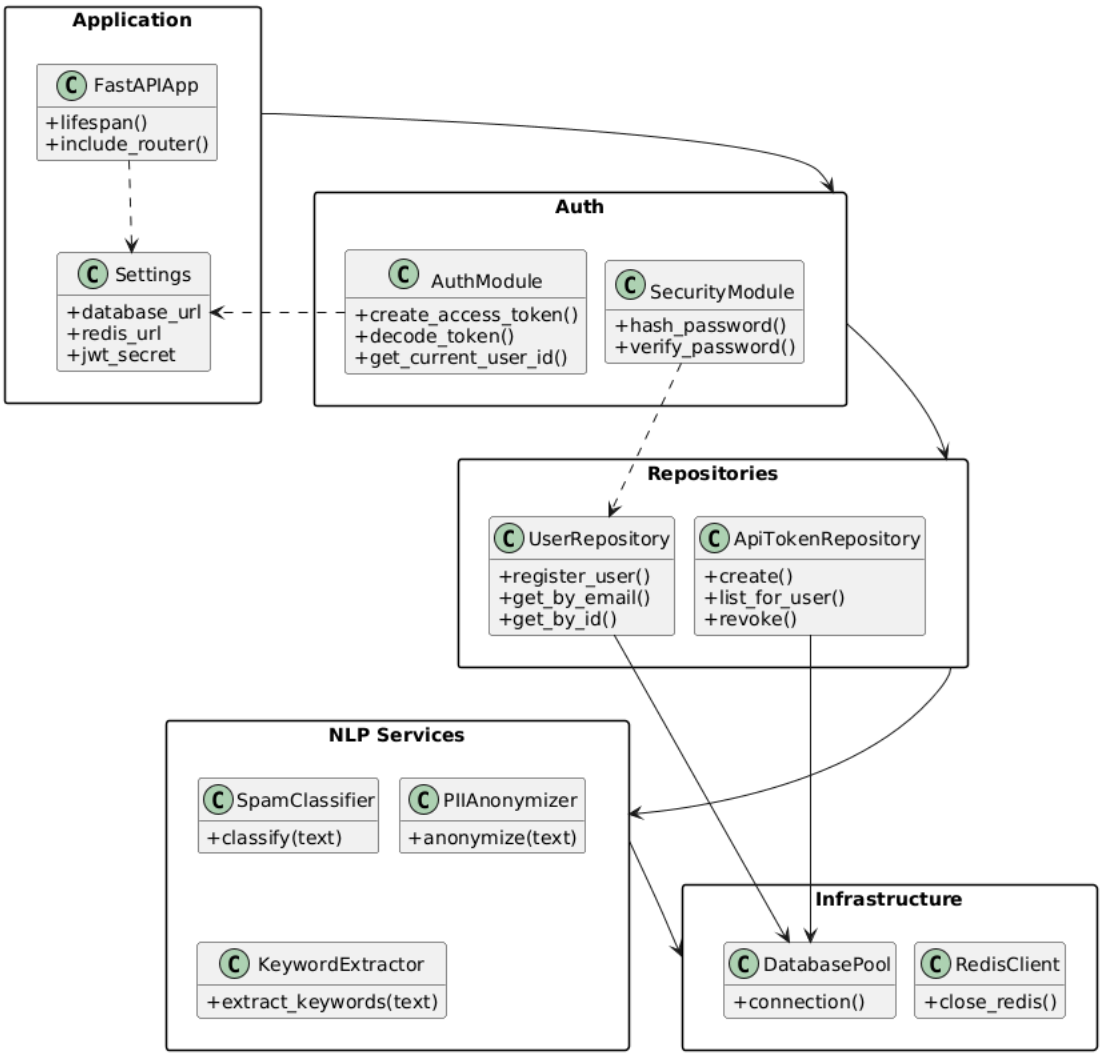
Система інтелектуального аналізу текстових даних із використанням методів обробки природної мови (NLP)

Функціональна схема (діаграма класів)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ – 2026



					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Галай А. В.			**Система інтелектуального аналізу текстових даних із використанням методів обробки природної мови (NLP)** Функціональна схема (діаграма класів)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Трочун Є. В.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21
Реценз.		Шимкович В. М.				
Н. Контр.		Коренко Д. В.				
Затвердив		Волокита А. М.				

ДОДАТОК В

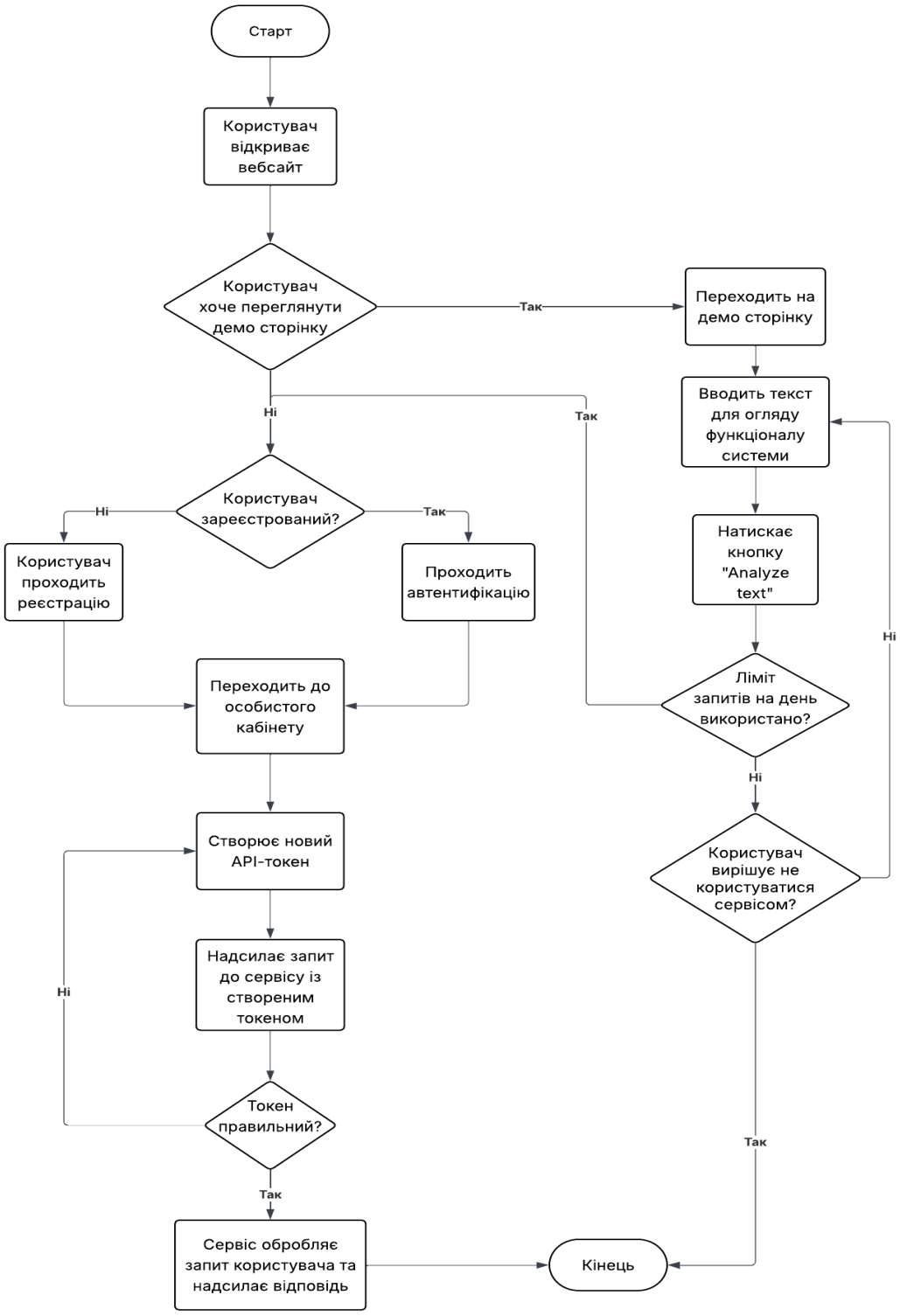
Система інтелектуального аналізу текстових даних із використанням методів обробки природної мови (NLP)

Принципова схема (блок-схема алгоритму)

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ – 2026



					ІАЛЦ.467200.006 ДЗ
		№ докум.	Підпис	Дата	
Розробив		Галай А. В.			**Система інтелектуального аналізу текстових даних із використанням методів обробки природної мови (NLP)** Принципова схема (блок-схема алгоритму)
Перевірив		Трочун Є. В.			
Реценз.		Шимкович В. М.			
Н. Контр.		Коренко Д. В.			
Затвердив		Волокита А. М.			

Літ.	Аркуш	Аркушів
	1	1

КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21

ДОДАТОК Г

Система інтелектуального аналізу текстових даних із використанням методів обробки природної мови (NLP)

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 1

Київ – 2026

					ІАЛЦ.467200.007 Д4		
		№ докум.	Підпис	Дата			
Розробив		Галай А. В.			**Система інтелектуального аналізу текстових даних із використанням методів обробки природної мови (NLP)** Текст програмного коду	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Трочун Є. В.					
Реценз.		Шимкович В. М.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21	
Н. Контр.		Коренко Д. В.					
Затвердив		Волокита А. М.					