

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ  
імені ІГОРЯ СІКОРСЬКОГО»**

**Інститут прикладного системного аналізу**

**Кафедра Системного проектування**

До захисту допущено:

Завідувач кафедри

\_\_\_\_\_ Вадим МУХІН

«\_\_» \_\_\_\_\_ 20\_\_ р.

**Дипломна робота**

**на здобуття ступеня бакалавра**

**за освітньо-професійною програмою «Інтелектуальні сервіс-орієнтовані  
розподілені обчислювання»**

**спеціальності 122 «Системний аналіз»**

**на тему: «Алгоритми та програмна реалізація методів глибокого навчання  
для класифікації текстових документів»**

Виконав (-ла):

студент (-ка) IV курсу, групи ДА-92

Мінюк Валерія Русланівна \_\_\_\_\_

Керівник:

Професор,

Рогоза Валерій Станіславович \_\_\_\_\_

Консультант з нормконтролю:

доцент, к.т.н.

Кирюша Б.А. \_\_\_\_\_

Рецензент:

проф., д.т.н.

Тимощук Оксана Леонідівна \_\_\_\_\_

Засвідчую, що у цій дипломній роботі  
немає запозичень з праць інших авторів  
без відповідних посилань.

Студент (-ка) \_\_\_\_\_

Київ – 2023 року

**Національний технічний університет України**  
**«Київський політехнічний інститут імені Ігоря Сікорського»**  
**Інститут прикладного системного аналізу**  
**Кафедра Системного проектування**

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – **122 «Системний аналіз»**

Освітньо-професійна програма «Інтелектуальні сервіс-орієнтовані розподілені обчислювання»

ЗАТВЕРДЖУЮ

Завідувач кафедри

\_\_\_\_\_ Вадим МУХІН

«\_\_» \_\_\_\_\_ 20\_\_ р.

**ЗАВДАННЯ**  
**на дипломну роботу студенту**  
**Мінюк Валерії Русланівні**

1. Тема роботи «Алгоритми та програмна реалізація методів глибокого навчання для класифікації текстових документів», керівник роботи Рогоза Валерій Станіславович, затверджені наказом по університету від «\_\_» \_\_\_\_\_ 20\_\_ р. № \_\_\_\_\_

2. Термін подання студентом роботи

3. Вихідні дані до роботи

1. Операційна система MacOS Big Sur
2. Частота процесора 1.6 ГГц
3. Мова програмування Python
4. Середовище розробки – PyCharm
5. Бібліотеки, що використовувалися: NumPy, Pandas, Sklearn, Matplotlib, NLTK, Keras, Time, Matplotlib, Ssl, Seaborn

## 4. Зміст роботи

1. Вступ.
2. Постановка задачі проектування.
3. Методи класифікації тексту.
4. Обробка даних.
5. Методи та алгоритми векторного представлення тексту.
6. Створення продукту для класифікації тексту.
7. Функціонально-вартісний аналіз програмного забезпечення.
8. Висновки.

## 5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)

1. Презентація

6. Консультанти розділів роботи<sup>1\*</sup>

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Рощина Н.В.		

7. Дата видачі завдання \_\_\_\_\_

## Календарний план

з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	10.02.2022	
2	Дослідження поняття дизайн операцій	10.03.2022	
3	Проведення аналізу щодо актуальності та важливості класифікації тексту, а також методів реалізації, що використовуються	29.03.2022	
4	Проведення порівняльного аналізу методів векторизації тексту, що застосовуються	10.04.2022	
5	Проведення аналізу методі	15.04.2022	

<sup>1\*</sup> Якщо визначені консультанти. Консультантом не може бути зазначено керівника дипломної роботи.

	попередньої обробки даних		
6	Розробка моделі глибокого навчання для класифікації тексту	30.04.2022	
7	Тестування розробленої моделі	15.05.2022	
8	Виправлення помилок, знайдених при тестуванні	23.05.2022	

Студент

Валерія МІНЮК

Керівник

Валерій РОГОЗА

## АНОТАЦІЯ

бакалаврської дипломної роботи Мінюк Валерії Русланівни на тему  
«Алгоритми та програмна реалізація методів глибокого навчання для  
класифікації текстових документів»

В даній роботі досліджуються методи глибокого навчання, його застосування для задачі класифікації текстових документів та порівняння його ефективності з базовим алгоритмом машинного навчання, у виді дерева рішень.

Задача класифікації тексту є актуальною в сучасному інформаційному суспільстві з урахуванням розміру та складності текстових даних, що генеруються щодня. Швидкість і точність обробки цих даних мають вирішальне значення для різних галузей, таких як медицина, фінанси, соціальні мережі, медіа та багато інших.

Метою дипломної роботи є дослідження ефективності методів глибокого навчання, зокрема згорткових нейронних мереж, у контексті класифікації текстових документів. Розробка універсальної моделі, заснованої на нейронних мережах, може сприяти покращенню результатів класифікації тексту та забезпечити високу точність і швидкість обробки даних.

Результатом дипломної роботи є програмний продукт для класифікації текстових документів, що написаний мовою програмування Python з використанням моделі згорткової нейронної мережі.

Використання даного програмного продукту дозволяє класифікувати документи, незалежно від їх тематики, отже може знайти застосування в різноманітних сферах, від класифікації повідомлень у пошті, до класифікації тем наукових статей.

Підсумовуючи, дослідження актуальності задачі класифікації тексту та використання глибокого навчання для досягнення кращих результатів є

важливим кроком у напрямку поліпшення обробки та аналізу текстових даних у різних сферах.

**Загальний обсяг роботи 97 с., 36 рис., 6 таблиць, 2 додатки, 27 джерела.**

**Ключові слова:** нейронні мережі, класифікація, глибоке навчання, машинне навчання, алгоритми.

## ABSTRACT

bachelor's thesis of Miniuk Valeriia Ruslanivna on “Algorithms and software implementation of deep learning for the classification of text documents”

This paper investigates deep learning methods, its application to the task of text document classification, and compares its effectiveness with the basic machine learning algorithm in the form of a decision tree.

The task of text classification is relevant in today's information society, given the size and complexity of text data generated on a daily basis. The speed and accuracy of processing this data is crucial for various industries such as medicine, finance, social networks, media, and many others.

The purpose of this thesis is to study the effectiveness of deep learning methods, in particular convolutional neural networks, in the context of text document classification. The development of a universal model based on neural networks can help improve text classification results and ensure high accuracy and speed of data processing.

The result of the thesis is a software product for text document classification written in the Python programming language using a convolutional neural network model.

The use of this software product allows it to classify documents regardless of their subject matter, so it can be used in a variety of areas, from classifying messages in mail to classifying topics of scientific articles.

To summarize, the study of the relevance of the text classification task and the use of deep learning to achieve better results is an important step towards improving the processing and analysis of text data in various fields.

**The total volume of work is 97 pages, 36 figures, 6 tables, 2 appendices, 27 sources.**

**Keywords:** neural networks, classification, deep learning, machine learning, algorithms.

# ЗМІСТ

**ЗМІСТ** .....

**ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ**.....

**ВСТУП**.....

**1 МЕТОДИ КЛАСИФІКАЦІЇ ТЕКСТУ** .....

1.1 Наївний Байєсівський класифікатор .....

1.2 Метод опорних векторів.....

1.3 Глибоке навчання.....

1.3.1 Відмінність між машинним та глибоким навчанням .....

1.3.2 Нейронні мережі .....

1.3.4 Згорткові нейронні мережі.....

1.4 Ефективніший метод для класифікації тексту .....

1.5 Висновки до розділу 1 .....

**2 ОБРОБКА ДАНИХ** .....

2.1 Роль обробки даних в проекті.....

2.2 Основні задачі попередньої обробки даних .....

2.2.1 виправлення скорочень .....

2.2.2 Приведення до нижнього регістру .....

2.2.3 Видалення пунктуації.....

2.2.4 Видалення стоп слів.....

2.2.5 Стемінг та лематизація .....

2.3 Висновки до розділу 2 .....

**3 МЕТОДИ ТА АЛГОРИТМИ ВЕКТОРНОГО ПРЕДСТАВЛЕННЯ**

<b>ТЕКСТУ .....</b>	
3.1 Роль та значення векторизації тексту .....	
3.1.1 Bag-of-Words .....	
3.1.2 TF-IDF .....	
3.1.3 Word2Vec .....	
3.1.4 GloVe .....	
3.2 Висновки до розділу 3 .....	
<b>4 СТВОРЕННЯ ПРОДУКТУ ДЛЯ КЛАСИФІКАЦІЇ ТЕКСТУ .....</b>	
4.1 Вибір бібліотеки для реалізації нейронної моделі .....	
4.2 Перенавчання та базова модель .....	
4.3 Вибір мови програмування для створення проекту .....	
4.4 Keras .....	
4.5 Аналіз даних .....	
4.6 Тестування розробленої програми .....	
4.7 Висновки до розділу 4 .....	
<b>5 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ .....</b>	
5.1 Постановка завдання проектування .....	
5.2 Обґрунтування функцій програмного продукту .....	
5.3 Варіанти реалізації основних функцій .....	
5.4 Обґрунтування системи параметрів програмного продукту .....	
5.4.1 Кількісна оцінка параметрів .....	
5.5 Аналіз експертного оцінювання параметрів .....	
5.6 Економічний аналіз варіантів розробки програмного продукту .....	
5.7 Висновки до розділу 5 .....	
<b>ВИСНОВКИ ПО РОБОТІ .....</b>	
<b>ПЕРЕЛІК ПОСИЛАНЬ .....</b>	

**ДОДАТОК А**.....

**ДОДАТОК Б**.....

## ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

Машинне навчання (Machine Learning, ML) - це галузь штучного інтелекту, яка займається розробкою та застосуванням алгоритмів, які дозволяють комп'ютерам самостійно вчитися та вдосконалювати свою продуктивність на основі даних без явного програмного втручання.

Класифікація(Classification) - це процес розподілу об'єктів (даних, текстів, зображень тощо) на певні категорії або класи на основі їх характеристик або властивостей. Це важний метод аналізу даних, який дозволяє автоматично визначати приналежність об'єктів до певних категорій заздалегідь визначених класів.

Згортова нейронна мережа (Convolutional Neural Network, CNN) - це тип нейронної мережі. Вона використовує спеціалізовані шари згортки та пулінгу для ефективного виявлення особливостей та шаблонів в даних.

Дерево рішень (Decision Tree) - це графічна модель прийняття рішень, яка використовується для вирішення задач класифікації та регресії. Дерево рішень представляє собою ієрархічну структуру, де кожен внутрішній вузол представляє рішення, а кожен листовий вузол - категорію чи значення.

Keras - це високорівнева бібліотека для глибокого навчання, яка працює поверх нижчорівневих фреймворків, таких як TensorFlow або Theano.

NLTK (Natural Language Toolkit) - це бібліотека для обробки природної мови (Natural Language Processing, NLP) у середовищі програмування Python.

Python - це високорівнева, інтерпретована мова програмування, яка підтримує об'єктно-орієнтований підхід.

TF-IDF (Term Frequency-Inverse Document Frequency) - це статистичний метод, що використовується для оцінки важливості термів у колекції текстових документів.

GloVe (Global Vectors for Word Representation) - це модель для отримання векторних представлень слів, яка використовується для представлення семантичних схожостей між словами у векторному просторі.

## ВСТУП

Існує безліч алгоритмів класифікації тексту [4]. Необхідність в класифікації тексту виникла з появою великого обсягу документів та текстових даних, що потребували організації, категоризації та аналізу.

З розвитком технологій, особливо Інтернету та цифрових медіа, з'явилася можливість створювати, зберігати та поширювати великі обсяги текстової інформації [3]. Це означає, що виникає потреба в ефективних методах обробки та аналізу цих даних для отримання цінної інформації.

Зі зростанням кількості текстових документів, включаючи веб-сторінки, електронні книги, статті, електронні листи та інші джерела, стало складніше знаходити та організовувати необхідну інформацію [5]. Класифікація тексту дозволяє автоматично розподіляти документи за категоріями та допомагає при пошуку конкретної інформації.

З розширенням даних та зростанням обчислювальних можливостей виникає потреба в автоматичному аналізі текстових даних для отримання інсайтів, розуміння тенденцій, класифікації документів за певними критеріями та іншій обробці [1], [2].

Класифікація тексту має широке застосування у багатьох галузях, таких як машинне навчання, обробка природної мови, інформаційний пошук, соціальні медіа, фінанси, медицина та багато інших [3]. Ці галузі потребують ефективних методів класифікації тексту для здійснення автоматичних рішень та аналізу текстових даних.

Перші алгоритми для класифікації тексту з'явилися у 1950-х роках [4]. Один з найвідоміших алгоритмів - "Bag of Words" (мішок слів), був запропонований у 1954 році [6]. Цей алгоритм представляв текстові документи у вигляді множини слів та розраховував частоту входження кожного слова в документи.

З цього часу пройшло більш ніж 50 років розвитку технології і з'явилося безліч нових алгоритмів, таких як:

- Наївний Байєсівський класифікатор
- Метод опорних векторів (SVM):
- Дерево рішень (Decision Tree)
- Випадковий ліс (Random Forest)
- Згорткові нейронні мережі (Convolutional Neural Networks, CNN)
- Рекурентні нейронні мережі (Recurrent Neural Networks, RNN)
- Трансформери (Transformers)

Метою даної роботи є створення нейронної мережі для класифікації текстових документів, а також порівняння цього методу з більш простішими. Також велику частину уваги в цій роботі буде виділено попередній обробці даних, а також алгоритмам векторизації тексту.

Підсумовуючи все вищесказане, предметом дослідження є різноманітні методи класифікації текстових документів, а об'єктом згорткові нейронні мережі, що будуть реалізовані в даній роботі.

# 1 МЕТОДИ КЛАСИФІКАЦІЇ ТЕКСТУ

## 1.1 Наївний Байєсівський класифікатор

Наївний байєсів класифікатор є ймовірнісним класифікатором, який використовує теорему Байєса для визначення ймовірності приналежності спостереження (елемента вибірки) до одного з класів. Він базується на припущенні про незалежність змінних [7].

Класифікатор обчислює ймовірність приналежності спостереження до кожного з класів на основі значень змінних. В ідеальному випадку, коли можна однозначно визначити приналежність до класу, байєсів класифікатор повертає одну ймовірність. У випадках, коли спостереження може належати до різних класів з різною ймовірністю, класифікатор повертає вектор ймовірностей для кожного класу [7].

Теорема Баєса формулюється наступним чином:

Нехай  $H^1, H^2, \dots$  – повна група подій, і  $A$  – деяка подія, ймовірність якої додатня. Тоді умовна ймовірність того, що має місце подія  $H^i$ , якщо в результаті експерименту відбулася подія  $A$ , може бути вирахована по формулі:[8]

$$P(H^i | A) = \frac{P(H^i)P(A|H^i)}{\sum_{j=1}^{\infty} P(H^j)P(A|H^j)} \quad (1.1)$$

Ідеальний байєсів класифікатор є оптимальним, оскільки його результат не може бути покращений. Він дає однозначну відповідь, коли це можливо, і кількісно характеризує ступінь неоднозначності в інших випадках [8].

Однак на практиці побудова ідеального байєсового класифікатора вимагає великої вибірки, яка містить всі можливі комбінації змінних. Розмір такої вибірки зростає експоненційно зі збільшенням числа змінних. Тому на практиці використовується наївний байєсів класифікатор, який базується на припущенні про незалежність змінних. Це дозволяє обмежити вивчення

взаємодії між змінними і зосередитися на впливі кожної змінної окремо на класифікацію [8].

Перевагою наївного байесового класифікатора є зменшення вимог до розміру вибірки з експоненційних до лінійних. Крім того перевагами є швидкість роботи, простота і масштабованість, помірні вимоги до пам'яті [8].

Недолік полягає в тому, що модель є точною лише у випадку, коли виконується припущення про незалежність змінних. В інших випадках обчислені ймовірності можуть бути неправильними, а сума ймовірностей може не дорівнювати одиниці, що вимагає нормування результату. Однак незначні відхилення від незалежності зазвичай мають незначний вплив на точність класифікатора [8].

## 1.2 Метод опорних векторів

Метод опорних векторів (SVM) є одним з популярних алгоритмів для класифікації тексту. Метод використовує представлення зразків як точок у просторі і будує гіперплощину або набір гіперплощин у високо- або нескінченно-вимірному просторі для вирішення задач класифікації, регресії та інших. Головна ідея полягає в тому, щоб знайти гіперплощину, яка максимально віддалена від найближчих точок тренувального набору даних для кожного з класів. Ця гіперплощина називається прогалиною і є найширшою. Чим ширша прогалина, тим менше ймовірність помилки класифікатора на нових даних[9].

Переваги:

SVM добре справляється з класифікацією високорозмірних просторів, що часто відповідає векторним поданням тексту. Він може ефективно використовувати лінійні та нелінійні границі рішень для розділення класів.

SVM зазвичай демонструє високу точність класифікації тексту, особливо коли маємо обмежену кількість даних для навчання. Він шукає оптимальний роздільний гіперплощину, яка максимізує межу між класами.

SVM може використовуватись з різними ядровими функціями для розширення можливостей класифікації. Це дозволяє йому працювати з різноманітними типами даних та вирішувати різні завдання класифікації тексту.

Оскільки SVM оптимізує границю рішень на основі опорних векторів, він уникає локальних мінімумів, які можуть виникати в інших методах класифікації[9].

Недоліки:

Тренування SVM може бути обчислювально витратним, особливо коли використовуються ядрові функції. Для великих наборів даних час навчання може бути значною проблемою.

Вибір відповідного ядра є складною задачею, оскільки різні ядра можуть показувати кращі результати залежно від конкретного набору даних. Це вимагає додаткових експериментів та налаштування параметрів для досягнення оптимальних результатів.

SVM може бути чутливим до шуму та викидів в даних. Неправильні мітки або викиди можуть суттєво вплинути на розділяючу границю та результати класифікації.

Основний фокус SVM - досягнення точності класифікації, а не пряме виведення значущості ознак. Це може ускладнювати інтерпретацію результатів та розуміння того, які особливості допомагають приймати рішення[9].

У підсумку, метод опорних векторів є потужним і ефективним алгоритмом класифікації тексту з високою точністю, однак, він має свої недоліки, такі як обчислювальна складність та чутливість до шуму.

### 1.3 Глибоке навчання

Глибоке навчання (англ. deep learning) - це галузь машинного навчання, що займається розробкою та застосуванням алгоритмів, здатних самостійно вчитися та розуміти складні структури даних. Воно моделює взаємозв'язки у

великих наборах даних за допомогою штучних нейронних мереж, що містять багато внутрішніх параметрів для виявлення прихованих залежностей та ознак.

Основна ідея глибокого навчання полягає в тому, що нейронна мережа самостійно визначає оптимальні параметри для вирішення певної задачі. Кожен шар нейронної мережі виконує певні обчислення над вхідними даними та передає результати до наступного шару. Під час тренування мережі ваги та зсуви (bias) автоматично адаптуються таким чином, щоб мінімізувати помилку. Завдяки спеціалізації шарів та ієрархічній обробці даних, глибокі мережі є більш природними для сприйняття та аналізу людиною.

У порівнянні з "мілкими" нейронними мережами, глибокі мережі мають більше нейронів та зв'язків, що дозволяє їм мати більшу обчислювальну потужність та здатність моделювати складніші залежності[26].

### 1.3.1 Відмінність між машинним та глибоким навчанням

Незважаючи на те, що глибоке навчання є формою машинного навчання, вони є досить різними. На прикладі задачі класифікації тексту можна зрозуміти, в чому полягає основна різниця між машинним та глибоким навчанням. Процес створення моделі машинного навчання починається з того, що відповідні ознаки витягуються з тексту вручну. Потім ці ознаки використовуються для створення моделі, яка класифікує текст. При глибокому навчанні відповідні ознаки витягуються автоматично.[26] Крім того, в моделях глибокого навчання задача виконується з початку і до кінця, це означає, що коли мережі надаються необроблені дані і завдання для виконання, наприклад, класифікація, то вона вчиться робити це автоматично.

Ще одна ключова відмінність полягає в тому, що алгоритми глибокого навчання масштабуються разом з даними, на відміну від алгоритмів машинного навчання[1].

### 1.3.2 Нейронні мережі

Нейронні мережі є результатом довгого еволюційного процесу. Їх історія починається в 1943 році, коли Уоррен МакКаллок та Уолтер Піттс представили першу концепцію штучного нейрона, яка інспірувалася біологічною нейронною мережею.

У 1950-60 роках Розенблатт Френк розробив перцептрон - один з найпростіших видів нейронних мереж. Він здатний виконувати бінарну класифікацію і став важливим кроком у розвитку нейронних мереж.

Протягом наступних десятиліть розвиток нейронних мереж був обмежений обчислювальною потужністю та обмеженими навчальними алгоритмами. Проте в 1980-х роках почали з'являтися нові архітектури мереж і покращені алгоритми навчання.

Після того, як Юрген Шмідхубер та Деміс Хасабіс впровадили глибокі нейронні мережі з використанням алгоритму зворотного поширення помилок у 1980-х роках, глибоке навчання почало набирати популярність. Завдяки покращенню обчислювальних ресурсів і розширенню наборів даних нейронні мережі стали успішними у вирішенні складних завдань, таких як розпізнавання образів, розпізнавання мови, машинний переклад та багато інших[11].

Останнім часом нейронні мережі, зокрема глибокі нейронні мережі, досягли великого прогресу завдяки новим алгоритмам, наприклад, згортковим нейронним мережам і рекурентним нейронним мережам. Ці технології знаходять застосування в багатьох сферах, включаючи комп'ютерне зору, обробку природної мови, автономну навігацію, медицину та інші.

#### **Переваги:**

Нейронні мережі можуть автоматично вивчати внутрішні залежності в текстових даних, що дозволяє їм адаптуватись до різноманітних типів текстів та складних патернів.

Нейронні мережі можуть враховувати семантичні зв'язки між словами та контекстом, що допомагає уточнити класифікацію тексту. Вони можуть розуміти значення слів та фраз і враховувати їх в контексті.

Нейронні мережі можуть бути створені з різними архітектурами та глибиною, що дозволяє їм адаптуватись до різноманітних завдань класифікації тексту. Вони також можуть навчатися на великій кількості даних і покращувати свою точність з часом.

Нейронні мережі можуть ефективно працювати з великими обсягами текстових даних. Завдяки паралельному обчисленню на графічних процесорах (GPU) або застосуванню розподілених обчислювальних ресурсів, нейронні мережі можуть швидко обробляти великі обсяги тексту[1].

#### **Недоліки:**

Нейронні мережі потребують великої кількості маркованих даних для навчання. Це може бути проблемою в випадку обмежених даних або коли маркування тексту вимагає великої експертної праці.

Деякі нейронні мережі можуть бути обчислювально витратними, особливо якщо вони мають глибоку архітектуру та велику кількість параметрів. Це може вимагати потужних обчислювальних ресурсів для тренування та використання мережі.

Нейронні мережі можуть бути частково чорними ящиками, що ускладнює інтерпретацію причини, з якої вони роблять певні прогнози. Важко зрозуміти, як саме мережа робить висновки на основі текстових даних.

Деякі аспекти нейронних мереж, такі як вибір архітектури та налаштування гіперпараметрів, можуть бути складними і вимагати дослідження та експериментів для досягнення кращих результатів[1].

### 1.3.4 Згорткові нейронні мережі

Згорткові нейронні мережі - це клас глибоких нейронних мереж, які розпізнавати і класифікувати певні ознаки з об'єкту і широко

використовуються для аналізу візуальних зображень. Це не єдина їх сфера застосування, проте класифікація зображень вимагає високої точності, що робить ЗНМ одним з найкращих варіантів для виконання завдань цієї категорії. Проте ЗНШ використовують також і для класифікації тексту, що і буде продемонстровано в даній роботі.

Термін "згортка" в назві ЗНМ означає математичну функцію, яка є особливим видом лінійної операції, коли дві функції перемножуються для отримання третьої функції, яка виражає, як форма однієї функції змінюється під впливом іншої[26]. Як приклад, матриці двох текстів можна перемножити, для того, щоб отримати результуючу матрицю, з якої можна буде отримувати набір ознак.

ЗНМ складається з декількох основних шарів(див рис. 1.1)[27] :

1. Шар згортки (Convolutional Layer)
2. Агрегувальний шар (Pooling Layer)
3. Шар активації (Activation Layer)
4. Повнозв'язаний шар (Fully Connected Layer)

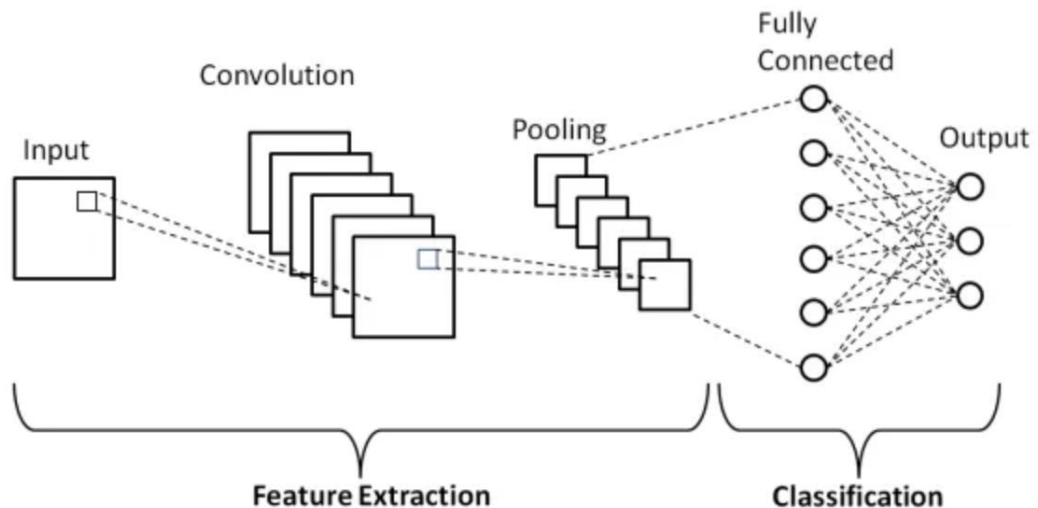


Рисунок 1.1 - Структура згорткової нейронної мережі[27]

Перший шар виконує операцію згортки, описаної вище. Кожен фільтр рухається по вхідних даних і виконує операцію згортки для створення фільтрованих карт ознак.

Після шару згортки може бути доданий агрегаційний шар. Його основна властивість полягає в тому, що шар зменшує просторові розміри карт ознак, зберігаючи основні особливості. Найпоширенішим типом підсумовування є шар максимальної агрегації (Max Pooling), де вибирається максимальне значення з кожного регіону[26].

Наступним йде шар активації, що використовує нелінійну функцію активації, таку як ReLU (Rectified Linear Unit) або Sigmoid, для нелінійності в мережі. Цей шар допомагає моделі виражати складніші залежності між вхідними і вихідними даними. Він не зображений на рисунку, проте є дуже важливою частиною моделі[26].

Після одного або декількох шарів згортки та агрегації, йде повнозв'язаний шар. Цей шар складається з ваг і зсувів разом з нейронами і використовується для з'єднання нейронів між двома різними шарами. Ці шари зазвичай розміщуються перед вихідним шаром і утворюють останні кілька шарів архітектури ЗНМ та зазвичай використовується для остаточної класифікації або регресії[26].

Ці основні шари згорткової нейронної мережі можуть повторюватися в різних комбінаціях для розробки більш складних архітектур. Кожен шар має свою функцію і допомагає моделі витягти важливі ознаки з вхідних даних та здійснити кінцеву класифікацію чи регресію.

## 1.4 Ефективніший метод для класифікації тексту

Нейронні мережі є кращими для класифікації тексту порівняно з методом опорних векторів (SVM) та баєсівським класифікатором[10].

Нейронні мережі можуть автоматично виявляти складні залежності та шаблони у текстових даних завдяки своїй гнучкості та здатності до

відображення нелінійних зв'язків між ознаками. Вони можуть адаптуватися до різноманітних типів текстової інформації, включаючи великі обсяги тексту та складні структури даних. Також вони досягають кращої точності класифікації, особливо в складних завданнях, де існує багато класів або незбалансована розподіленість даних. Надзвичайно важливою перевагою є те, нейронні мережі здатні до автоматичного вивчення репрезентацій тексту на основі вхідних даних, що дозволяє уникнути ручної інженерії ознак. Вони можуть виявляти корисні ознаки та знаходити скриті закономірності, що полегшує процес моделювання і покращує результати класифікації[1].

Те, що дає помітну перевагу поруч з SVM алгоритмом є те, що нейронні мережі можуть бути гнучкими та масштабованими, дозволяючи їх використання для обробки великих обсягів тексту та розширення до складніших моделей, які враховують контекстуальні залежності та семантичні взаємозв'язки[10].

## 1.5 Висновки до розділу 1

В даному розділі було розглянуте питання класифікації тексту, його актуальність, а також сучасні алгоритми, що використовують для виконання цієї операції.

З цього розділу випливає, що інструментів для класифікації тексту на даний момент існує досить багато, найбільш поширеними з них є байєсівський класифікатор, метод опорних векторів, нейронні мережі, а також більш новий підхід - трансформери. Кожен інструмент має свою сферу застосування, проте нейронні мережі є досить гнучкими, для виконання цієї задачі, і саме ця модель буде реалізована в цій роботі.

## 2 ОБРОБКА ДАНИХ

### 2.1 Роль обробки даних в проєкті

За останні кілька років кількість даних, що зберігаються у світі, значно зросла. Сьогодні ми маємо великі масиви даних, які є надзвичайно важливими для багатьох сфер людської діяльності, таких як наука, медицина, економіка, соціальна сфера тощо. Однак, збільшення кількості даних виникає швидше, ніж можливості комп'ютерів для їх обробки. Це ставить перед нами складні завдання з обробки та аналізу великих обсягів даних.

Для успішної реалізації AI проєкту, необхідність в глибокому вивченні проблеми, аналізі й дослідженні нюансів є невід'ємною складовою процесу. Однак, без належного збору даних, будь-який проєкт не матиме можливості реалізуватися. Якщо на етапі розробки проєкту виявляється, що необхідної інформації неможливо зібрати або її просто немає, то такий проєкт не може функціонувати.

На початковому етапі розвитку штучного інтелекту, відсутність достатньої кількості даних стала серйозною проблемою. Це призвело до того, що алгоритми штучного інтелекту в основному зосереджувались на моделюванні для отримання якісних результатів з обмежених даних. Така парадигма розвитку штучного інтелекту називається "Model Centric AI"[12].

Однак з появою нових соціальних мереж та збільшенням обсягу електронних даних, кількість наявних даних значно зросла. Це дозволило зсунути акцент розвитку штучного інтелекту з моделювання на роботу з даними.

Зафіксовано стрімке зростання обсягів даних у світі, що перевищує можливості сучасних обчислювальних систем. У зв'язку з цим, головною складовою процесу роботи з даними стає їх очищення та перетворення у зручний формат для подальшого використання в нейронних мережах[12]. Одним із таких форматів є векторизація.

## 2.2 Основні задачі попередньої обробки даних

Однак перед тим як векторизувати текст він має пройти шлях препроцесингу. Це необхідний крок, який має декілька цілей, які впливають на швидкість та якість роботи моделі.

### 1. Очищення даних

Часто реальні дані мають помилки, пропуски, аномалії або шум, що може негативно вплинути на якість моделі. Попередня обробка даних дозволяє виявити і виправити ці проблеми: видалити або замінити некоректні значення, заповнити пропуски або видалити аномальні спостереження[13].

### 2. Нормалізація і стандартизація даних

Різні атрибути можуть мати різні діапазони значень. Це може призвести до проблем з вагою або швидкістю збіжності моделі під час навчання. Попередня обробка даних може включати нормалізацію або стандартизацію, щоб привести всі атрибути до спільного масштабу і поліпшити продуктивність моделі[13].

### 3. Видалення зайвих ознак

У наборі даних можуть бути ознаки, які не мають впливу на цільову змінну або мають дуже слабкий вплив. Вони можуть вносити шум і займати зайве місце. Попередня обробка даних може включати видалення таких зайвих ознак для поліпшення ефективності моделі[13].

### 4. Видалення дублікатів

Деякі набори даних можуть містити повторювані спостереження, що не несуть додаткової інформації. Видалення дублікатів дозволяє зменшити розмір набору даних і уникнути зайвого навчання на однакових зразках. Або ж видалення слів в тексті, що не несуть змістового навантаження[13].

Розглянемо ці етапи детально.

### 2.2.1 Виправлення скорочень

Скорочення - це слова, що використовуються здебільшого в розмовній мові та в побуті. Ці слова виникають шляхом вилучення букв або заміною їх апострофом. Цей крок є надзвичайно важливим при обробці коментарів в соцмережах, таких як Twitter, LinkedIn, Instagram, для, наприклад визначення емоційного забарвлення тексту.

Існують бібліотеки, що допомагають визначати такі слова та заміщати їх повноцінною версією. Такі бібліотеки існують на мові Python і здебільшого для англійської мови. В даній роботі будуть використовуватись тести англійською мовою, тож цей крок можна буде автоматизувати бібліотекою(рис. 2.1).

```
▶ contractions_dict = {
    "don't": "do not",
    "can't": "cannot",
    "isn't": "is not",
    "aren't": "are not",
    "won't": "will not",
    "couldn't": "could not",
    "shouldn't": "should not",
    "wouldn't": "would not",
    "didn't": "did not",
    "doesn't": "does not"
    # Та інші
}

def expand_contractions(text):
    words = text.split()
    expanded_text = []
    for word in words:
        if word.lower() in contractions_dict:
            expanded_text.append(contractions_dict[word.lower()])
        else:
            expanded_text.append(word)
    return ' '.join(expanded_text)

input_text = "I don't know if I can make it."
expanded_text = expand_contractions(input_text)
print(expanded_text)
```

↳ I do not know if I can make it.

Рисунок 2.1 – Реалізація мовою Python виявлення скорочень

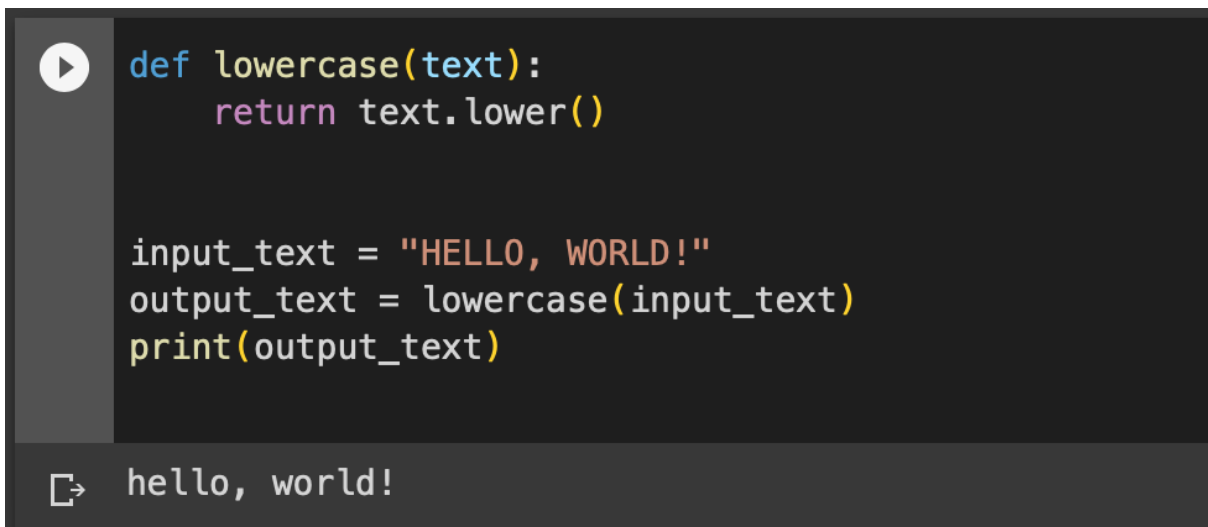
### 2.2.2 Приведення до нижнього регістру

Приведення слів до нижнього регістру використовується в попередній обробці даних з декількох причин:

По-перше, це дозволяє уніфікувати текстові дані. Тобто розрізняти слова, незалежно від того, як вони написані (великі чи малі літери). Наприклад, "Сонце", "сонце" і "СонЦе" будуть перетворені в "сонце", що дозволяє сприймати їх як однакові слова.

По-друге, приведення слів до нижнього регістру допомагає зменшити розмірність словника. У текстових даних часто зустрічаються однакові слова, які написані з великої або малої літери, але мають ту саму семантичну значущість. Приведення до нижнього регістру дозволяє розглядати такі слова як одне слово, що спрощує обробку і аналіз тексту.

По-третє, підвищення швидкості обробки. Робота з нижнім регістром може бути швидшою, оскільки не потрібно враховувати різницю між великими і малими літерами. Це може позитивно позначитися на продуктивності обробки текстових даних та тренуванні моделей[14](рис. 2.2).



```
def lowercase(text):  
    return text.lower()  
  
input_text = "HELLO, WORLD!"  
output_text = lowercase(input_text)  
print(output_text)
```

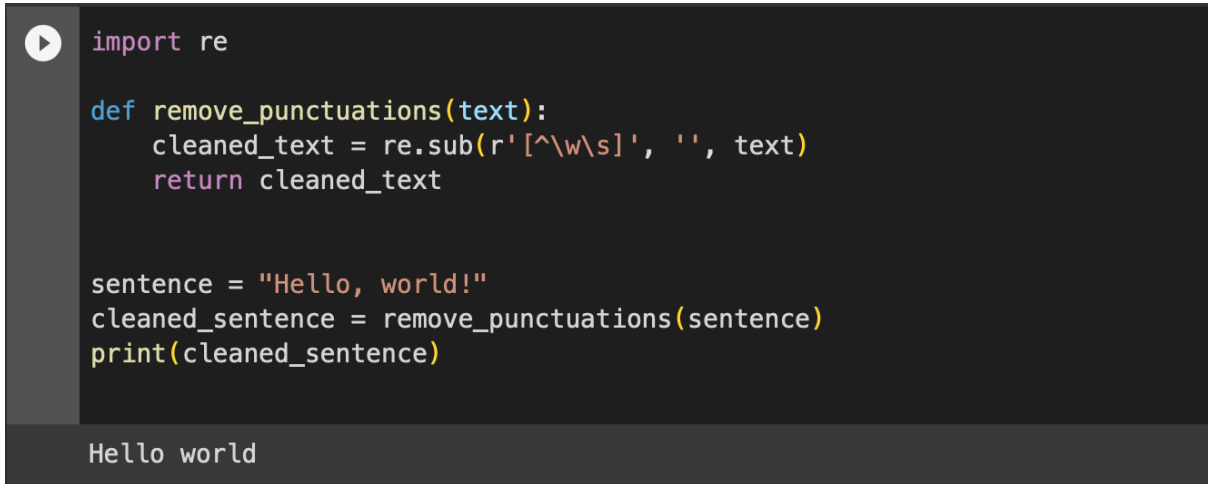
hello, world!

Рисунок 2.2 – Реалізація мовою Python приведення до нижнього регістру

### 2.2.3 Видалення пунктуації

Видалення пунктуації має схоже значення в попередній обробці тексту як і приведення тексту до нижнього регістру. Адже має такі ж функції : уніфікація тексту, зменшення розмірності словника та підвищення швидкості обробки текстових даних, та часу тренування моделі.

Видалення пунктуації є важливим етапом в процесі токенізації, коли текст розбивається на окремі слова або так звані токени. Пунктуація може бути використана як роздільник між словами, і видалення її допомагає правильно розподілити слова та підготувати текст для подальшої обробки[14](рис. 2.3).



```
import re

def remove_punctuations(text):
    cleaned_text = re.sub(r'^\w\s]', '', text)
    return cleaned_text

sentence = "Hello, world!"
cleaned_sentence = remove_punctuations(sentence)
print(cleaned_sentence)
```

Hello world

Рисунок 2.3 – Реалізація мовою Python видалення пунктуації

## 2.2.4 Видалення стоп слів

Стоп слова - це слова, що є найуживанішими в мовленні та текстах, проте не несуть важливої семантичної цінності. Зазвичай їх видалення з тексту приносить користь, адже зменшує розмір словника, а також призводить до того, що словник наповнений лише тими словами, які несуть суттєве значення в документі.

До прикладу, в NLP моделях часто використовують такий інструмент, як Word Cloud. Що відображає частоту появи слова в документі розміром цього слова на зображенні. Стоп слова зазвичай займають X% тексту, а отже, пропустивши крок, на якому їх видаляють, маємо високу ймовірність невірною відображення wordcloud. Це зумовлено тим фактом, що велика частина простору “хмаринки” буде заповнена словами, які не матимуть семантичного забарвлення.

Видалення стоп-слів допомагає забезпечити консистентність у текстових даних, оскільки вони виключаються з усіх документів. Це дозволяє зробити

подальшу обробку і порівняння текстових документів більш однорідними та порівнянними.

До таких слів зокрема належать:

- Загальні службові слова:

Це такі слова, як "я", "він", "вона", "це", "ми", "ви" і "вони", які використовуються для вказівки особи, множини або безособового формату.

- Загальні прийменники та сполучники:

Це такі слова, як "в", "на", "з", "до", "та", "але", "або", "як", "що", "якщо" і "хоча", які використовуються для позначення місця, часу, відношень та зв'язків між словами.

- Часті дієслова:

Це такі слова, як "бути", "мати", "робити", "йти", "дати", "бачити", "розуміти" і "знати".

- Загальні займенники:

Це такі слова, як "цей", "той", "свій", "хто", "що", "який" і "скільки", які використовуються для позначення замітника або посилання на об'єкти.

- Числівники:

Це такі слова, як "один", "два", "три", "десять" і "багато", які вказують на кількість або порядковий номер.

Це не весь перелік слів, що відносяться до стоп слів. Більш того, стоп слова можуть варіюватись в залежності від мови та датасету, що використовується[15].

В мові програмування Python є готовий сет, який містить базові стоп слова, а також бібліотека, що допомагає виконати необхідні маніпуляції, для видалення цих слів[14](рис. 2.4).

```

import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

def remove_stopwords(text):
    # Завантажуємо список слів
    nltk.download('stopwords')
    stop_words = set(stopwords.words('english'))

    # Токенізуємо текст
    tokens = word_tokenize(text)

    # Видаляємо слова
    filtered_tokens = [word for word in tokens if word.lower() not in stop_words]

    # З'єднуємо слова назад в рядок
    filtered_text = ' '.join(filtered_tokens)

    return filtered_text

#
text = "This is he she it demonstration lover."
filtered_text = remove_stopwords(text)
print(filtered_text)

```

Рисунок 2.4 – Реалізація мовою Python видалення стоп слів

## 2.2.5 Стемінг та лематизація

Stemming і lemmatization є техніками лінгвістичної нормалізації, які застосовуються в попередній обробці текстових даних з наступними цілями[14]:

Зведення слів до їх базової форми: Stemming і lemmatization допомагають звести слова до їх базової, нормалізованої форми, що називається стем або лема відповідно. Це полегшує обробку і аналіз тексту, оскільки різні форми того ж слова розглядаються як одне слово. Наприклад, слова "running", "runs" і "ran" після stemming можуть бути зведені до стема "run", а після lemmatization - до леми "run".

Зменшення розмірності словника: Stemming і lemmatization допомагають зменшити розмірність словника та кількість унікальних слів. Замість зберігання різних форм одного слова як окремі токени, можна зберігати лише їх стеми або леми, що спрощує обробку тексту та зменшує вимоги до обсягу пам'яті.

Покращення збігання слів: Stemming і lemmatization допомагають забезпечити краще збігання слів. Оскільки різні форми одного слова будуть

зведені до одного стема або леми, це сприяє встановленню зв'язку між різними випадками вживання того ж слова в тексті, що може покращити точність аналізу та класифікації.

Зменшення шуму: Stemming і lemmatization допомагають знизити вплив морфологічних варіацій слів на аналіз тексту. Це може бути особливо корисним у випадках, коли морфологічні зміни не несуть суттєвої інформації для конкретної задачі аналізу тексту[16].

Загалом, застосування стемінгу або лематизації в попередній обробці даних залежить від конкретних вимог проекту(рис. 2.5).

```

▶ from nltk.stem import PorterStemmer, WordNetLemmatizer
   from nltk.tokenize import word_tokenize

   def stemming_and_lemmatization(text):
       # Ініціалізуємо об'єкти для stemming та lemmatization
       stemmer = PorterStemmer()
       lemmatizer = WordNetLemmatizer()

       # Токенізуємо текст на окремі слова
       tokens = word_tokenize(text)

       stemmed_words = []
       lemmatized_words = []

       for word in tokens:
           # Stemming
           stemmed_word = stemmer.stem(word)
           stemmed_words.append(stemmed_word)

           # Lemmatization
           lemmatized_word = lemmatizer.lemmatize(word)
           lemmatized_words.append(lemmatized_word)

       return stemmed_words, lemmatized_words

   # Приклад використання
   text = "The quick brown foxes jumped over the lazy dogs"
   stemmed_words, lemmatized_words = stemming_and_lemmatization(text)

   print("Stemmed words:", stemmed_words)
   print("Lemmatized words:", lemmatized_words)

```

Рисунок 2.5 – Реалізація мовою Python виявлення скорочень

## 2.3 Висновки до розділу 2

Цей розділ присвячений важливому етапу створення моделей машинного навчання, а саме попередній обробці даних. Окрім того, що попередня обробка

даних допомагає звузити потік даних до цільових, тобто забрати лишній шум, та уніфікувати їх, цей крок також допомагає уникати сучасної проблеми - великого потоку даних, який став можливим через розвиток технологій.

Даний розділ описує найбільш часті методи попередньої обробки тексту для подальшого аналізу моделлю машинного навчання.

## 3 МЕТОДИ ТА АЛГОРИТМИ ВЕКТОРНОГО ПРЕДСТАВЛЕННЯ ТЕКСТУ

### 3.1 Роль та значення векторизації тексту

Векторизація - це процес перетворення текстових даних на числовий вектор, що може бути подано у вигляді числового масиву. Векторизація використовується для підготовки даних до подальшого аналізу або машинного навчання, де модель отримує векторизовані дані на вхід.

Для покращення роботи алгоритму векторизації, дані піддаються очищенню. Це допомагає підвищити точність векторизації та зменшити кількість вимірів вектору, що в свою чергу сприятиме покращенню ефективності обробки даних та зменшенню вимог до ресурсів під час навчання моделі.

Різні техніки векторизації тексту можуть давати різні результати[17].

#### 3.1.1 Bag-of-Words

Алгоритм Bag-of-Words (Мішок Слів) - це метод, що використовується для представлення текстової інформації в числовому вигляді. Його основна ідея полягає у тому, що текст розбивається на окремі слова, а потім кожному слову ставиться у відповідність певне числове значення. Після цього текст можна розглядати як вектор, де кожна координата відповідає числовому значенню певного слова[17].

Алгоритм був створений в 1950-х роках та використовувався для автоматичної обробки природних мов з допомогою комп'ютерів.

Формальний алгоритм Bag-of-Words можна описати наступним чином(рис. 3.1):

1. Зібрати всі унікальні слова, які зустрічаються в даних. Це можна зробити шляхом побудови словника.

2. Для кожного слова в словнику створити відповідну координату у векторі. Ця координата буде відображати кількість входжень даного слова у текст.
3. Пройтися по кожному тексту і підрахувати кількість входжень кожного слова зі словника. Результатом буде вектор, де кожна координата відображає кількість входжень відповідного слова у текст.
4. Вектори, що відповідають різним текстам, можна порівнювати між собою за допомогою різних метрик подібності, таких як косинусна подібність[17].

```

from sklearn.feature_extraction.text import CountVectorizer

# Задаємо список текстових документів
documents = [
    "Це перший документ.",
    "Це другий документ.",
    "Це третій документ."
]

# Ініціалізуємо CountVectorizer
vectorizer = CountVectorizer()

# Застосовуємо Bag-of-Words до документів
bag_of_words = vectorizer.fit_transform(documents)

# Отримуємо слова, які були використані для побудови Bag-of-Words
feature_names = vectorizer.get_feature_names_out()

# Виводимо результати
print("Bag-of-Words:")
print(bag_of_words.toarray())
print("\nСлова:")
print(feature_names)

```

```

Bag-of-Words:
[[1 0 1 0 1]
 [1 1 0 0 1]
 [1 0 0 1 1]]

Слова:
['документ' 'другий' 'перший' 'третій' 'це']

```

Рисунок 3.1 – Реалізація мовою Python алгоритму Bag-of-Words

Переваги :

- Легкий у використанні та зрозумілий;
- Можна використовувати для будь-якого типу текстів;
- Ефективний при великих наборах даних.

Недоліки:

- Не враховує порядок слів у тексті;

- Не враховує семантичні зв'язки між словами[17].

### 3.1.2 TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) - це інший метод представлення тексту у числовому вигляді, що використовується для оцінки важливості кожного слова у тексті. Він базується на ідеї, що слова, які зустрічаються рідко у тексті, мають більшу вагу, ніж слова, що зустрічаються дуже часто. Алгоритм був запропонований в 1972 році Каренко та Мосесом, але досяг популярності в 1990-х роках завдяки його використанню в пошукових системах[24].

Формальний алгоритм TF-IDF можна описати наступним чином:

1. Зібрати всі унікальні слова, які зустрічаються в даних. Це можна зробити шляхом побудови словника.
2. Для кожного слова в словнику створити відповідну координату у векторі. Ця координата буде відображати значення TF-IDF для даного слова.
3. Обчислити значення TF-IDF для кожного слова у тексті. Значення TF-IDF обчислюється за формулою:  $TF-IDF = TF * IDF$ , де TF - це частота слова у тексті, а IDF - це інвертована частота документів, що містять дане слово. IDF можна обчислити за формулою:  $IDF = \log(N / df)$ , де N - це кількість документів у корпусі, а df - це кількість документів, що містять дане слово.
4. Вектор, що відповідає тексту, можна отримати як вектор значень TF-IDF для кожного слова у тексті[17].

Цей алгоритм є більш точним у порівнянні з Bag-of-Words, оскільки він враховує не тільки частоту входжень слова у текст, але і його важливість для всього корпусу текстів(рис. 3.2).

```

▶ from sklearn.feature_extraction.text import TfidfVectorizer

# Задані документи
documents = [
    "Це перший документ.",
    "Це другий документ.",
    "Це третій документ.",
    "Це останній документ."
]

# Ініціалізуємо TfidfVectorizer
vectorizer = TfidfVectorizer()

# Обчислюємо TF-IDF
tfidf_matrix = vectorizer.fit_transform(documents)

# Виводимо результат
feature_names = vectorizer.get_feature_names_out()
for i, doc in enumerate(documents):
    print(f"TF-IDF для документа {i+1}:")
    for j, term in enumerate(feature_names):
        tfidf = tfidf_matrix[i, j]
        if tfidf != 0:
            print(f"{term}: {tfidf:.2f}")
    print()

```

↗ TF-IDF для документа 1:  
 документ: 0.42  
 перший: 0.80  
 це: 0.42

Рисунок 3.2 – Реалізація мовою Python алгоритму TF-IDF

#### Переваги:

- Враховує частоту вживання та важливість слів в документі;
- Можна використовувати для багатьох типів текстів;
- Ефективний при великих наборах даних.

#### Недоліки:

- Не враховує порядок слів у тексті;
- Не враховує семантичні зв'язки між словами[17].

### 3.1.3 Word2Vec

Word2Vec - це алгоритм машинного навчання, який використовується для отримання векторних представлень слів у тексті. Ці вектори можна використовувати для вирішення різноманітних задач, таких як знаходження семантичної близькості між словами, класифікація документів і т. д.

був запропонований в 2013 році групою дослідників з Google, включаючи Томаса Міколова, Квентина Ле та Томаса Швенка[25].

Алгоритм Word2Vec можна описати наступним чином(рис. 3.3):

1. Побудувати словник унікальних слів, які зустрічаються у тексті.
2. Створити випадкову матрицю ваг, яка буде відображати вектори слів у просторі з невеликою кількістю вимірів (наприклад, 100 або 300).
3. Обрати підмножину тексту із корпусу текстів та підготувати дані для навчання, перетворивши кожне слово на вектор із створеної вагової матриці. Для цього можна використати підхід one-hot encoding, де кожне слово буде представлено у вигляді вектора з нулями всюди, крім одного місця, де буде одиниця.
4. Навчити модель Word2Vec за допомогою вибраного алгоритму машинного навчання, такого як Skip-Gram або CBOW. Skip-Gram вивчає контекст кожного слова, тобто використовує вектори слів, що оточують дане слово, для прогнозування самого слова. CBOW, навпаки, вивчає слова на основі їх контексту, тобто використовує вектори слів у тексті для прогнозування вектора центрального слова.
5. Отримати векторне представлення кожного слова у тексті за допомогою створеної вагової матриці[17].

```

▶ from gensim.models import Word2Vec
sentences = [['I', 'love', 'machine', 'learning'],
             ['I', 'love', 'deep', 'learning'],
             ['I', 'enjoy', 'NLP'],
             ['I', 'enjoy', 'computer', 'vision']]

# Задаємо параметри моделі Word2Vec
model = Word2Vec(sentences, min_count=1)

# Отримуємо вектор для певного слова
vector = model.wv['learning']

# Знаходимо схожі слова для певного слова
similar_words = model.wv.most_similar('learning')
print(similar_words)

[('machine', 0.1991206258535385), ('vision', 0.17018885910511017),

```

Рисунок 3.3 – Реалізація мовою Python алгоритму Word2Vec

Переваги:

- Враховує семантичні зв'язки між словами;
- Можна використовувати для багатьох типів текстів;
- Ефективний при великих наборах даних;
- Може знаходити аналогії між словами, такі як "король - чоловік + жінка = королева".

Недоліки:

- Не враховує порядок слів у тексті;
- Може бути складним у використанні;
- Може вимагати більшої кількості даних для тренування;
- Не може враховувати контекст на більш широкому рівні, так як модель тут звертає увагу лише на околиці слів, а не на цілий текст[17].

### 3.1.4 GloVe

GloVe (Global Vectors for Word Representation) - це алгоритм машинного навчання, який використовується для отримання векторних представлень слів у тексті. Ці вектори можна використовувати для вирішення різноманітних задач,

таких як знаходження семантичної близькості між словами, класифікація документів і т. д.

був створений в 2014 році студентами Стенфордського університету, включаючи Джеффри Пенью[18].

Алгоритм GloVe можна описати наступним чином(рис. 3.4):

1. Побудувати матрицю співвідношень між словами на основі корпусу текстів. Для цього потрібно підрахувати, скільки разів кожна пара слів зустрічається поруч у текстах корпусу.
2. Створити випадкову матрицю ваг, яка буде відображати вектори слів у просторі з невеликою кількістю вимірів (наприклад, 100 або 300).
3. Визначити функцію втрат для навчання моделі GloVe. Мета полягає в тому, щоб зменшити значення функції втрат для пар слів, які зустрічаються поруч у текстах корпусу, і збільшити значення функції втрат для пар слів, які не зустрічаються поруч.
4. Навчити модель GloVe за допомогою методу градієнтного спуску. Модель буде навчатися зменшувати значення функції втрат, щоб знайти оптимальні векторні представлення слів у тексті.
5. Отримати векторне представлення кожного слова у тексті за допомогою створеної вагової матриці[18]

```

import gensim
import numpy as np

corpus = [
    ['I', 'like', 'apples'],
    ['I', 'like', 'oranges'],
    ['I', 'enjoy', 'eating', 'apples'],
    ['I', 'enjoy', 'eating', 'bananas'],
    ['I', 'like', 'to', 'eat', 'apples'],
    ['I', 'like', 'to', 'eat', 'oranges'],
    ['I', 'like', 'to', 'eat', 'bananas']
]

# Побудова моделі GloVe
model = gensim.models.Word2Vec(corpus, min_count=1, vector_size=100, workers=4, window=5, sg=1)

# Отримання матриці вкладень слів
embedding_matrix = np.zeros((len(model.wv.key_to_index), model.vector_size))
for i, word in enumerate(model.wv.key_to_index):
    embedding_vector = model.wv[word]
    embedding_matrix[i] = embedding_vector

for word in model.wv.key_to_index:
    print(f"Word: {word}, Embedding: {model.wv[word]}")

```

```

Word: I, Embedding: [-5.3627364e-04  2.3644030e-04  5.1033380e-03  9.0093156e-03
 -9.3029849e-03 -7.1168183e-03  6.4589200e-03  8.9730574e-03
 -5.0154841e-03 -3.7633514e-03  7.3805125e-03 -1.5334529e-03
 -4.5366911e-03  6.5541049e-03 -4.8602205e-03 -1.8160546e-03
  2.8765788e-03  9.9191407e-04 -8.2852319e-03 -9.4489064e-03
  7.3118163e-03  5.0702426e-03  6.7577823e-03  7.6292612e-04
  6.3509131e-03 -3.4053822e-03 -9.4640278e-04  5.7686423e-03
 -7.5217364e-03 -3.9361245e-03 -7.5115873e-03 -9.3003456e-04
  9.5381849e-03 -7.3192688e-03 -2.3337321e-03 -1.9377909e-03
  8.0774607e-03 -5.0200117e-03  4.5158060e-05 -4.7527540e-03

```

Рисунок 3.3 – Реалізація мовою Python алгоритму GloVe

Переваги:

- Враховує семантичні зв'язки між словами;
- Ефективний при великих наборах даних.

Недоліки:

- Не враховує порядок слів у тексті;
- Може вимагати більшої кількості даних для тренування[18].

## 3.2 Висновки до розділу 3

В даному розділі описано основні методи векторизації, а також поняття, що з ними пов'язані. Векторизація - це процес перетворення текстових даних на числовий вектор, що може бути подано у вигляді числового масиву.

Векторизація використовується для підготовки даних до подальшого аналізу або машинного навчання, де модель отримує векторизовані дані на вхід.

Існує достатньо методів векторизації тексту, що можуть застосовуватись в залежності від поставленої задачі.

## 4 СТВОРЕННЯ ПРОДУКТУ ДЛЯ КЛАСИФІКАЦІЇ ТЕКСТУ

### 4.1 Вибір бібліотеки для реалізації нейронної моделі

Keras - це API для глибокого навчання та нейронних мереж від Франсуа Шолле, який може працювати поверх Tensorflow (Google), Theano або CNTK (Microsoft)[19].

Цитуючи книгу Франсуа Шолле "Глибоке навчання за допомогою Python"[1]:

Keras - це бібліотека на рівні моделей, що надає високорівневі будівельні блоки для розробки моделей глибокого навчання. Вона не обробляє низькорівневі операції, такі як маніпуляції з тензорами та диференціювання. Замість цього вона покладається на спеціалізовану, добре оптимізовану тензорну бібліотеку, яка слугує внутрішнім рушієм Keras. Ця бібліотека допомагає швидше почати експериментувати з нейронними мережами без необхідності реалізовувати кожен шар і частину самостійно. Tensorflow - з іншого боку, теж хороша бібліотека для машинного навчання, але недоліком є те, що потрібно реалізувати багато шаблонного коду, щоб запустити модель[1].

### 4.2 Перенавчання та базова модель

При роботі з машинним навчанням, одним з важливих кроків є визначення базової моделі. Зазвичай це проста модель, яка потім використовується для порівняння з більш складними моделями, які варто буде протестувати. У цьому випадку базова модель використовується для порівняння з більш складними методами, що включають (глибокі) нейронні мережі[21].

Спочатку дані розділяються на навчальний і тестовий набір, що дозволить оцінити точність і побачити, чи добре прогнозує модель. Це означає,

що модель здатна добре працювати на даних, з якими вона раніше не зустрічалась. Це спосіб перевірити, чи модель не є перенавченою.

Перенавчання - це коли модель занадто добре показує себе на навчальних даних. Цього варто уникати, оскільки це означатиме, що модель здебільшого просто запам'ятала навчальні дані. Це призведе до високої точності з навчальними даними, але низької точності на тестових даних[1].

### 4.3 Вибір мови програмування для створення проекту

Очевидним вибором для створення нейронної мережі є мова програмування Python. Вона є однією з найпопулярніших мов програмування для виконання такого типу задач.

#### 1. Простота використання:

Python має простий і зрозумілий синтаксис, що робить його дуже доступним для початківців і ефективним для розробників. Чистий і зрозумілий код сприяє швидкому розробці, налагодженню та зрозумінню алгоритмів нейронних мереж.

#### 2. Багата екосистема бібліотек:

Python має широкий вибір бібліотек, таких як TensorFlow, Keras, PyTorch і scikit-learn, які спеціально розроблені для роботи з нейронними мережами. Ці бібліотеки надають потужні інструменти для побудови, навчання і оцінювання нейронних мереж, а також для обробки даних.

#### 3. Велика спільнота розробників:

Python має активну та розширену спільноту розробників, яка постійно вносить внесок до розвитку бібліотек та інструментів для нейронних мереж. Це означає, що ви зможете знайти велику кількість документації, прикладів коду та підтримки спільноти, що допоможе вам вирішувати проблеми та розвиватися як розробник нейронних мереж.

#### 4. Інтеграція з іншими мовами:

Python може легко інтегруватися з іншими мовами програмування, такими як C++ або Java. Це дає змогу використовувати швидкі і оптимізовані бібліотеки написані на мовах нижчого рівня, а також сполучати їх з потужними можливостями Python.

#### 5. Велика кількість ресурсів для навчання:

Python має безліч підручників, курсів та онлайн-ресурсів для навчання нейронних мереж. Ви зможете знайти відеоуроки, блоги, форуми та інші ресурси, що допоможуть вам розуміти концепції та методи нейронних мереж і їх реалізацію на Python.

Усе це робить Python найкращим вибором для створення нейронних мереж та розвитку у галузі глибокого навчання. Він надає зручний, ефективний та гнучкий інструментарій, що допомагає створювати потужні та складні моделі нейронних мереж[20].

## 4.4 Keras

Keras підтримує два основних типи моделей. Sequential model - це лінійний стек шарів, де можна використовувати велику кількість доступних шарів у Keras. Найпоширенішим є шар Dense, який є звичайним щільно пов'язаним шаром нейронної мережі з усіма вагами та зміщеннями[19].

## 4.5 Аналіз даних

Для даної роботи було обрано дані, що містять в собі новини з каналу BBC та їх тему. Отже класифікатору необхідно буде визначити теми новин.

Датасет містить в собі 1490 текстів новин, що в свою чергу промарковані відповідно до їхньої тематики[23].

EDA, або аналіз даних перед виконанням (Exploratory Data Analysis) це процес дослідження та розуміння набору даних для виявлення його особливостей та взаємозв'язків[22].

Враховуючи, що ми аналізуємо текстові дані, то вони піддаються попередній обробці безпосередньо перед аналізом. Сам аналіз виглядатиме так:

- Огляд структури даних(рис. 4.1)

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1490 entries, 0 to 1489
Data columns (total 5 columns):
#   Column          Non-Null Count  Dtype
---  ---
0   ArticleId       1490 non-null   int64
1   Text            1490 non-null   object
2   Category        1490 non-null   object
3   text1           1490 non-null   object
4   Text_Length     1490 non-null   int64
dtypes: int64(2), object(3)
memory usage: 58.3+ KB
None
```

	ArticleId	Text_Length
count	1490.000000	1490.000000
mean	1119.696644	1713.257718
std	641.826283	1184.064953
min	2.000000	20.000000
25%	565.250000	904.750000
50%	1112.500000	1555.500000
75%	1680.750000	2339.250000
max	2224.000000	13826.000000

Рисунок 4.1 – Загальна інформація про дані

- Пропорція та теми новин у датасеті(рис. 4.2). По осі X виведені всі варіанти категорій в даних, а по осі Y кількість категорій в таблиці.

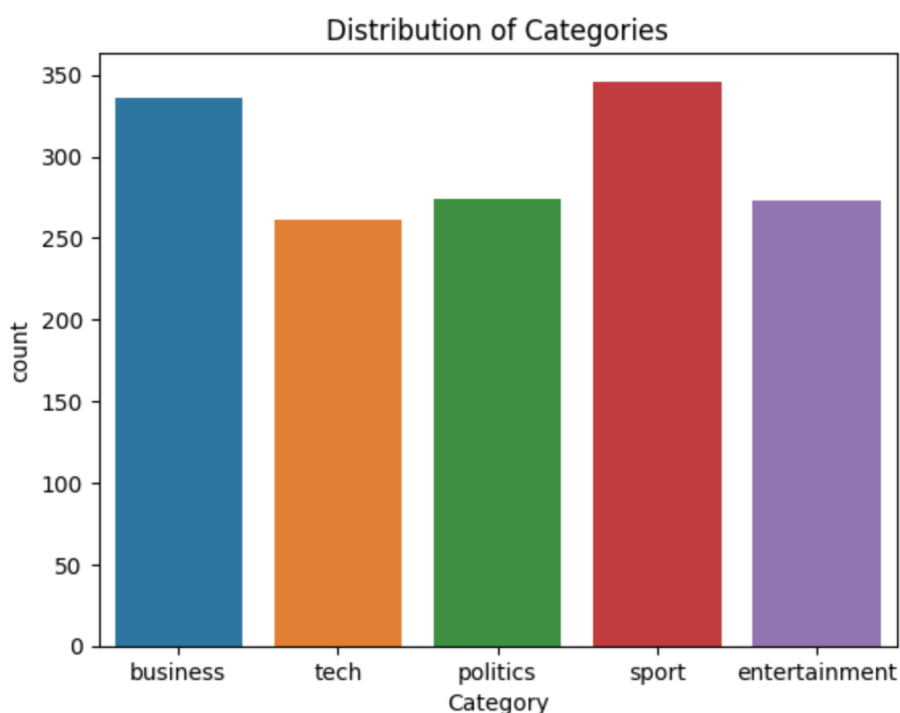
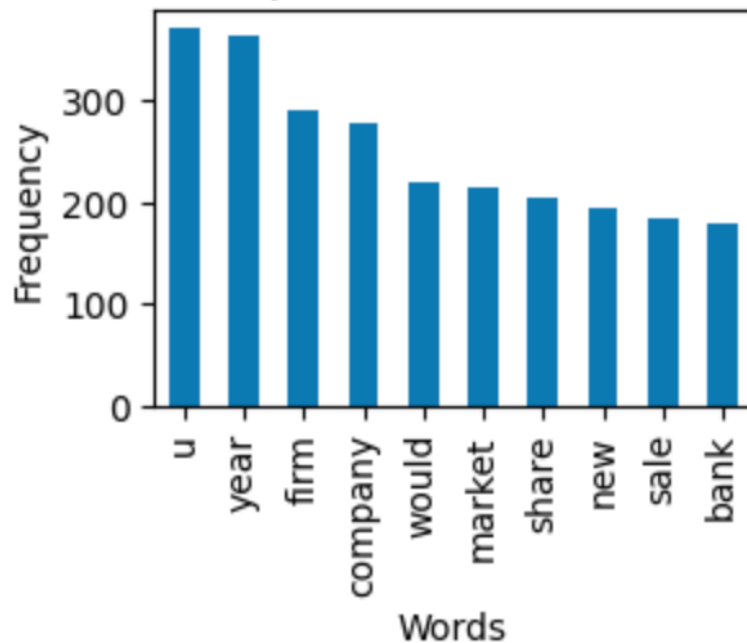


Рисунок 4.2 – Пропорція новин у наборі

- Зображення слів, що найчастіше зустрічаються в новинах в залежності від їх теми(рис. 4.3-4.5) На осі X знаходяться слова, що найчастіше вживаються в новинах певної тематики, а на осі Y частота зустрічей цих слів.

Top 10 Most Frequent Words in Business Category



Top 10 Most Frequent Words in Tech Category

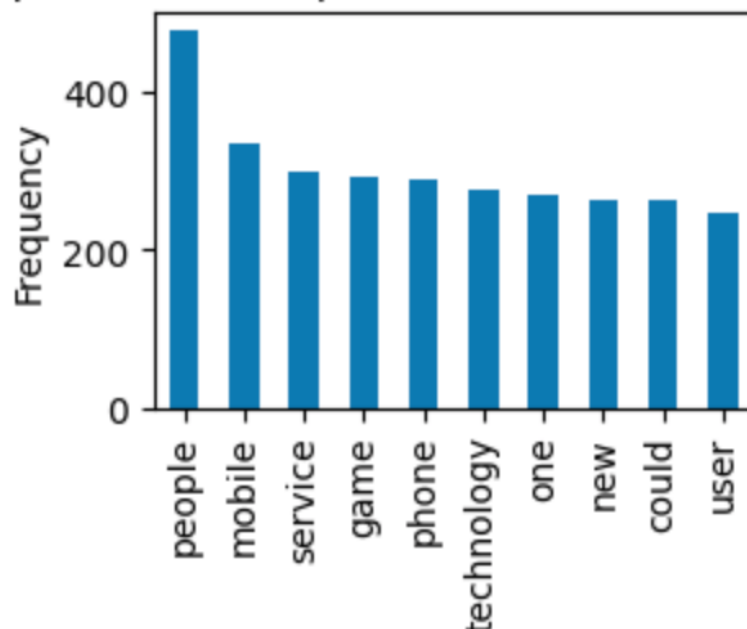


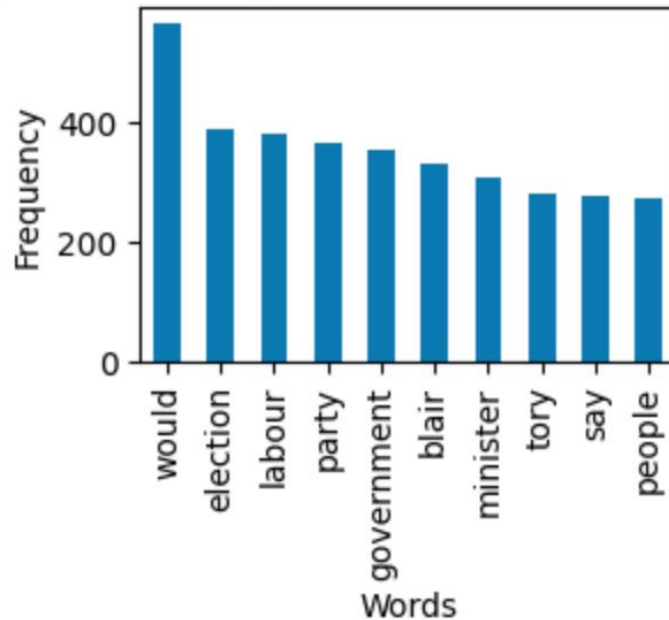
Рисунок 4.3 – Розподіл слів в новинах



Рисунок 4.4 - Розподіл слів в новинах

Top 10

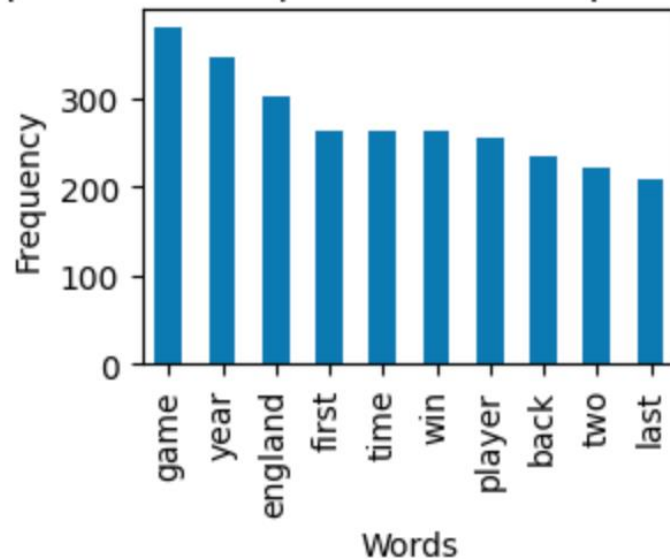
Top 10 Most Frequent Words in Politics Category



Рисунок

4.5 –

Top 10 Most Frequent Words in Sport Category



Розподіл слів в новинах

## 4.6 Тестування розробленої програми

Отже в попередніх розділах був проведений EDA аналіз, що дав зрозуміти, який розподіл категорій в нашому датасеті, які найбільш поширені слова у кожній категорії та їх середня довжина. Також була проведена

попередня обробка даних, що включала в себе етапи очищення, описані в розділах вище.

На етапі тестування матимемо дві моделі. Першою, а тобто базовою буде модель дерева рішень. А другою - згорткова нейронна мережа. При моделюванні нейронної мережі на вхід будуть подані дані, векторизовані трьома методами : TF-IDF, Bag of Words GloVe.

```
Time taken to train Decision Tree model: 0.21271920204162598 seconds
Decision Tree Accuracy: 79.62466487935657
```

Протестуємо модель дерева рішень(рис. 4.6):

Рисунок 4.6 – Результат роботи дерева рішень

Точність моделі 79,62%. Від цього результату будемо відштовхуватись при оцінці нейронної мережі.

Першою буде згорткова нейронна мережа, що приймає на вхід текст, векторизований за допомогою методу Bag of Words.

Другою буде згорткова нейронна мережа, що приймає на вхід текст, векторизований за допомогою методу TF-IDF.

Другою буде згорткова нейронна мережа, що приймає на вхід текст, векторизований за допомогою методу GloVe.

Для кожної нейронної мережі проведемо по 4 експеримента, та оберемо ту, що найкраще впоралась з завданням. Оцінимо середній час на навчання нейронної мережі(time taken to train model), а також середню точність(accuracy).

**Перший експеримент:**

- Bag of Words(рис. 4.7 )

```
Time taken to train model: 4.380444049835205 seconds
BoW Accuracy: 97.3154366016388
```

Рисунок 4.7 – Експеримент №1, нейронна мережа №1

- TF-IDF(рис. 4.8)

```
Time taken to train model: 3.8386359214782715 seconds
TF-IDF Accuracy: 96.78283929824829
```

Рисунок 4.8 – Эксперимент №1, нейронна мережа №2

- GloVe(рис. 4.9 )

Time taken to train model: 223.46087312698364 seconds  
GloVe Accuracy: 95.30201554298401

Рисунок 4.9 – Эксперимент №1, нейронна мережа №3

**Другий** експеримент:

- Bag of Words(рис. 4.10 )

Time taken to train model: 4.196646213531494 seconds  
BoW Accuracy: 95.97315192222595

Рисунок 4.10 – Эксперимент №2, нейронна мережа №1

- TF-IDF(рис. 4.11 )

Time taken to train model: 4.440446853637695 seconds  
TF-IDF Accuracy: 96.78283929824829

Рисунок 4.11 – Эксперимент №2, нейронна мережа №2

- GloVe(рис. 4.12 )

Time taken to train model: 347.8795518875122 seconds  
GloVe Accuracy: 95.30201554298401

Рисунок 4.12 – Эксперимент №2, нейронна мережа №3

**Третій** експеримент:

- Bag of Words(рис. 4.13 )

Time taken to train model: 3.790540933609009 seconds  
BoW Accuracy: 96.64429426193237

Рисунок 4.13 – Эксперимент №3, нейронна мережа №1

- TF-IDF(рис. 4.14 )

Time taken to train model: 6.198430061340332 seconds  
TF-IDF Accuracy: 96.51474356651306

Рисунок 4.14 – Эксперимент №3, нейронна мережа №2

- GloVe(рис. 4.15):

Time taken to train model: 268.83695793151855 seconds  
GloVe Accuracy: 95.63758373260498

Рисунок 4.15 – Эксперимент №3, нейронна мережа №3

**Четвертый эксперимент:**

- Bag of Words(рис. 4.16):

Time taken to train model: 4.043793201446533 seconds  
BoW Accuracy: 96.97986841201782

Рисунок 4.16 – Эксперимент №4, нейронна мережа №1

- TF-IDF(рис. 4.17):

Time taken to train model: 3.7691872119903564 seconds  
TF-IDF Accuracy: 96.51474356651306

Рисунок 4.17 – Эксперимент №4, нейронна мережа №2

- GloVe(рис. 4.18):

Time taken to train model: 225.32828092575073 seconds  
GloVe Accuracy: 95.30201554298401

Рисунок 4.18 – Эксперимент №4, нейронна мережа №3

**П'ятий эксперимент:**

- Bag of Words(рис. 4.19):

Time taken to train model: 4.432371139526367 seconds  
BoW Accuracy: 97.3154366016388

Рисунок 4.19 – Эксперимент №5, нейронна мережа №1

- TF-IDF(рис. 4.20):

Time taken to train model: 6.157184600830078 seconds  
TF-IDF Accuracy: 96.24664783477783

Рисунок 4.20 – Эксперимент №5, нейронна мережа №2

- GloVe(рис. 4.21 ):

Time taken to train model: 266.84219789505005 seconds  
GloVe Accuracy: 95.97315192222595

Рисунок 4.21 – Експеримент №5, нейронна мережа №3

Отже нейронні мережі змогли суттєво покращити результати базової моделі дерева рішень. Також дані тести показують, що на результат моделі також впливає метод векторизації, що використовується для обробки даних.

В таблиці 4.1 наведені середні значення всіх п'яти експериментів. З цієї таблиці випливає, що найкраще з задачею впоралась згортова нейронна мережа, на вхід якої було подано текст, векторизований алгоритмом Bag of Words. Ця модель є найоптимальнішою як з точки зору результату, так і з точки зору часу, витраченого на її тренування.

Модель, що отримала на вхід дані, векторизовані алгоритмом TF-IDF має трішки гірші результати, проте не сильно відстає від попереднього.

Алгоритм GloVe потребує досить багато часу на виконання, адже виконується на попередньо натренованих даних. Проте, як показали експерименти, точність алгоритму та час виконання є довгими порівняно з іншими двома методами.

Таблиця 4.1 - Результати роботи алгоритмів

	Decision Tree		Bag of Words		TF-IDF		GloVe	
	Час виконання,(сек)	Точність,%	Час виконання,(сек)	Точність,%	Час виконання,(сек)	Точність,%	Час виконання,(сек)	Точність,%
Експеримент 1	0,21	79,62%	4,38	97,31%	3,83	96,78%	223,46	95,30%
Експеримент 2	0,21	79,62%	4,19	95,97%	4,44	96,78%	347,88	95,30%
Експеримент 3	0,21	79,62%	3,79	96,64%	6,19	96,51%	268,84	95,63%
Експеримент 4	0,21	79,62%	4,04	96,97%	3,76	95,51%	225,32	95,30%
Експеримент 5	0,21	79,62%	4,43	97,32%	6,15	96,24%	226,84	95,97%
<b>Підсумок</b>	<b>0,21</b>	<b>79,62%</b>	<b>4,166</b>	<b>96,84%</b>	<b>4,874</b>	<b>96,36%</b>	<b>258,468</b>	<b>95,50%</b>

## 4.7 Висновки до розділу 4

В даному розділі було проведено EDA, з метою кращого ознайомлення з даними, також даний розділ був присвячений вибору програмного забезпечення для реалізації програми, а також тестування програми на обраних даних.

Для реалізації було обрано дві моделі: базова(проста), а також модель нейронної мережі. Зважаючи на результати тестувань можна побачити, що нейронна мережа добре впоралась з поставленою задачею, її точність є вищою, ніж точність базової моделі. Також експериментально був визначений найкращий метод векторизації тексту для прогнозування категорії новин.

## 5 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

### 5.1 Постановка завдання проектування

Цей розділ був спрямований на оцінювання ключових функціональних і кошторисних параметрів програмного засобу, що використовується для класифікації текстів за допомогою AI. Вказана програма була створена на основі мови програмування Python і фреймворку ?. Цей програмний продукт розроблено для використання на персональних комп'ютерах, що працюють на будь-якій ОС. Нижче представлено аналіз альтернативних варіантів розробки модуля для визначення найбільш ефективної стратегії створення програмного засобу.

### 5.2 Обґрунтування функцій програмного продукту

Основною функцією є створення програмного продукту, що використовуватиметься для класифікації текстових документів і матиме широке застосування в задачах цього типу.

Знаючи основну функцію можна виділити наступні основні функції даного програмного продукту:

- F1 – вибір мови програмування;
- F2 – вибір бібліотек та фреймворків;
- F3 – вибір середовища розробки;
- F4 – функціонал інтерфейсу програми;

Функція F<sub>1</sub>:

1. Python
2. R
3. C++

Функція F<sub>2</sub>:

1. Keras
2. Caffe
3. TensorFlow

Функція  $F_3$ :

1. Visual Studio
2. PyCharm
3. Jupyter Notebook

Функція  $F_4$ :

1. Максимум функціоналу (вікна, візуалізація, відображення помилок)
2. Мінімум функціоналу (лише відображення результату)

### 5.3 Варіанти реалізації основних функцій

Варіанти реалізації основних функцій наведені у морфологічній карті на рисунку 1. Дана карта показує будь-які можливі комбінації вирішення реалізації основних функцій програми.

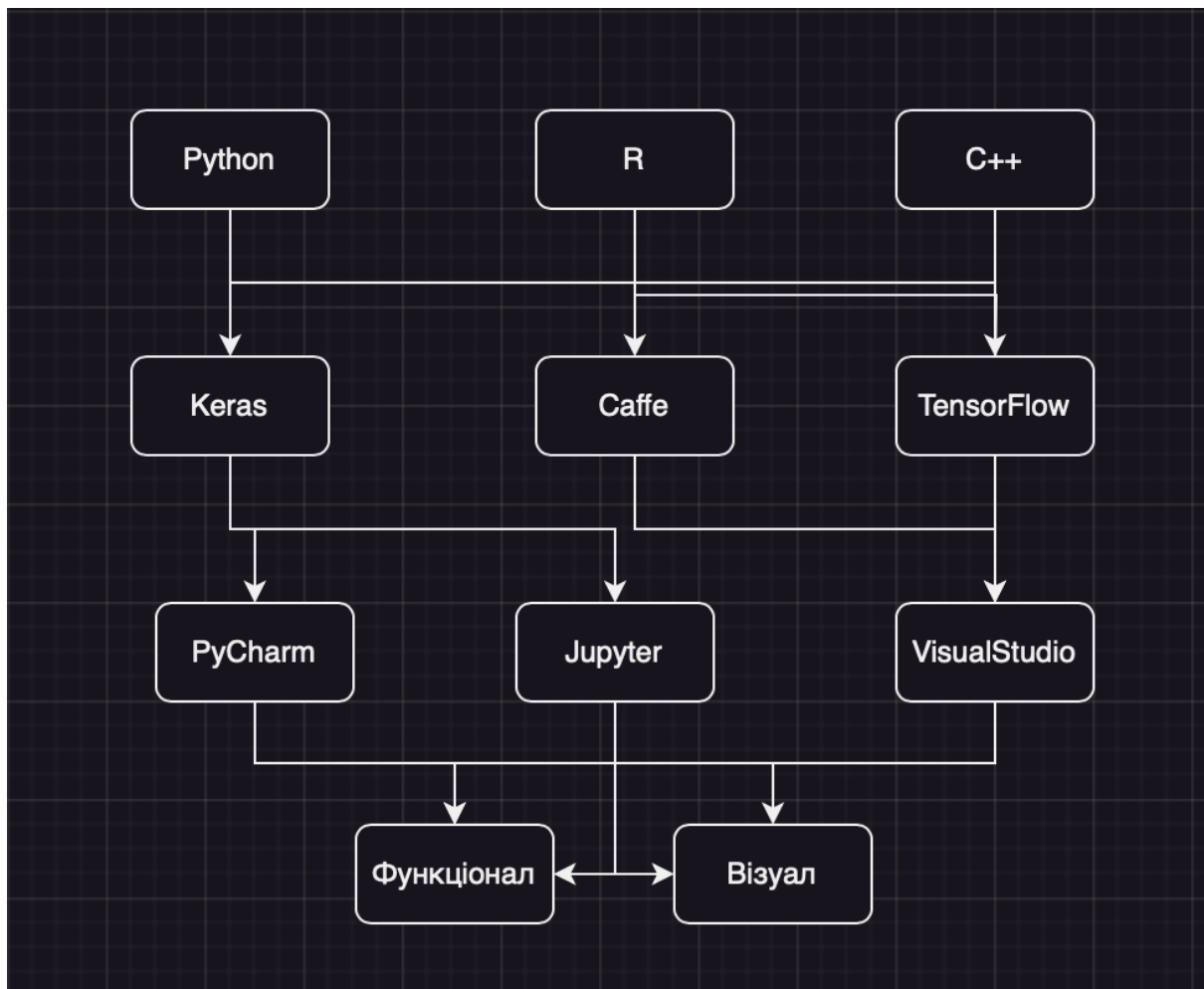


Рис. 1. Морфологічна карта.

Отримавши таку морфологічну карту можна побудувати позитивно-негативну матрицю, її представлено у таблиці 1:

Осн овні функ ції	Варі анти реал ізаці ї	Переваги	Недоліки
F1	1	Широкий вибір бібліотек для роботи з AI, гнучка мова, легко вивчити, активна спільнота	Не така швидка як низькорівневі мови програмування
	2	Мова, що широко використовується при розв'язанні статистичних та аналітичних задач. Потужні інструменти візуалізації	Менша кількість бібліотек для AI, не така популярна як Python
	3	Висока швидкість виконання, можливість низькорівневого програмування	Складна для вивчення, потребує розуміння пам'яті і її керування. Рідше вживається для нейронних мереж
F2	1	Легкість використання, модульність, гнучкість	Не такий швидкий, як TensorFlow
	2	Безкоштовна та надійна. Підтримує C++	Необхідність додаткової інсталяції, низький рівень користувацької підтримки
	3	Масштабований, активна спільнота, підтримує високопродуктивні обчислення	Складний для вивчення
F3	1	Повнофункціональне середовище розробки, інтегровані засоби тестування	Великий та ресурсоємкий

	2	Спрощує процес розробки, підтримує багато мов програмування	Часто потребує платної версії для повного набору функцій
	3	Безкоштовний, зручний у використанні	Потрібні додаткові плагіни для кращої роботи. Менше функцій ніж в PyCharm
F4	1	Кращий контроль над процесом, візуально естетичний	Більше часу на розробку
	2	Простота використання, менше часу на розробку	Обмежені можливості для відстежування помилок, менш гнучкий

Табл. 1. Позитивно-негативна матриця.

На основі аналізу позитивно-негативної матриці робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, що вони не відповідають поставленим перед програмним продуктом задачам. Ці варіанти відзначені у морфологічній карті.

Функція :

Перевагу даємо Python, оскільки це найбільш зрозуміла та проста мова програмування, яка має величезну спільноту, що безумовно допоможе при реалізації проекту.

Функція :

Оскільки для написання програмного коду було обрано саме Python, то варіант 2 не підлягає дискусії, залишається Keras та TensorFlow. З них оберемо перший варіант, адже з ним легше почати працювати одразу і він є досить потужним.

Функція :

Отже, другий та третій варіанти стають обраними, оскільки середовище розробки - це лише питання особистих переваг, і воно має мінімальний вплив на якість та вартість продукту.

Функція :

Візуалізація для цього проекту є неважливою, то мінімум функціоналу є оптимальним рішенням, тому що це надасть можливість зосередитися на можливостях програми.

Отже, виходячи із отриманих даних, будемо розглядати наступні комбінації створення програмного продукту:

- $F_1(1) - F_2(1) - F_3(2) - F_4(2)$
- $F_1(1) - F_2(3) - F_3(3) - F_4(2)$
- $F_1(1) - F_2(1) - F_3(3) - F_4(2)$
- $F_1(1) - F_2(3) - F_3(2) - F_4(2)$

## 5.4 Обґрунтування системи параметрів програмного продукту

Виходячи з вищенаведених даних, визначаємо ключові параметри для вибору, які слугуватимуть основою для обчислення коефіцієнта технічного рівня.

Для характеристики програмного продукту використовуватимемо такі параметри:

- X1 – швидкодія мови програмування;
- X2 – об'єм пам'яті для обчислень та збереження даних;
- X3 – час тренування даних;
- X4 – можливий об'єм програмного коду.

### 5.4.1 Кількісна оцінка параметрів

Найгірші, середні та найкращі значення параметрів вибираються з огляду на вимоги замовника та умови експлуатації програмного продукту, як це зазначено в таблиці 2.

Назва параметра	Умовні позначення	Одиниці виміру	Значення параметра		
			гірші	середні	кращі
Швидкодія мови програмування	X1	Оп/мс	940	1432	1712

Об'єм пам'яті для збереження даних	X2	Мб	141	79	66
Час запуску і обробки задач програмою	X3	с	420	382	265
Потенційний об'єм програмного коду	X4	рядків коду	460	423	395

За даними таблиці 2 будуються графічні характеристики параметрів – рис. 2 – рис. 5

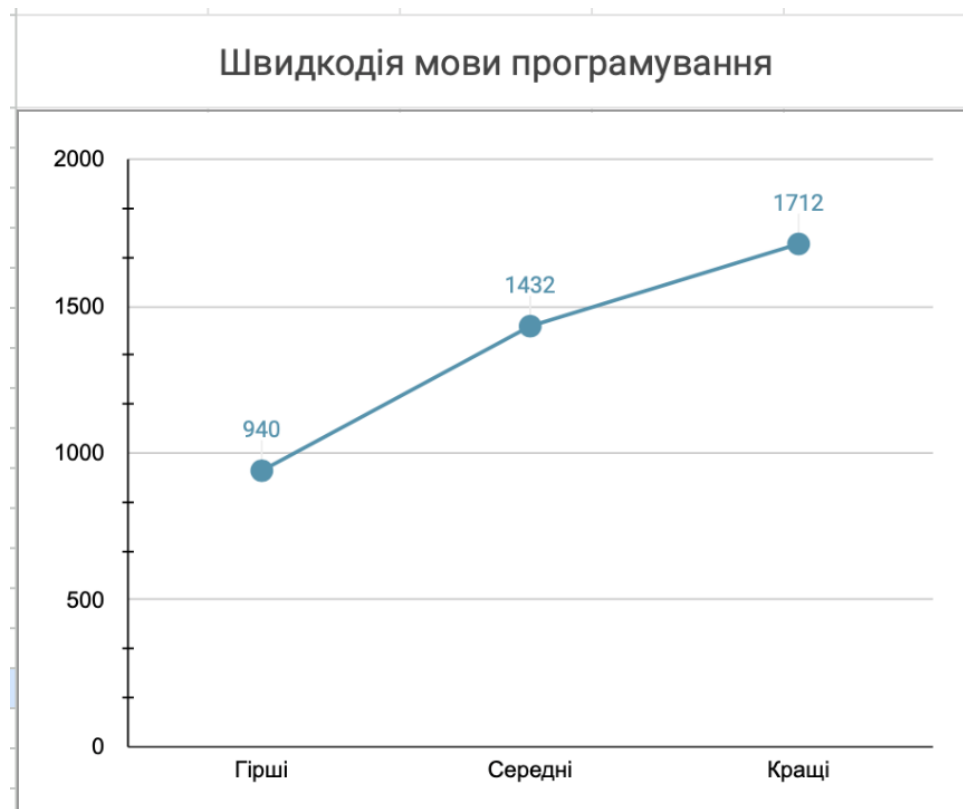


Рис. 2 Швидкодія мови програмування

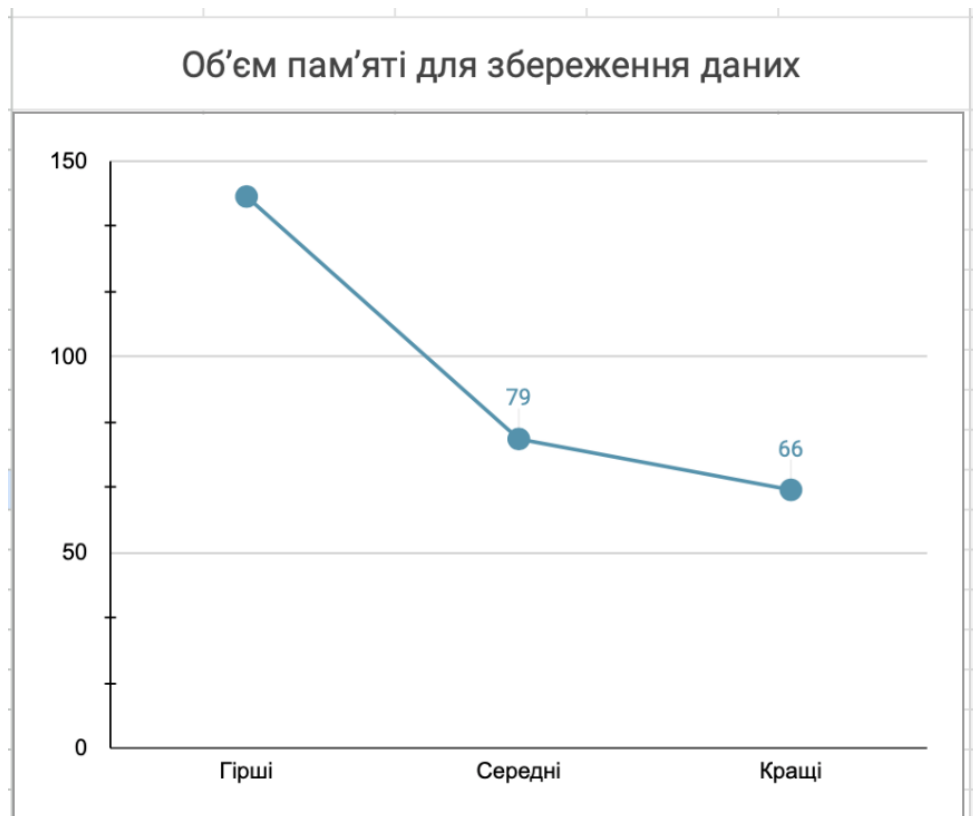


Рис. 3. Об'єм пам'яті для збереження даних

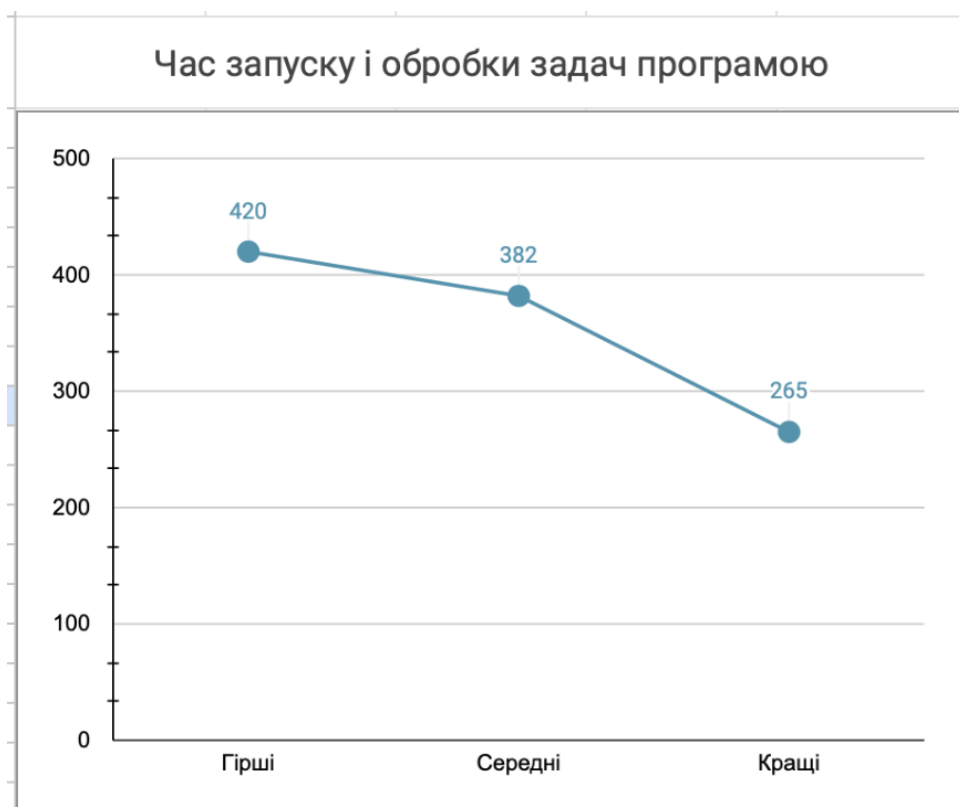


Рис. 4. Час запуску і обробки задач програмою

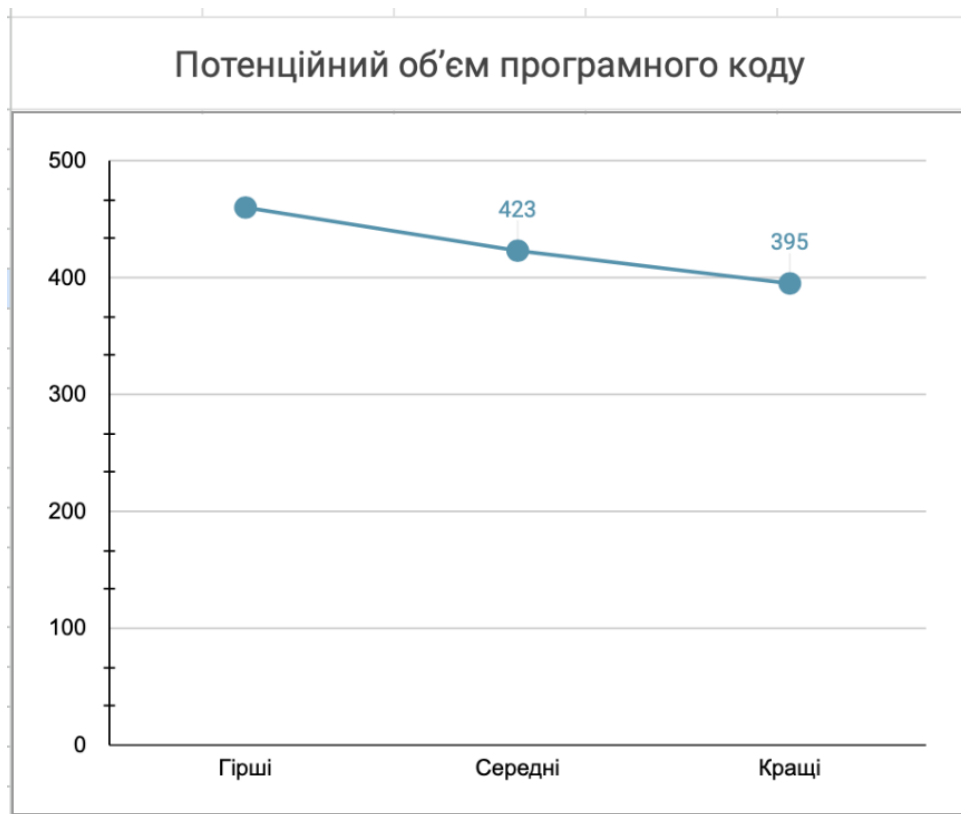


Рис. 5. Потенційний об'єм програмного коду

## 5.5 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який має найбільш зручний інтерфейс та зрозумілу взаємодію з користувачем

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 5 людей. Визначення коефіцієнтів значущості передбачає:

- визначення рівня значимості параметра шляхом присвоєння різних рангів;
- перевірку придатності експертних оцінок для подальшого використання;
- визначення оцінки попарного пріоритету параметрів;
- обробку результатів та визначення коефіцієнту значимості.

Таблиця 3 - Результати експертного ранжування

Параметр	Ранг параметра за оцінкою експерта					Сума рангі в	Відхилення	
	1	2	3	4	5		$\Delta_i$	$\Delta_i^2$
X1	4	4	4	4	4	20	6	36
X2	3	4	4	3	2	16	2	4
X3	2	2	3	2	4	13	-1	1
X4	1	1	1	2	2	7	-7	49
Разом	10	11	12	11	12	56	0	90

Для перевірки ступеню достовірності експертних оцінок, визначимо наступні параметри:

$$R_i = \sum_{j=1}^N r_{ij} = 56,$$

а) сума рангів кожного з параметрів і загальна сума рангів:

де – ранг  $i$ -го параметра, визначений  $j$ -м експертом;

$N$  – число експертів.

$$T = \frac{1}{n} R_i = 14$$

б) середня сума рангів  $T$ :

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T$$

г) загальна сума квадратів відхилення:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 90}{5^2(4^3 - 4)} = 0,72 > W_k = 0,67$$

д) коефіцієнт узгодженості (конкордації):

Ранжирування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67. Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 4. Числові значення, що визначає ступінь переваги  $i$ -го параметра над  $j$ -тим, визначається за формулою:

$$a_{ij} = \{1,5 \ x_i > x_j; 1,0 \ x_i = x_j; 0,5 \ x_i < x_j$$

З отриманих числових оцінок переваги складемо матрицю . Для кожного параметра розрахунок вагомості проводиться за наступною формулою:

$$K_{B_i} = \frac{b_i}{\sum_{i=1}^n b_i},$$

де  $b_i = \sum_{j=1}^N a_{ij}$  – вагомість  $i$ -го параметра за результатами оцінок всіх експертів;

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%). На другому і наступних кроках відносні оцінки розраховуються за наступною

$$K_{B_i} = \frac{b'_i}{\sum_{i=1}^n b'_i},$$

формулою:

$$b'_i = \sum_{j=1}^N a_{ij} b_j.$$

де

Як видно з таблиці 4.5, різниця значень коефіцієнтів вагомості після другої ітерації не перевищує 2%, тому додаткові ітерації не потрібні.

Таблиця 4.5–Розрахунок вагомості параметрів

$X_i$	$X_j$				Перша ітерація		Друга ітерація	
	$X_1$	$X_2$	$X_3$	$X_4$	$b_i$	$K_{vi}$	$b_i^1$	$K_{vi}^1$
$X_1$	1,0	1,5	1,5	1,5	5,5	0,344	21,25	0,36
$X_2$	0,5	1,0	1,5	1,5	4,5	0,281	16,25	0,275
$X_3$	0,5	0,5	1,0	1,5	3,5	0,218	12,25	0,208
$X_4$	0,5	0,5	0,5	1,0	2,5	0,156	9,25	0,157
Всього:					16	1	59	1

### 1. Аналіз рівня якості варіантів реалізації функцій

Рівень якості кожного варіанту виконання основних функцій визначається окремо.

Абсолютне значення параметрів  $X_1$  (швидкодія мови програмування) та  $X_4$  (Потенційний об'єм програмного коду) відповідають технічним вимогам умов функціонування даного ПП.

Абсолютне значення параметра X2 та X3 обрано не найгіршим (не максимальним) тобто це варіанти А або Б

Коефіцієнт технічного рівня якості для кожного варіанта реалізації ПП розраховується за формулою:

$$K_{TP} = \sum_{i=1}^n K_{B_{i,j}} B_{i,j}$$

де  $n$  – кількість параметрів;

$K_{B_{i,j}}$  – коефіцієнт вагомості  $i$ -го параметра;

$B_{i,j}$  – оцінка  $i$ -го параметра в балах.

Таблиця 6 - Розрахунок показників рівня якості

Основні функції	Варіант реалізації функції	Параметри	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	1	X1	1712	10	0,36	3,6
F2	1	X2	66	10	0,275	2,75
	3	X2	79	5	0,208	1,04
F3	2	X3	382	10	0,208	2,08
	3	X3	382	5	0,175	0,73
F4	1	X4	359	10	0,175	1,75

За цими даними визначаємо рівень якості кожного з варіантів:

$$- F1(1) - F2(1) - F3(2) - F4(2) = 3,6 + 2,75 + 2,08 + 1,75 = 5,35 + 4,83$$

$$- F1(1) - F2(3) - F3(3) - F4(2) = 3,6 + 1,04 + 0,73 + 1,75 = 5,35 + 1,77$$

$$- F1(1) - F2(1) - F3(3) - F4(2) = 3,6 + 2,75 + 0,73 + 1,75 = 5,35 + 3,48$$

$$- F1(1) - F2(3) - F3(2) - F4(2) = 3,6 + 1,04 + 2,08 + 1,75 = 5,35 + 3,12$$

Отже, найкращим є перший варіант, для якого коефіцієнт технічного рівня має найбільше значення.

## 5.6 Економічний аналіз варіантів розробки програмного продукту

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

1. Розробка проекту програмного продукту;
2. Розробка програмної оболонки;

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 3.

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 використовує інформацію у вигляді даних.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань. Загальна трудомісткість обчислюється як

$$T_0 = T_p \cdot K_{\Pi} \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТ.М},$$

де  $T_p$  – трудомісткість розробки ПП;  $K_{\Pi}$  – поправочний коефіцієнт;  $K_{СК}$  – коефіцієнт на складність вхідної інформації;  $K_M$  – коефіцієнт рівня мови програмування;  $K_{СТ}$  – коефіцієнт використання стандартних модулів і прикладних програм;  $K_{СТ.М}$  – коефіцієнт стандартного математичного забезпечення

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру ступеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює:  $T_p = 41$  людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання:  $K_{\Pi} = 1,8$ . Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх завдань рівний 1:  $K_{СК} = 1$ . Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта  $K_{СТ} = 0,8$ . Тоді загальна трудомісткість програмування першого завдання дорівнює:

$$T_1 = 41 \cdot 1,8 \cdot 0,8 = 59,04 \text{ людино-днів.}$$

Для другого завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто  $T_P = 18$  людино-днів,  $K_{II} = 0,9$ ,  $K_{СК} = 1$ ,  $K_{СТ} = 0,8$ :

$$T_2 = 18 \cdot 0,9 \cdot 0,8 = 12,96 \text{ людино-днів.}$$

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$$T_I = (59,04 + 12,96) \cdot 8 = 576 \text{ людино-годин;}$$

$$T_{II} = (59,04 + 12,96) \cdot 8 = 576 \text{ людино-годин;}$$

Оскільки трудомісткість обох варіантів рівна то її можна об'єднати в одну групу:

$$T_0 = (59,04 + 12,96) \cdot 8 = 576 \text{ людино-годин;}$$

В розробці беруть участь: один програміст з окладом 30000 грн. та один аналітик даних з окладом 42500 грн. Визначимо зарплату за годину за формулою:

$$CЧ = \frac{M}{T_m \cdot t} \text{ грн.}$$

де  $M$  – місячний оклад працівників;

$T_m$  – кількість робочих днів на місяць;

$t$  – кількість робочих годин в день.

$$CЧ = \frac{30000+42500}{3 \cdot 21 \cdot 8} = 143,85 \text{ грн.}$$

Тоді, розрахуємо заробітну плату за формулою:

$$C_{ЗП} = CЧ \cdot T_i \cdot K_D, \quad (31)$$

де  $CЧ$  – величина погодинної оплати праці програміста;

$T_i$  – трудомісткість відповідного завдання;

$K_D$  – норматив, який враховує додаткову заробітну плату.

Зарплата розробників становить:

$$C_{3П} = 143,85 \cdot 576 \cdot 1,2 = 99428 \text{ грн.}$$

Відрахування становить 22%:

$$C_{ВІД} = C_{3П} \cdot 0,22 = 99428 \cdot 0,22 = 21874 \text{ грн.}$$

Тепер визначимо витрати на оплату однієї машино-години. ( $C_M$ )

Так як одна ЕОМ обслуговує одного працівника з окладом 30000 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$$C_{Г} = 12 \cdot M \cdot K_3 = 12 \cdot 30000 \cdot 0,2 = 72000 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$C_{3П} = C_{Г} \cdot (1 + K_3) = 72000 \cdot (1 + 0,2) = 86400 \text{ грн.}$$

Відрахування становить 22%:

$$C_{ВІД} = C_{3П} \cdot 0,22 = 72000 \cdot 0,22 = 19008 \text{ грн.}$$

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 15000 грн.

$$C_A = K_{ТМ} \cdot K_A \cdot Ц_{ПР} = 1,15 \cdot 0,25 \cdot 15000 = 4312 \text{ грн.,}$$

де  $K_{ТМ}$  – коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача;  $K_A$  – річна норма амортизації;  $Ц_{ПР}$  – договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$$C_P = K_{ТМ} \cdot Ц_{ПР} \cdot K_P = 1,15 \cdot 15000 \cdot 0,05 = 862 \text{ грн.,}$$

де  $K_P$  – відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$T_{ЕФ} = (D_K - D_B - D_C - D_P) \cdot t_3 \cdot K_B = (365 - 104 - 12 - 16) \cdot 8 \cdot 0,95 = 1770,8 \text{ годин,}$$

де  $D_K$  – календарна кількість днів у році;  $D_B$ ,  $D_C$  – відповідно кількість вихідних та святкових днів;  $D_P$  – кількість днів планових ремонтів устаткування;  $t$  – кількість робочих годин в день;  $K_B$  – коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$$C_{ЕЛ} = T_{ЕФ} \cdot N_C \cdot K_3 \cdot Ц_{ЕН} = 1770,8 \cdot 0,55 \cdot 0,3 \cdot 4,87 = 1422 \text{ грн.,}$$

де  $N_C$  – середньо-споживча потужність приладу;  $K_3$  – коефіцієнтом зайнятості приладу;  $C_{ЕН}$  – тариф за 1 кВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$$C_H = C_{ПР} \cdot 0,67 = 15000 \cdot 0,67 = 10050 \text{ грн.}$$

Тоді, річні експлуатаційні витрати будуть:

$$C_{ЕКС} = C_{ЗП} + C_{ВІД} + C_A + C_P + C_{ЕЛ} + C_H$$

$$C_{ЕКС} = 86400 + 19008 + 4312 + 862 + 1422 + 10050 = 122054 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$C_{М-Г} = C_{ЕКС} / T_{ЕФ} = 122054 / 1770,8 = 68,92 \text{ грн/год.}$$

Оскільки в даному випадку всі роботи, які пов'язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу складає:

$$C_M = C_{М-Г} \cdot T$$

$$C_M = 68,92 \cdot 576 = 39701 \text{ грн.};$$

Накладні витрати складають 67% від заробітної плати:

$$C_H = C_{ЗП} \cdot 0,67$$

$$C_H = 86400 \cdot 0,67 = 57888 \text{ грн.};$$

Отже, вартість розробки ПП становить:

$$C_{ПП} = C_{ЗП} + C_{ВІД} + C_M + C_H$$

$$C_{ПП} = 86400 + 19008 + 39701 + 57888 = 150897 \text{ грн.};$$

## 5.7 Висновки до розділу 5

Проект передбачав економічний розділ, в рамках якого було здійснено техніко-економічне обґрунтування обраного методу функціонально-вартісного аналізу. Це поглибило наші теоретичні знання в області економіки та організації виробництва.

Проведений аналіз ключових функцій програмного продукту дозволив нам виокремити чотири найефективніші варіанти реалізації. Перший варіант виявився особливо продуктивним, оскільки він гарантує найвище значення

коефіцієнта техніко-економічного рівня з витратами на розробку у розмірі 150897 грн.

Цей варіант включає використання мови програмування Python і модуля Keras для машинного навчання на Python. Розробка програмного продукту здійснюється за допомогою PyCharm. При цьому, ми зосереджуємося на розширенні функціоналу продукту, приділяючи менше уваги його зовнішньому вигляду.

## ВИСНОВКИ ПО РОБОТІ

В ході виконання дипломної роботи було проведено аналіз, а також порівняльний аналіз методів та алгоритмів, що використовуються для класифікації тексту. Було визначено набір інструментів, що використовуватимуться для реалізації проекту, а також спроектовано та розроблено програму глибокої нейронної мережі, що виконуватиме задачу класифікації тексту.

Матеріал цього диплому не обмежений лише моделями та містить дослідження такого важливого етапу, як попередня обробка даних перед подачею їх на вхід моделі. В даній роботі було проведено огляд основних етапів попередньої обробки даних, їх важливість, а також наведено приклади мовою python.

Наступною важливою частиною даної роботи є дослідження методів векторизації тексту. Описані методи векторизації, такі як Bag-of-Words, TF-IDF, Word2Vec та GloVe. В роботі наведено висновки, які стосуються розглянутих методів та алгоритмів векторного представлення тексту. Також експериментально доказано, що метод векторизації тексту впливає на точність класифікації нейронною мережею тексту.

Головною частиною даної роботи було створення моделі глибокої навчання. В даній роботі було чітко визначено, що таке глибоке навчання, які відмінності між глибоким навчанням та машинним навчанням, а також які моделі нейронних мереж вважаються глибокими. На основі цих даних було спроектовано згорткову нейронну мережу та реалізовано з використанням мови Python та бібліотеки Keras. Хоч згорткові нейронні мережі в своїй більшості застосовуються для класифікації зображень, проте вони показали хороший результат в задачі класифікації тексту порівняно з алгоритмом дерева рішень.

В якості перспективи розвитку даної моделі передбачається класифікація документів, що необхідно просканувати з зображення.

## ПЕРЕЛІК ПОСИЛАНЬ

1. Chollet, F., "Deep Learning with Python," Manning Publications, 2018.
2. Goodfellow, I., Bengio, Y., and Courville, A., "Deep Learning," MIT Press, 2016, [Online] Available: <http://www.deeplearningbook.org>.
3. Aman, V., "Step-by-Step Text Classification using different models and compare them." Towards Data Science, 2020, [Online] Available: <https://medium.com/analytics-vidhya/step-by-step-text-classification-using-different-models-and-compare-them-8a34204c34f8>.
4. Aggarwal, C.C., and Zhai, C., "A Survey of Text Classification Algorithms," in Mining Text Data, Springer, 2012, pp. 163-222.
5. Manning, C. D., Raghavan, P., and Schütze, H., "Introduction to Information Retrieval," Cambridge University Press, 2008.
6. Harris, Zellig (1954). "Distributional Structure". Word. 10 (2/3): 146–62. doi:10.1080/00437956.1954.11659520
7. Rohith, G., "Naive Bayes Classifier." Towards Data Science, 2018, [Online] Available: <https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7>.
8. "Теорія Випадкових Множин," Головна електронна бібліотека України, [Online] Available: <http://ebooks.git-elt.hneu.edu.ua/tvms/p-2-5.html>.

9. "Support Vector Machines(SVM): A Complete Guide for Beginners," Analytics Vidhya, 2021, [Online] Available:  
<https://www.analyticsvidhya.com/blog/2021/10/support-vector-machinessvm-a-complete-guide-for-beginners/>.
10. S. Z. Li, R. Chu, S. Liao, and L. Zhang, "Illumination Invariant Face Recognition Using Near-Infrared Images," in IEEE Transactions on Pattern Analysis and Machine Intelligence, April 2007, doi: 10.1109/TPAMI.2007.1045, [Online] Available:[https://link.springer.com/chapter/10.1007/978-3-540-72383-7\\_148](https://link.springer.com/chapter/10.1007/978-3-540-72383-7_148).
11. K. D. Foote, "A Brief History of Neural Networks," DATAVERSITY, 09-Nov-2021. [Online]. Available: <https://www.dataversity.net/a-brief-history-of-neural-networks/>. [Accessed: 01-Jun-2023].
12. "Дані для ML-моделей в NLP, recommendation і CV — пошук, підготовка і трансформація" DOU, 2022, [Online] Available:  
<https://dou.ua/forums/topic/38872/>.
13. Pandey, P., "Text Preprocessing in Natural Language Processing using Python," Towards Data Science, 2020, [Online] Available:  
<https://towardsdatascience.com/text-preprocessing-in-natural-language-processing-using-python-6113ff5decd8>.
14. "Text Preprocessing," Codecademy, [Online] Available:  
<https://www.codecademy.com/learn/dsnlp-text-preprocessing/modules/nlp-text-preprocessing/cheatsheet>.
15. A. Smokin, "Стоп-слова," [Online] Available:  
<https://alexsmokinof.lviv.ua/стоп-слова/>.

16. "Stemming vs Lemmatization in NLP: Must Know Differences," Analytics Vidhya, June 2022, [Online] Available:  
<https://www.analyticsvidhya.com/blog/2022/06/stemming-vs-lemmatization-in-nlp-must-know-differences/#:~:text=Stemming%20is%20a%20process%20that,%27%20would%20return%20%27Car%27.>
17. "Getting Started with Text Vectorization," Towards Data Science, [Online] Available: <https://towardsdatascience.com/getting-started-with-text-vectorization-2f2efbec6685>.
18. J. Pennington, R. Socher, C. D. Manning et al., "GloVe: Global Vectors for Word Representation," Journal of Machine Learning Research, 2011, [Online] Available: <https://aclanthology.org/D14-1162/>.
19. "Keras: Deep Learning for humans," Keras, 2023, [Online] Available: <https://keras.io/guides/>.
20. "Pros and Cons of Python: A Definitive Python Web Development Guide," PixelCrayons, 2023, [Online] Available: <https://www.pixelcrayons.com/blog/python-pros-and-cons/>.
21. P. Singh, "Baseline Models: Your Guide for Model Building," Towards Data Science, 2023, [Online] Available: <https://towardsdatascience.com/baseline-models-your-guide-for-model-building-1ec3aa244b8d>.
22. "Exploratory Data Analysis," IBM, [Online] Available: <https://www.ibm.com/topics/exploratory-data-analysis>.

23. "Learn AI BBC," Kaggle, [Online] Available:  
<https://www.kaggle.com/competitions/learn-ai-bbc/data>.
24. Spärck Jones, K. (1972). "A Statistical Interpretation of Term Specificity and Its Application in Retrieval". Journal of Documentation, [Online] Available:  
<https://www.emerald.com/insight/content/doi/10.1108/eb026526/full/html>
25. Mikolov, T., Chen, K., Corrado, G., and Dean, J., "Efficient Estimation of Word Representations in Vector Space," arXiv preprint arXiv:1301.3781, 2013, [Online] Available: <https://arxiv.org/abs/1301.3781>.
26. Kolisnyk, V.V., Matsiuk, O.V., & Subbotin, O.Y. (2015). Neural networks: an approach to modeling and utilization. Bulletin of Zaporizhzhia National Technical University. [Online] Available:  
[http://eir.zntu.edu.ua/bitstream/123456789/6800/1/Subbotin\\_Neural.pdf](http://eir.zntu.edu.ua/bitstream/123456789/6800/1/Subbotin_Neural.pdf)
27. Techiepedia. (2023, June 6). Binary Image Classifier: CNN using TensorFlow. [Online] Available: <https://medium.com/techiepedia/binary-image-classifier-cnn-using-tensorflow-a3f5d6746697>

## ДОДАТОК А

Лістинг прикладів векторизації та попередньої обробки тексту

```
contractions_dict = {
    "don't": "do not",
    "can't": "cannot",
    "isn't": "is not",
    "aren't": "are not",
    "won't": "will not",
    "couldn't": "could not",
    "shouldn't": "should not",
    "wouldn't": "would not",
    "didn't": "did not",
    "doesn't": "does not"
    # Та інші
}

def expand_contractions(text):
    words = text.split()
    expanded_text = []
    for word in words:
        if word.lower() in contractions_dict:
            expanded_text.append(contractions_dict[word.lower()])
        else:
            expanded_text.append(word)
    return ' '.join(expanded_text)
```

```
input_text = "I don't know if I can make it."
expanded_text = expand_contractions(input_text)
print(expanded_text)
```

```
def lowercase(text):
    return text.lower()
```

```
input_text = "HELLO, WORLD!"
output_text = lowercase(input_text)
print(output_text)
```

```
import re
```

```
def remove_punctuations(text):
    cleaned_text = re.sub(r'^[\w\s]', "", text)
    return cleaned_text
```

```
sentence = "Hello, world!"
cleaned_sentence = remove_punctuations(sentence)
print(cleaned_sentence)
```

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
```

```
def remove_stopwords(text):
    # Завантажуємо список зупинних слів
    nltk.download('stopwords')
```

```
stop_words = set(stopwords.words('english'))

# Токенізуємо текст на окремі слова
tokens = word_tokenize(text)

# Видаляємо зупинні слова
filtered_tokens = [word for word in tokens if word.lower() not in stop_words]

# З'єднуємо слова назад в рядок
filtered_text = ''.join(filtered_tokens)
return filtered_text

# Приклад використання
text = "This is an example sentence demonstrating the removal of stopwords."
filtered_text = remove_stopwords(text)
print(filtered_text)

import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

def remove_stopwords(text):
    # Завантажуємо список слів
    nltk.download('stopwords')
    stop_words = set(stopwords.words('english'))

    # Токенізуємо текст
    tokens = word_tokenize(text)

    # Видаляємо слова
```

```
filtered_tokens = [word for word in tokens if word.lower() not in stop_words]

# З'єднуємо слова назад в рядок
filtered_text = ''.join(filtered_tokens)

return filtered_text

text = "This is he she it demonstration lover."
filtered_text = remove_stopwords(text)
print(filtered_text)

from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.tokenize import word_tokenize

def stemming_and_lemmatization(text):
    # Ініціалізуємо об'єкти для stemming та lemmatization
    stemmer = PorterStemmer()
    lemmatizer = WordNetLemmatizer()

    # Токенізуємо текст на окремі слова
    tokens = word_tokenize(text)

    stemmed_words = []
    lemmatized_words = []

    for word in tokens:
        # Stemming
        stemmed_word = stemmer.stem(word)
        stemmed_words.append(stemmed_word)
```

```
# Lemmatization
lemmatized_word = lemmatizer.lemmatize(word)
lemmatized_words.append(lemmatized_word)

return stemmed_words, lemmatized_words

# Приклад використання
text = "The quick brown foxes jumped over the lazy dogs"
stemmed_words, lemmatized_words = stemming_and_lemmatization(text)

print("Stemmed words:", stemmed_words)
print("Lemmatized words:", lemmatized_words)

from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.tokenize import word_tokenize

def stemming_and_lemmatization(text):
    # Ініціалізуємо об'єкти для stemming та lemmatization
    stemmer = PorterStemmer()
    lemmatizer = WordNetLemmatizer()

    # Токенізуємо текст
    tokens = word_tokenize(text)

    stemmed_words = []
    lemmatized_words = []

    for word in tokens:
        # Stemming
        stemmed_word = stemmer.stem(word)
```

```
stemmed_words.append(stemmed_word)
```

```
# Lemmatization
```

```
lemmatized_word = lemmatizer.lemmatize(word)
```

```
lemmatized_words.append(lemmatized_word)
```

```
return stemmed_words, lemmatized_words
```

```
# Приклад використання
```

```
text = "The quick brown foxes jumped over the lazy dogs"
```

```
stemmed_words, lemmatized_words = stemming_and_lemmatization(text)
```

```
print("Stemmed words:", stemmed_words)
```

```
print("Lemmatized words:", lemmatized_words)
```

```
from sklearn.feature_extraction.text import CountVectorizer
```

```
# Задаємо список текстових документів
```

```
documents = [
```

```
    "Це перший документ.",
```

```
    "Це другий документ.",
```

```
    "А це третій документ."
```

```
]
```

```
# Ініціалізуємо CountVectorizer
```

```
vectorizer = CountVectorizer()
```

```
# Застосовуємо Bag-of-Words до документів
```

```
bag_of_words = vectorizer.fit_transform(documents)
```

```
# Отримуємо слова, які були використані для побудови Bag-of-Words
```

```
feature_names = vectorizer.get_feature_names_out()
```

```
# Виводимо результати
```

```
print("Bag-of-Words:")
```

```
print(bag_of_words.toarray())
```

```
print("\nСлова:")
```

```
print(feature_names)
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
# Задані документи
```

```
documents = [
```

```
    "Це перший документ.",
```

```
    "Це другий документ.",
```

```
    "Це третій документ.",
```

```
    "Це останній документ."
```

```
]
```

```
# Ініціалізуємо TfidfVectorizer
```

```
vectorizer = TfidfVectorizer()
```

```
# Обчислюємо TF-IDF
```

```
tfidf_matrix = vectorizer.fit_transform(documents)
```

```
# Виводимо результат
```

```
feature_names = vectorizer.get_feature_names_out()
```

```
for i, doc in enumerate(documents):
```

```
    print(f"TF-IDF для документа {i+1}:")
```

```
        for j, term in enumerate(feature_names):
```

```
tfidf = tfidf_matrix[i, j]
if tfidf != 0:
    print(f"{term}: {tfidf:.2f}")
print()

from gensim.models import Word2Vec
sentences = [['I', 'love', 'machine', 'learning'],
             ['I', 'love', 'deep', 'learning'],
             ['I', 'enjoy', 'NLP'],
             ['I', 'enjoy', 'computer', 'vision']]

# Задаємо параметри моделі Word2Vec
model = Word2Vec(sentences, min_count=1)

# Отримуємо вектор для певного слова
vector = model.wv['learning']

# Знаходимо схожі слова для певного слова
similar_words = model.wv.most_similar('learning')
print(similar_words)

import gensim
import numpy as np

corpus = [
    ['I', 'like', 'apples'],
    ['I', 'like', 'oranges'],
    ['I', 'enjoy', 'eating', 'apples'],
    ['I', 'enjoy', 'eating', 'bananas'],
    ['I', 'like', 'to', 'eat', 'apples'],
```

```
['I', 'like', 'to', 'eat', 'oranges'],  
['I', 'like', 'to', 'eat', 'bananas']  
]
```

```
# Побудова моделі GloVe
```

```
model = gensim.models.Word2Vec(corpus, min_count=1, vector_size=100,  
workers=4, window=5, sg=1)
```

```
# Отримання матриці вкладень слів
```

```
embedding_matrix = np.zeros((len(model.wv.key_to_index), model.vector_size))
```

```
for i, word in enumerate(model.wv.key_to_index):
```

```
    embedding_vector = model.wv[word]
```

```
    embedding_matrix[i] = embedding_vector
```

```
for word in model.wv.key_to_index:
```

```
    print(f"Word: {word}, Embedding: {model.wv[word]}")
```

## ДОДАТОК Б

Код готової моделі для класифікації тексту

```
#!/usr/bin/env python
# coding: utf-8

# ## Data upload & preparation

import ssl
try:
    _create_unverified_https_context = ssl._create_unverified_context
except AttributeError:
    pass
else:
    ssl._create_default_https_context = _create_unverified_https_context

import pandas as pd
import numpy as np
import re
df = pd.read_csv('learn-ai-bbc/BBC News Train.csv')
df.head()

# ## Data preparation
# #### Remove url from text
```

```
df['Text'] = df['Text'].apply(lambda x: re.split('https://\.*', str(x))[0])
df.head()
```

```
# ### Remove emails and mobile numbers and emoji
```

```
df['Text'] = df['Text'].apply(lambda x: re.split('\S+@\S+', str(x)) [0])
```

```
df['Text'] = df['Text'].apply(lambda x: re.split('[+\(\)?\d[\d .-\(\)]{6,}',str(x)) [0])
```

```
emoji_pattern = re.compile("[
```

```
    u"\U0001F600-\U0001F64F" # emoticons
```

```
    u"\U0001F300-\U0001F5FF" # symbols & pictographs
```

```
    u"\U0001F680-\U0001F6FF" # transport & map symbols
```

```
    u"\U0001F1E0-\U0001F1FF" # flags (iOS)
```

```
    u"\U00002500-\U00002BEF" # chinese char
```

```
    u"\U00002702-\U000027B0"
```

```
    u"\U00002702-\U000027B0"
```

```
    u"\U000024C2-\U0001F251"
```

```
    u"\U0001f926-\U0001f937"
```

```
    u"\U00010000-\U0010ffff"
```

```
    u"\u2640-\u2642"
```

```
    u"\u2600-\u2B55"
```

```
    u"\u200d"
```

```
    u"\u23cf"
```

```
    u"\u23e9"
```

```
    u"\u231a"
```

```
    u"\ufe0f" # dingbats
```

```
    u"\u3030"
```

```
")+", flags=re.UNICODE)
```

```
df['Text'] = df['Text'].apply(lambda x: re.split(emoji_pattern,str(x)) [0])
```

```
# ### Remove punctuation
```

```
df['Text'] = df['Text'].str.replace('[^\w\s]','')
```

```
# ### Lemmatization
```

```
import nltk
```

```
from nltk.stem import WordNetLemmatizer
```

```
from nltk.tokenize import word_tokenize
```

```
df['text1'] = df['Text'].apply(lambda x: word_tokenize(x))
```

```
#nltk.download('wordnet') - DOWNLOAD
```

```
#nltk.download('omw-1.4') - DOWNLOAD
```

```
lemmatizer = WordNetLemmatizer()
```

```
df['text1'] = df['text1'].apply(lambda x:[lemmatizer.lemmatize(w) for w in x ])
```

```
df.head()
```

```
# ### Removing stop words
```

```
import nltk
```

```
from nltk.corpus import stopwords
```

```
stop_words = set(stopwords.words('english'))
```

```
add = {'said','wa','ha','mr','mrs'}
```

```
stop_words.update(add)
```

```
df['text1'] = df['text1'].apply(lambda x: [w.lower() for w in x if not w.lower() in stop_words])
```

```
# #### Non English words removal
```

```
#nltk.download('words')
```

```
words = set(nltk.corpus.words.words())
```

```
df['text1'] = df['text1'].apply(lambda x: [w for w in x if w.lower() in words or w.isalpha()])
```

```
df.head(20)
```

```
# ## EDA
```

```
print(df.info())
```

```
print(df.describe())
```

```
import matplotlib.pyplot as plt
```

```
import seaborn as sns
```

```
sns.countplot(x='Category', data=df)
```

```
plt.title('Distribution of Categories')
```

```
plt.show()
```

```
df['Text_Length'] = df['Text'].apply(len)
```

```
sns.histplot(data=df, x='Text_Length', bins=20)
```

```
plt.title('Distribution of Text Length')
```

```
plt.show()
```

```
all_words = [word for sublist in df['text1'] for word in sublist]
word_freq = pd.Series(all_words).value_counts()
word_freq[:10].plot(kind='bar')
plt.title('Top 10 Most Frequent Words')
plt.xlabel('Words')
plt.ylabel('Frequency')
plt.show()

# Get unique categories
categories = df['Category'].unique()

# Set the figure size

# Iterate over each category
for category in categories:
    # Filter data for the current category
    category_data = df[df['Category'] == category]

    # Generate word frequency for the current category
    category_words = [word for sublist in category_data['text1'] for word in sublist]
    category_word_freq = pd.Series(category_words).value_counts()

    # Plot the top 10 most frequent words for the current category
    plt.figure(figsize=(3, 2)) # Adjust the width and height values as desired

    category_word_freq[:10].plot(kind='bar')
    plt.title(f'Top 10 Most Frequent Words in {category.capitalize()} Category')
    plt.xlabel('Words')
    plt.ylabel('Frequency')
```

```
plt.show()

## DecisionTree

import time
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Convert list of tokens to strings
df['text1_strings'] = df['text1'].apply(' '.join)

# Use TF-IDF Vectorization
vectorizer = TfidfVectorizer(max_features=5000)
X = vectorizer.fit_transform(df['text1_strings'])

# Convert categories into numerical labels
le = LabelEncoder()
y = le.fit_transform(df['Category'])

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=17)

# Train the Decision Tree model
start_time = time.time()
dt_model = DecisionTreeClassifier()
dt_model.fit(X_train, y_train)
```

```
end_time = time.time()

# Print time taken to train model
print(f'Time taken to train Decision Tree model: {end_time - start_time} seconds')

# Predict labels for the test set
y_pred = dt_model.predict(X_test)

# Print accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Decision Tree Accuracy: {accuracy*100}')
```

## TF-IDF

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, Conv1D, MaxPooling1D, Flatten, Embedding
from keras.utils import pad_sequences
from keras.utils import to_categorical
import numpy as np

# Use TF-IDF Vectorization
vectorizer = TfidfVectorizer(max_features=5000)
df['text1_strings'] = df['text1'].apply(lambda tokens: ' '.join(tokens))

# Then fit and transform with vectorizer
X = vectorizer.fit_transform(df['text1_strings'].values.astype('U'))
#X = vectorizer.fit_transform(df['text1'].values.astype('U'))
```

```
# Convert categories into numerical labels
from sklearn import preprocessing
le = preprocessing.LabelEncoder()
y = le.fit_transform(df['Category'])

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=17)

# Convert labels to categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Define the model
model = Sequential()
model.add(Dense(128, input_shape=(5000,), activation='relu'))
model.add(Dense(len(np.unique(y)), activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

# Train the model
start_time = time.time()
model.fit(X_train.toarray(), y_train, epochs=10, batch_size=64)
end_time = time.time()

# Evaluate the model
accuracy = model.evaluate(X_test.toarray(), y_test)[1]
```

```
print(f'Time taken to train model: {end_time - start_time} seconds')

print(f'TF-IDF Accuracy: {accuracy*100}')

## BOW

from sklearn.feature_extraction.text import CountVectorizer

# Use BoW Vectorization
vectorizer = CountVectorizer(max_features=5000)
df['text1_strings'] = df['text1'].apply(lambda tokens: ' '.join(tokens))

# Then fit and transform with vectorizer
X = vectorizer.fit_transform(df['text1_strings'].values.astype('U'))

# Use the same steps as in TF-IDF vectorization from here onwards
from sklearn import preprocessing
le = preprocessing.LabelEncoder()
y = le.fit_transform(df['Category'])

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=17)

# Convert labels to categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Define the model
model = Sequential()
```

```
model.add(Dense(128, input_shape=(5000,), activation='relu'))
model.add(Dense(len(np.unique(y)), activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

# Train the model
start_time = time.time()
model.fit(X_train.toarray(), y_train, epochs=10, batch_size=64)
end_time = time.time()

# Evaluate the model
accuracy = model.evaluate(X_test.toarray(), y_test)[1]

print(f'Time taken to train model: {end_time - start_time} seconds')
print(f'BoW Accuracy: {accuracy*100}')
```

## GloVE

```
from keras.preprocessing.text import Tokenizer

# Tokenize the text
tokenizer = Tokenizer()
tokenizer.fit_on_texts(df['text1'])
sequences = tokenizer.texts_to_sequences(df['text1'])

# Padding sequences
X = pad_sequences(sequences, maxlen=5000)
```

```
# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Convert labels to categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Load GloVe vectors
embeddings_index = {}
with open('/Users/valeriiminuk/Downloads/glove/glove.6B.100d.txt') as f:
    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs

# Create a weight matrix
embedding_matrix = np.zeros((len(tokenizer.word_index) + 1, 100))
for word, i in tokenizer.word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector

# Define the model
model = Sequential()
model.add(Embedding(len(tokenizer.word_index) + 1, 100,
weights=[embedding_matrix], input_length=5000, trainable=False))
model.add(Conv1D(128, 5, activation='relu'))
model.add(MaxPooling1D(5))
```

```
model.add(Flatten())
model.add(Dense(len(np.unique(y)), activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

# Train the model
start_time = time.time()
model.fit(X_train, y_train, epochs=10, batch_size=64)
end_time = time.time()

# Evaluate the model
accuracy = model.evaluate(X_test, y_test)[1]
print(f'Time taken to train model: {end_time - start_time} seconds')
print(f'GloVe Accuracy: {accuracy*100}')
```