Sushant University

SCHOOL OF ENGINEERING AND TECHNOLOGY

NATURAL LANGUAGE PROCESSING

Course Code: 25BCS-0NL42C

Submitted By: Divya Sain (210BTCSEAM039)

Submitted To: Mr. Kartikeya Srivastava

INDEX

S. No	Name of the experiment	Date of performance	Page No	Signature	Remarks
1	To introduce fundamental concepts of NLP and implement basic text processing techniques	31-01-25	5		
2	To analyze sentiments in text data using NLP techniques	31-01-25	6		
3	To address common NLP challenges such as ambiguity and noise	05-02-25	7		
4	To scrape a webpage and preprocess the extracted text	05-02-25	8		
5	To convert raw text into numerical representations using vectorization and one-hot encoding techniques	12-02-25	9-10		
6	To convert text data into numerical features using BoW and n-gram models for text classification tasks	12-02-25	11-12		
7	To compute TF-IDF scores for text data and understand its impact on feature weighting	19-02-25	13-14		

8	To apply neural embeddings for text classification and compare results with traditional methods	19-02-25	15-17		
9	To extract named entities (persons, locations, organizations) and assign grammatical roles to words in a given text	05-03-25	18-19		
10	To identify and extract relationships (e.g., employee-company, person-location) between named entities in a sentence To	05-03-25	20-21		
11	design a chatbot that can understand user queries and respond appropriately	12-03-25	22-25		
Value Added Programs					
12	To classify chatbots into different types and create a basic chatbot using predefined rules	19-03-25	26-26		
13	To analyze Twitter sentiments using Natural Language Processing techniques	26-03-25	28-29		
14	To perform sentiment analysis on product reviews and determine user satisfaction	02-04-25	30-31		

15	To improve product search using NLP-based techniques	09-04-25	32-33		
16	To build an NLP-based recommendation system for e-commerce platforms	23-04-25	34-35		

Experiment No. 1

Aim: To introduce fundamental concepts of NLP and implement basic text processing techniques.

Apparatus Required:

- Python
- NLTK
- spaCy

Theory:

Natural Language Processing (NLP) enables computers to interpret, understand, and generate human language. It is used in applications such as chatbots, sentiment analysis, and machine translation. Tokenization splits text into words or sentences, while stemming and lemmatization reduce words to their base forms.

Code Steps:

1. Install required libraries (pip install nltk spacy)
2. Import necessary modules (import nltk, spacy)
3. Tokenize text using NLTK (nltk.word_tokenize("Hello world!"))
4. Perform stemming and lemmatization (nltk.stem.PorterStemmer())
5. Apply part-of-speech tagging (nltk.pos_tag(tokens))

```python
import nltk
import spacy
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer
from nltk import pos_tag

nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')

spacy.cli.download("en_core_web_sm")
nlp = spacy.load("en_core_web_sm")

text = "Hello world! Natural Language Processing is fascinating."

tokens = word_tokenize(text)
print("Tokens:", tokens)

stemmer = PorterStemmer()
stemmed = [stemmer.stem(word) for word in tokens]
print("Stemmed:", stemmed)

lemmatizer = WordNetLemmatizer()
lemmatized = [lemmatizer.lemmatize(word) for word in tokens]
print("Lemmatized:", lemmatized)

pos_tags = pos_tag(tokens)
print("POS Tags:", pos_tags)
```

Experiment No. 2

Aim: To analyze sentiments in text data using NLP techniques.

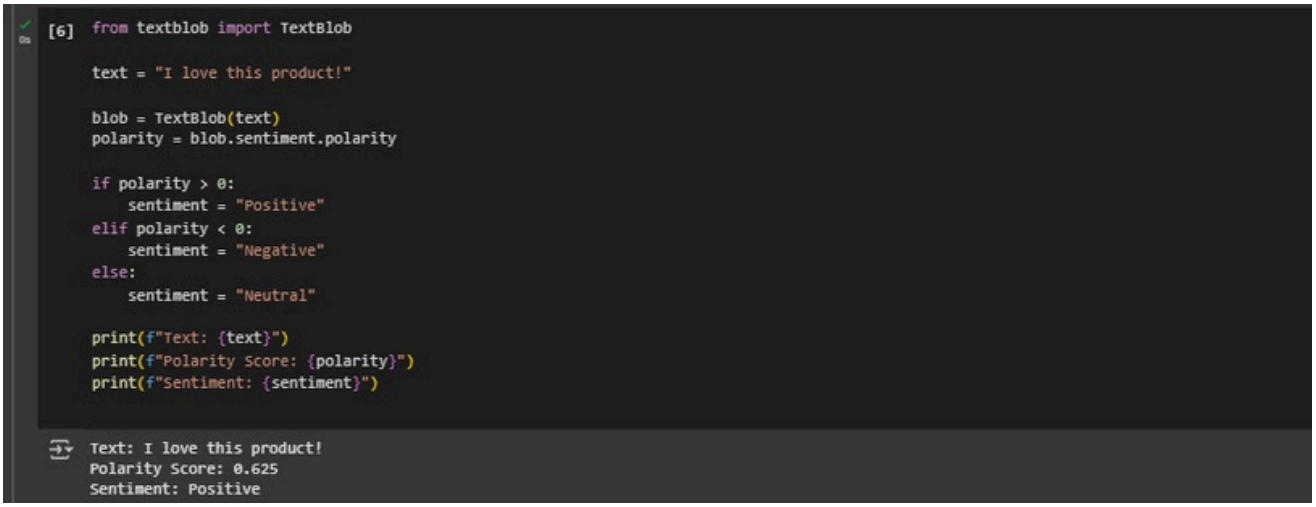
Apparatus Required:

- Python
- NLTK
- TextBlob

Theory: NLP is used in various real-world applications such as chatbots, recommendation systems, and social media analysis. Sentiment analysis is a crucial application where text is classified into positive, negative, or neutral sentiments. This is useful for businesses to understand customer feedback.

Code Steps:

1. Install dependencies (pip install textblob)
2. Import TextBlob (from textblob import TextBlob)
3. Input text (text = "I love this product!")
4. Analyze sentiment (TextBlob(text).sentiment.polarity)
5. Interpret results (positive, neutral, or negative sentiment)



```
[6] from textblob import TextBlob

    text = "I love this product!"

    blob = TextBlob(text)
    polarity = blob.sentiment.polarity

    if polarity > 0:
        sentiment = "Positive"
    elif polarity < 0:
        sentiment = "Negative"
    else:
        sentiment = "Neutral"

    print(f"Text: {text}")
    print(f"Polarity Score: {polarity}")
    print(f"Sentiment: {sentiment}")

    Text: I love this product!
    Polarity Score: 0.625
    Sentiment: Positive
```

Experiment No. 3

Aim: To address common NLP challenges such as ambiguity and noise.
Apparatus Required:

- Python
- NLTK

Theory: NLP faces challenges like polysemy (words with multiple meanings), syntactic ambiguity, and noisy data. Stopwords are common words (e.g., "is," "the") that provide little value in text analysis.

Code Steps:

1. Install NLTK: pip install nltk
2. Import libraries: import nltk
3. Download stopwords: nltk.download('stopwords')
4. Load stopwords: from nltk.corpus import stopwords
5. Remove stopwords from text: words = [w for w in text.split() if w.lower() not in stopwords.words('english')]
6. Handle ambiguity using context: Implement word sense disambiguation with nltk.wsd

```
[33] # Step 2: Import libraries
     import nltk
     nltk.download('punkt')
     nltk.download('stopwords')
     nltk.download('wordnet')
     nltk.download('omw-1.4')
     nltk.download('punkt_tab')

     [nltk_data] Downloading package punkt to /root/nltk_data...
     [nltk_data]   Package punkt is already up-to-date!
     [nltk_data] Downloading package stopwords to /root/nltk_data...
     [nltk_data]   Package stopwords is already up-to-date!
     [nltk_data] Downloading package wordnet to /root/nltk_data...
     [nltk_data]   Package wordnet is already up-to-date!
     [nltk_data] Downloading package omw-1.4 to /root/nltk_data...
     [nltk_data]   Package omw-1.4 is already up-to-date!
     [nltk_data] Downloading package punkt_tab to /root/nltk_data...
     [nltk_data]   Package punkt_tab is already up-to-date!
     True

[34] from nltk.corpus import stopwords
     from nltk.wsd import lesk
     from nltk.tokenize import word_tokenize
     from nltk.corpus import wordnet as wn

     text = "The bank will not accept deposits after closing hours."

     words = [w for w in text.split() if w.lower() not in stopwords.words('english')]
     print("Words without stopwords:", words)

     context = word_tokenize(text)
     sense = lesk(context, 'bank', 'n')  # 'n' for noun

     print("\nWord Sense Disambiguation for 'bank':")
     print("Sense:", sense)
     print("Definition:", sense.definition() if sense else "No definition found")

     Words without stopwords: ['bank', 'accept', 'deposits', 'closing', 'hours.']

     Word Sense Disambiguation for 'bank':
     Sense: Synset('depository_financial_institution.n.01')
     Definition: a financial institution that accepts deposits and channels the money into lending activities
```

Experiment No. 4

Aim: To scrape a webpage and preprocess the extracted text.
Apparatus Required:

- Python
- BeautifulSoup
- requests

Theory: Web scraping collects text data from websites for NLP tasks. It is widely used for news aggregation, social media analysis, and research.

Code Steps:

1. Install libraries: pip install requests beautifulsoup4
2. Import libraries: from bs4 import BeautifulSoup
3. Fetch webpage: response = requests.get('https://example.com')
4. Parse content: soup = BeautifulSoup(response.text, 'html.parser')
5. Extract text: text = soup.get_text()
6. Preprocess text: tokens = nltk.word_tokenize(text.lower())

```
0s
import requests
from bs4 import BeautifulSoup
import nltk


url = 'https://docs.djangoproject.com/en/5.2/'
response = requests.get(url)

soup = BeautifulSoup(response.text, 'html.parser')

text = soup.get_text()

tokens = nltk.word_tokenize(text.lower())
print("First 50 tokens:", tokens[:50])
```

OUTPUT

```
First 50 tokens: ['django', 'documentation', '|', 'django'
'documentation', '|', 'django', 'django', 'the', 'web', 'framework',
'for', 'perfectionists', 'with', 'deadlines', '.', 'toggle', 'theme',
'(', 'current', 'theme', ':', 'auto', ')', 'toggle', 'theme', '(',
'current', 'theme', ':', 'light', ')', 'toggle', 'theme', '(',
'current', 'theme', ':', 'dark', ')', 'toggle', 'light', '/', 'dark',
'/', 'auto', 'color', 'theme', 'menu', 'main']
```

Experiment N0. 5

Aim: To convert raw text into numerical representations using vectorization and one-hot encoding techniques.

Apparatus Required:

- Python
- Jupyter Notebook
- Numpy, Pandas
- Scikit-learn
- TensorFlow/ Keras

Theory:
Text data must be transformed into a machine-readable format for NLP tasks. One-hot encoding and basic vectorization are fundamental techniques used for this purpose. One-hot encoding represents words as binary vectors, where only one element is '1' while others remain '0'. However, this method results in sparse representations. Alternatively, vectorization converts words into numerical formats based on frequency or occurrence patterns, making them useful for machine learning models. These techniques help preprocess textual data for tasks like classification, sentiment analysis, and machine translation.

Code Steps:

1. Import necessary libraries (numpy, pandas, sklearn, etc.).
2. Define a sample text corpus.
3. Apply one-hot encoding using TensorFlow/Keras or Scikit-learn.
4. Convert text into numerical vectors using basic vectorization.
5. Display results and analyze the vector representations.

```python
import numpy as np
import pandas as pd
from sklearn.preprocessing import OneHotEncoder
from sklearn.feature_extraction.text import CountVectorizer
from tensorflow.keras.preprocessing.text import one_hot
from tensorflow.keras.preprocessing.text import Tokenizer

corpus = [
    "The cat sat on the mat",
    "The dog ate my homework",
    "My cat and dog are friends"
]

vocab_size = 50
encoded_docs = [one_hot(doc, vocab_size) for doc in corpus]
print("One-Hot Encoded (Keras word indices):")
print(encoded_docs)

vectorizer = CountVectorizer(binary=True)
X_binary = vectorizer.fit_transform(corpus).toarray()
print("\nOne-Hot Encoded (Scikit-learn binary matrix):")
print(pd.DataFrame(X_binary, columns=vectorizer.get_feature_names_out()))

vectorizer_freq = CountVectorizer()
X_freq = vectorizer_freq.fit_transform(corpus).toarray()
print("\nCount Vectorized Representation:")
print(pd.DataFrame(X_freq, columns=vectorizer_freq.get_feature_names_out()))

print("\nVocabulary:", vectorizer_freq.get_feature_names_out())
print("Shape of vectorized data:", X_freq.shape)
```

```
One-Hot Encoded (Keras word indices):
[[15, 22, 32, 22, 15, 43], [15, 42, 29, 44, 12], [44, 22, 41, 42, 10, 31]]

One-Hot Encoded (Scikit-learn binary matrix):
   and  are  ate  cat  dog  friends  homework  mat  my  on  sat  the
0    0    0    0    1    0        0         0    1   0   1    1    1
1    0    0    1    0    1        0         1    0   1   0    0    1
2    1    1    0    1    1        1         0    0   1   0    0    0

Count Vectorized Representation:
   and  are  ate  cat  dog  friends  homework  mat  my  on  sat  the
0    0    0    0    1    0        0         0    1   0   1    1    2
1    0    0    1    0    1        0         1    0   1   0    0    1
2    1    1    0    1    1        1         0    0   1   0    0    0

Vocabulary: ['and' 'are' 'ate' 'cat' 'dog' 'friends' 'homework' 'mat' 'my' 'on' 'sat'
 'the']
Shape of vectorized data: (3, 12)
```

Experiment No. 6

Aim: To convert text data into numerical features using BoW and n-gram models for text classification tasks.

Apparatus Required:

- Python
- Numpy, Pandas
- Scikit-learn

Theory:

The Bag of Words (BoW) model represents text as a collection of words without considering word order. It converts text into a numerical feature matrix based on word frequency. However, BoW ignores context, which is addressed by the N-Gram model. The N-Gram approach considers word sequences (e.g., bi-grams, tri-grams), capturing more linguistic structure. These techniques are widely used in spam detection, sentiment analysis, and text classification tasks. While BoW is computationally efficient, N-Gram models provide better context representation.

Code Steps:

1. Load and preprocess a text dataset.
2. Tokenize and clean the text.
3. Apply the BoW technique using Scikit-learn's CountVectorizer.
4. Implement N-Gram models with different ranges (bi-grams, tri-grams).
5. Display and analyze the transformed feature vectors.

```python
import pandas as pd
import re
from sklearn.feature_extraction.text import CountVectorizer

corpus = [
    "The quick brown fox jumps over the lazy dog.",
    "Never jump over the lazy dog quickly.",
    "A fast brown fox leaped over the sleepy dog."
]

def preprocess_text(text):
    text = text.lower()
    text = re.sub(r'[^a-z\s]', '', text)
    return text

cleaned_corpus = [preprocess_text(doc) for doc in corpus]

vectorizer_unigram = CountVectorizer()
X_unigram = vectorizer_unigram.fit_transform(cleaned_corpus).toarray()
print("=== Unigram BoW ===")
print(pd.DataFrame(X_unigram, columns=vectorizer_unigram.get_feature_names_out()))

vectorizer_bigram = CountVectorizer(ngram_range=(2, 2))
X_bigram = vectorizer_bigram.fit_transform(cleaned_corpus).toarray()
print("\n=== Bigram BoW ===")
print(pd.DataFrame(X_bigram, columns=vectorizer_bigram.get_feature_names_out()))

vectorizer_trigram = CountVectorizer(ngram_range=(3, 3))
X_trigram = vectorizer_trigram.fit_transform(cleaned_corpus).toarray()
print("\n=== Trigram BoW ===")
print(pd.DataFrame(X_trigram, columns=vectorizer_trigram.get_feature_names_out()))
```

```
=== Unigram BoW ===
   brown  dog  fast  fox  jump  jumps  lazy  leaped  never  over  quick  \
0      1    1     0    1     0      1     1       0      0     1      1
1      0    1     0    0     1      0     1       0      1     1      0
2      1    1     1    1     0      0     0       1      0     1      0

   quickly  sleepy  the
0        0       0    2
1        1       0    1
2        0       1    1

=== Bigram BoW ===
   brown fox  dog quickly  fast brown  fox jumps  fox leaped  jump over  \
0          1           0           0          1           0          0
1          0           1           0          0           0          1
2          1           0           1          0           1          0

   jumps over  lazy dog  leaped over  never jump  over the  quick brown  \
0           1         1            0           0         1            1
1           0         1            0           1         1            0
2           0         0            1           0         1            0

   sleepy dog  the lazy  the quick  the sleepy
0           0         1          1           0
1           0         1          0           0
2           1         0          0           1

=== Trigram BoW ===
   brown fox jumps  brown fox leaped  fast brown fox  fox jumps over  \
0                1                 0               0               1
1                0                 0               0               0
2                0                 1               1               0

   fox leaped over  jump over the  jumps over the  lazy dog quickly  \
0                0              0               1                 0
1                0              1               0                 1
2                1              0               0                 0

   leaped over the  never jump over  over the lazy  over the sleepy  \
0                0                 0              1                0
1                0                 1              1                0
2                1                 0              0                1

   quick brown fox  the lazy dog  the quick brown  the sleepy dog
0                1              1                1               0
1                0              1                0               0
2                0              0                0               1
```

Experiment No. 7

Aim: To compute TF-IDF scores for text data and understand its impact on feature weighting.

Apparatus Required:

- Python
- Numpy, Pandas
- Scikit-learn

Theory:

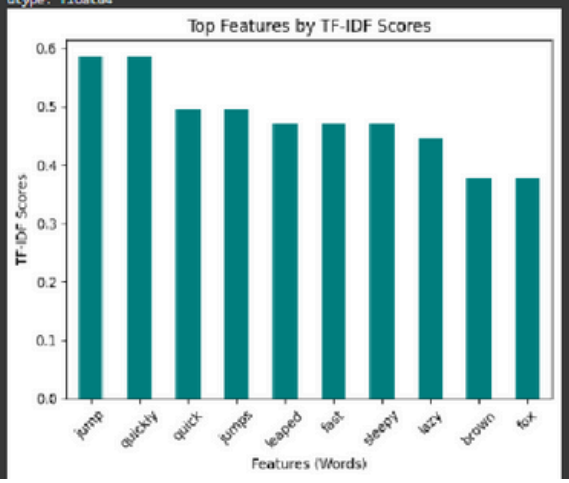
TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure that reflects word importance in a document relative to a collection of documents (corpus). It addresses the limitations of BoW by reducing the influence of frequently occurring words like “the” and “is.” TF calculates word occurrence within a document, while IDF reduces the weight of common words by penalizing them based on their frequency across documents. This technique is widely used in search engines, document ranking, and text classification. Compared to BoW, TF-IDF produces more meaningful numerical representations by giving less importance to generic words and highlighting significant terms.

Code Steps:

1. Load a sample text dataset.
2. Tokenize and preprocess the text (lowercasing, removing stopwords).
3. Apply Scikit-learn’s TfidfVectorizer.
4. Compute TF-IDF scores and compare them with BoW results.
5. Display feature importance and analyze the results.

```
[48] import pandas as pd
     import re
     from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
     from sklearn.feature_extraction import text
     import matplotlib.pyplot as plt

     corpus = [
         "The quick brown fox jumps over the lazy dog.",
         "Never jump over the lazy dog quickly.",
         "A fast brown fox leaped over the sleepy dog."
     ]

     def preprocess(text_str):
         text_str = text_str.lower()
         text_str = re.sub(r'[^a-z\s]', '', text_str)
         return text_str

     stop_words = text.ENGLISH_STOP_WORDS
     cleaned_corpus = [' '.join([word for word in preprocess(doc).split() if word not in stop_words]) for doc in corpus]

     bow_vectorizer = CountVectorizer()
     bow_matrix = bow_vectorizer.fit_transform(cleaned_corpus).toarray()
     bow_df = pd.DataFrame(bow_matrix, columns=bow_vectorizer.get_feature_names_out())
     print("=== Bag of Words (BoW) ===")
     print(bow_df)

     tfidf_vectorizer = TfidfVectorizer()
     tfidf_matrix = tfidf_vectorizer.fit_transform(cleaned_corpus).toarray()
     tfidf_df = pd.DataFrame(tfidf_matrix, columns=tfidf_vectorizer.get_feature_names_out())
     print("\n=== TF-IDF ===")
     print(tfidf_df)

     print("\n=== Top TF-IDF Scores by Feature ===")
     feature_scores = tfidf_df.max().sort_values(ascending=False)
     print(feature_scores)

     top_n = 10
     feature_scores[:top_n].plot(kind='bar', color='teal')
     plt.title('Top Features by TF-IDF Scores')
     plt.xlabel('Features (Words)')
     plt.ylabel('TF-IDF Scores')
     plt.xticks(rotation=45)
     plt.show()
```

```
   brown  dog  fast  fox  jump  jumps  lazy  leaped  quick  quickly  sleepy
0      1    1     0    1     0      1     1       0      1        0       0
1      0    1     0    0     1      0     1       0      0        1       0
2      1    1     1    1     0      0     0       1      0        0       1

=== TF-IDF ===
      brown       dog      fast       fox      jump     jumps      lazy  \
0  0.376331  0.292254  0.000000  0.376331  0.000000  0.49483  0.376331
1  0.000000  0.345205  0.000000  0.000000  0.584483  0.00000  0.444514
2  0.358291  0.278245  0.47111   0.358291  0.000000  0.00000  0.000000

     leaped     quick   quickly   sleepy
0  0.00000  0.49483  0.000000  0.00000
1  0.00000  0.00000  0.584483  0.00000
2  0.47111  0.00000  0.000000  0.47111

=== Top TF-IDF Scores by Feature ===
jump       0.584483
quickly    0.584483
quick      0.494830
jumps      0.494830
leaped     0.471110
fast       0.471110
sleepy     0.471110
lazy       0.444514
brown      0.376331
fox        0.376331
dog        0.345205
dtype: float64
```



Experiment No. 8

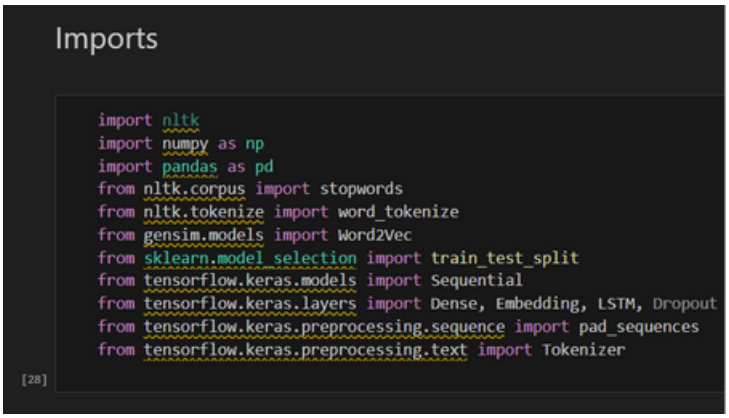
Aim: To apply neural embeddings for text classification and compare results with traditional methods.

Apparatus Required:

- Python
- TensorFlow/Keras
- Gensim
- Scikit-learn

Theory:
Traditional vectorization techniques like BoW and TF-IDF fail to capture semantic relationships between words. Neural embeddings, such as Word2Vec, GloVe, and FastText, map words into dense vector spaces where similar words have closer representations. These embeddings are learned from large corpora and help models understand word meanings better. Word2Vec uses Skip-gram and CBOW techniques, while GloVe captures global word co-occurrences. Such embeddings significantly improve deep learning models for tasks like sentiment analysis, spam detection, and document classification.
Code Steps:

1. Load a labeled text dataset.
2. Preprocess text (cleaning, tokenization, stopword removal).
3. Train or load pre-trained word embeddings (Word2Vec, GloVe).
4. Build a neural network model using embeddings as input.
5. Train and evaluate the text classification model.



Raw data

```
data = {
    'text': [
        "I love this movie",
        "This film was terrible",
        "An excellent performance",
        "Horrible acting and plot",
        "What a great story!",
        "Worst movie ever"
    ],
    'label': [1, 0, 1, 0, 1, 0]
}
df = pd.DataFrame(data)
```
[11]

Preprocessing

```
stop_words = set(stopwords.words('english'))
```
[12]

```
def preprocess(text):
    tokens = word_tokenize(text.lower())
    return [w for w in tokens if w.isalpha() and w not in stop_words]
```
[13]

```
df['tokens'] = df['text'].apply(preprocess)
```
[16]

Train Word2Vec

```
w2v_model = Word2Vec(sentences=df['tokens'], vector_size=100, window=5, min_count=1)
```
[19]

Tokenizer + Sequences

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(df['tokens'])
sequences = tokenizer.texts_to_sequences(df['tokens'])
word_index = tokenizer.word_index
```
[22]

```
max_len = max(len(seq) for seq in sequences)
X = pad_sequences(sequences, maxlen = max_len)
y = np.array(df['label'])
```
[23]

Create embedding matrix

```
embedding_dim = 100
embedding_matrix = np.zeros((len(word_index) + 1, embedding_dim))
for word, i in word_index.items():
    if word in w2v_model.wv:
        embedding_matrix[i] = w2v_model.wv[word]
```
[24]

Train-test split

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=42)
```
[25]

Modelling

```
model = Sequential()
model.add(Embedding(input_dim=len(word_index)+1, output_dim=embedding_dim,
                    weights=[embedding_matrix], input_length=max_len, trainable=False))
model.add(LSTM(64, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

[29]

```
c:\Users\risha\Desktop\NLP-lab\venv\Lib\site-packages\keras\src\layers\core\embedding.py:90: UserWarning: Argument `input_length` is deprecated
  warnings.warn(
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	?	1,400
lstm_1 (LSTM)	?	0 (unbuilt)
dense_1 (Dense)	?	0 (unbuilt)

Total params: 1,400 (5.47 KB)

Trainable params: 0 (0.00 B)

```
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test))
```

[30]

```
Epoch 1/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 4s 4s/step - accuracy: 0.5000 - loss: 0.6938 - val_accuracy: 0.5000 - val_loss: 0.6929
Epoch 2/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 105ms/step - accuracy: 0.5000 - loss: 0.6931 - val_accuracy: 1.0000 - val_loss: 0.6929
Epoch 3/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 97ms/step - accuracy: 1.0000 - loss: 0.6920 - val_accuracy: 0.5000 - val_loss: 0.6930
Epoch 4/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step - accuracy: 0.7500 - loss: 0.6926 - val_accuracy: 0.5000 - val_loss: 0.6931
Epoch 5/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 97ms/step - accuracy: 0.7500 - loss: 0.6917 - val_accuracy: 0.5000 - val_loss: 0.6932
Epoch 6/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 97ms/step - accuracy: 1.0000 - loss: 0.6917 - val_accuracy: 0.5000 - val_loss: 0.6933
Epoch 7/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 98ms/step - accuracy: 1.0000 - loss: 0.6915 - val_accuracy: 0.0000e+00 - val_loss: 0.6933
Epoch 8/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 91ms/step - accuracy: 1.0000 - loss: 0.6912 - val_accuracy: 0.5000 - val_loss: 0.6934
Epoch 9/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 107ms/step - accuracy: 1.0000 - loss: 0.6909 - val_accuracy: 0.5000 - val_loss: 0.6935
Epoch 10/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 91ms/step - accuracy: 1.0000 - loss: 0.6908 - val_accuracy: 0.5000 - val_loss: 0.6935
```

Evaluation

```
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.2f}")
```

[32]

Experiment No. 9

Aim: To extract named entities (persons, locations, organizations) and assign grammatical roles to words in a given text.

Apparatus Required:

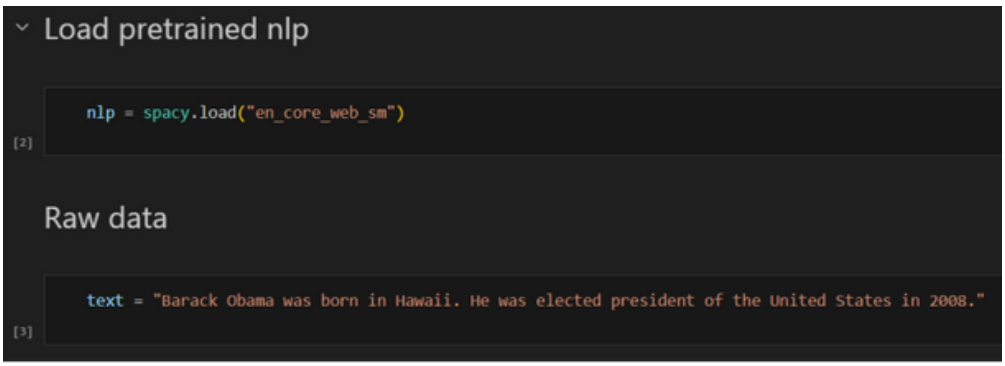
- Python
- spaCy or NLTK
- Jupyter Notebook

Theory:

Named Entity Recognition (NER) is a Natural Language Processing (NLP) technique that identifies proper names, dates, and numerical values in text. POS tagging assigns grammatical categories (noun, verb, adjective) to words. Libraries like spaCy and NLTK use pre-trained models for these tasks. NER helps in information retrieval, while POS tagging assists in syntactic analysis.

Code Steps:

1. Install and import necessary libraries. 2. Load a pre-trained NLP model (e.g., spaCy's "en_core_web_sm"). 3. Input a sample text. 4. Apply NER to extract named entities. 5. Apply POS tagging to classify words. 6. Display the results.



NER

```
doc = nlp(text)

for ent in  doc.ents:
    print(f"{ent.text} ({ent.label_}) - Start: {ent.start_char}, End: {ent.end_char}")
```

[4]

```
Barack Obama (PERSON) - Start: 0, End: 12
Hawaii (GPE) - Start: 25, End: 31
the United States (GPE) - Start: 61, End: 78
2008 (DATE) - Start: 82, End: 86
```

PoS

```
for token in doc:
    print(f"{token.text:<15} {token.pos_:<10} {token.dep_}")
```

[5]

```
Barack          PROPN      compound
Obama           PROPN      nsubjpass
was             AUX        auxpass
born            VERB       ROOT
in              ADP        prep
Hawaii          PROPN      pobj
.               PUNCT      punct
He              PRON       nsubjpass
was             AUX        auxpass
elected         VERB       ROOT
president       NOUN       oprd
of              ADP        prep
the             DET        det
United          PROPN      compound
States          PROPN      pobj
in              ADP        prep
2008            NUM        pobj
.               PUNCT      punct
```

Experiment No. 10

Aim: To identify and extract relationships (e.g., employee-company, person-location) between named entities in a sentence.

Apparatus Required:

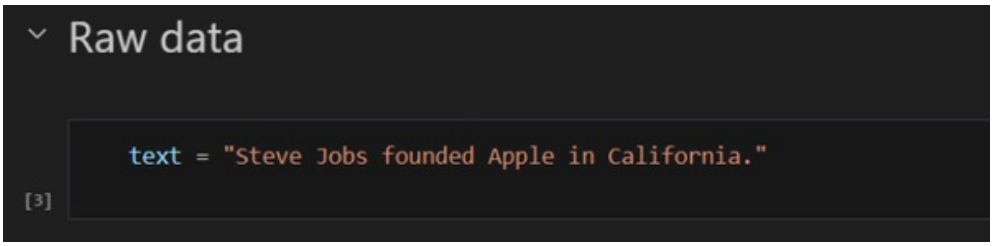
- Python
- spaCy
- NLTK
- OpenIE (Stanford NLP)

Theory:

Relationship extraction is a subtask of information extraction that detects semantic relationships between named entities. It uses NLP models to process textual data and derive meaningful connections. Dependency parsing and Open Information Extraction (OpenIE) techniques help identify relationships, which are useful in knowledge graph construction and data mining.

Code Steps:

1. Install and import required libraries.
2. Load a pre-trained NLP model.
3. Tokenize and preprocess the text.
4. Apply dependency parsing or OpenIE.
5. Extract relationships between named entities.
6. Print the extracted relationships.



Using Spacy

load spacy model

[2]
```
nlp = spacy.load("en_core_web_sm")
```

Data preprocessing

[4]
```
doc = nlp(text)
```

Extracting relationships

[5]
```
for token in doc:
    if token.dep_ in ("ROOT", "attr", "agent", "amod"):
        subject = [w for w in token.lefts if w.dep_ in ("nsubj", "nsubjpass")]
        object_ = [w for w in token.rights if w.dep_ in ("dobj", "pobj")]
        if subject and object_:
            print(f"Subject: {subject[0]}, Verb: {token}, Object: {object_[0]}")
```

```
Subject: Jobs, Verb: founded, Object: Apple
```

Experiment No. 11

Aim: To design a chatbot that can understand user queries and respond appropriately.

Apparatus Required:

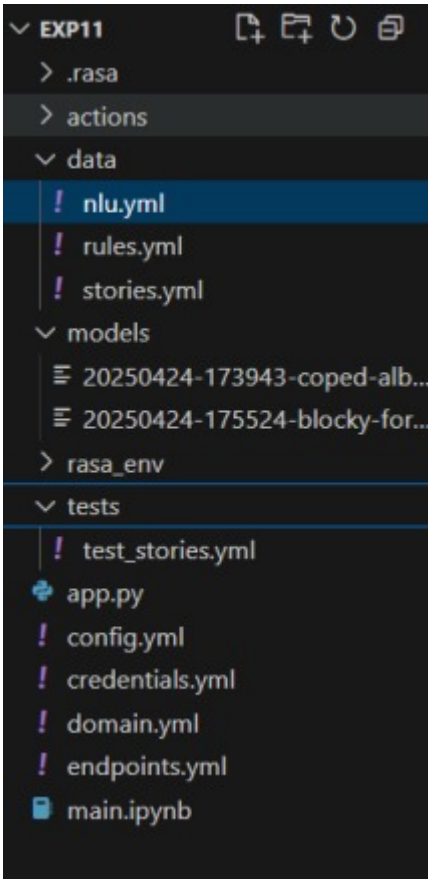
- Python
- Rasa NLU
- TensorFlow
- Jupyter Notebook

Theory:

Chatbots use NLP to interact with users in natural language. Rasa NLU is an open-source library for training chatbots using intent recognition and entity extraction. The chatbot pipeline includes training data preparation, intent classification, and dialogue management. Rasa's End-to-End approach automates the chatbot's response generation using machine learning.

Code Steps:

1. Install Rasa and necessary dependencies.
2. Create a Rasa project and define intents in training data.
3. Train the NLU model with example queries.
4. Implement dialogue management using Rasa stories.
5. Test the chatbot using sample user inputs.
6. Deploy the chatbot for interaction.



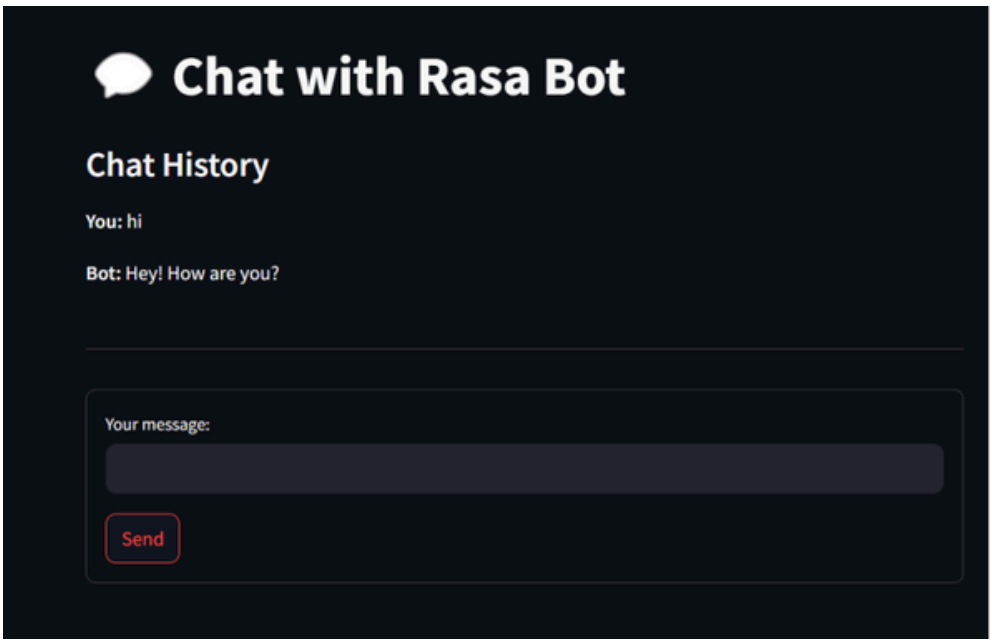

```yaml
- intent: ask_time
  examples: |
    - what time is it?
    - tell me the time
    - current time?
    - can you give me the time?
    - what's the time now?
    - do you know what time it is?

- intent: ask_date
  examples: |
    - what is today's date?
    - tell me the date
    - what's the date today?
    - give me the current date
    - do you know the date?

- intent: ask_weather
  examples: |
    - what's the weather like today?
    - is it raining?
    - will it be sunny tomorrow?
    - tell me the weather
    - what's the temperature outside?
    - do I need an umbrella today?

- intent: thanks
  examples: |
    - thank you
    - thanks a lot
    - appreciate it
    - thank you very much
    - thanks
    - many thanks
```

```yaml
utter_ask_time:
- text: "I'm not connected to the clock, but you can check your device"

utter_ask_date:
  - text: "I wish I had a calendar, but your device does!"

utter_ask_weather:
  - text: "I can't tell the weather right now, but you can ask Google or Alexa"

utter_thanks:
  - text: "You're welcome!"

utter_ask_name:
  - text: "I'm your friendly chatbot assistant."

utter_help:
  - text: "I can greet you, tell you the time, date, weather, and respond to your mood!"
```

```yaml
- story: asking for time
  steps:
  - intent: ask_time
  - action: utter_ask_time

- story: asking for date
  steps:
  - intent: ask_date
  - action: utter_ask_date
```

app.py > ...

```python
import streamlit as st
import requests

st.set_page_config(page_title="Rasa Chatbot", layout="centered")

st.title("💬 Chat with Rasa Bot")

# Store chat history
if "messages" not in st.session_state:
    st.session_state.messages = []

# Show chat history in a scrollable container
with st.container():
    st.markdown("### Chat History")
    for sender, message in st.session_state.messages:
        if sender == "user":
            st.markdown(f"**You:** {message}")
        else:
            st.markdown(f"**Bot:** {message}")

st.markdown("---")  # separator

# Input field at bottom
with st.form(key="chat_form", clear_on_submit=True):
    user_input = st.text_input("Your message:", "")
    submit = st.form_submit_button("Send")

# Handle user input
if submit and user_input:
    # Add user message
    st.session_state.messages.append(("user", user_input))

    # Send message to Rasa backend
    try:
        response = requests.post(
            "http://localhost:5005/webhooks/rest/webhook",
            json={"sender": "user", "message": user_input}
        )

        for msg in response.json():
            st.session_state.messages.append(("bot", msg.get("text", "")))
    except Exception as e:
        st.session_state.messages.append(("bot", f"⚠️ Error: {e}"))

```


Chat with Rasa Bot
Chat History
You: hi
Bot: Hey! How are you?
Your message:
Send

Experiment No. 12

Aim: To classify chatbots into different types and create a basic chatbot using predefined rules.

Software & Libraries Required:

- Python
- NLTK
- Flask (for deployment)

Theory:

Chatbots are categorized as retrieval-based (predefined responses) and generative (AI-driven). Rule-based chatbots use decision trees, while AI-based models use NLP and machine learning. Dialog systems process user queries through Natural Language Understanding (NLU) and generate responses using pre-defined rules or machine learning models.

Code Steps:

1. Define chatbot taxonomy and classify types.
2. Install required libraries.
3. Create a simple rule-based chatbot with predefined responses.
4. Implement basic NLP processing using NLTK.
5. Deploy the chatbot using Flask.
6. Test interactions with different queries.

```
[24] # Define a list of patterns and responses
     pairs = [
         (r'hi|hello|hey', ['Hello! How can I assist you today?', 'Hi! What can I do for you?']),
         (r'how are you?', ['I am fine, thank you! How can I help you?', 'I am doing well, how about you?']),
         (r'what is your name?', ['I am a chatbot, and I am here to help you!', 'I am your friendly assistant chatbot.']),
         (r'quit', ['Goodbye! Have a great day!', 'Take care!']),
         (r'(.*)', ['I am sorry, I did not understand that. Can you ask something else?'])
     ]

     # Create the chatbot using the pairs and reflections
     chatbot = Chat(pairs, reflections)
```

```
[25] # Sample NLP processing function
     def process_input(text):
         tokens = nltk.word_tokenize(text.lower())
         stopwords = nltk.corpus.stopwords.words('english')
         filtered_tokens = [word for word in tokens if word not in stopwords and word.isalpha()]
         return ' '.join(filtered_tokens)
```

```
# Initialize Flask app
app = Flask(__name__)

@app.route('/')
def home():
    return 'Chatbot is running! Go to /chat to interact with it.'

@app.route('/chat', methods=['POST'])
def chat():
    user_input = request.json.get('message')
    if user_input:
        # Process input using the NLP function
        processed_input = process_input(user_input)
        response = chatbot.respond(processed_input)
        return jsonify({'response': response})
    else:
        return jsonify({'response': 'Sorry, I did not get that.'})

if __name__ == '__main__':
    app.run(debug=True)
```

```
 * Serving Flask app '__main__'
 * Debug mode: on
INFO:werkzeug:WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
INFO:werkzeug:Press CTRL+C to quit
INFO:werkzeug: * Restarting with stat
```

Experiment No. 13

Aim: To analyze Twitter sentiments using Natural Language Processing techniques.

Apparatus Required:

- Python
- Jupyter Notebook
- tweepy, pandas, numpy, nltk, textblob, matplotlib

Theory:
Sentiment analysis helps in understanding public opinions on various topics. Twitter data, being unstructured, requires preprocessing before applying NLP techniques. Steps include data collection using Twitter API, cleaning tweets (removing stopwords, punctuations, and special characters), tokenization, and applying sentiment analysis models like TextBlob or VADER. The classified sentiments help businesses and organizations in decision-making.

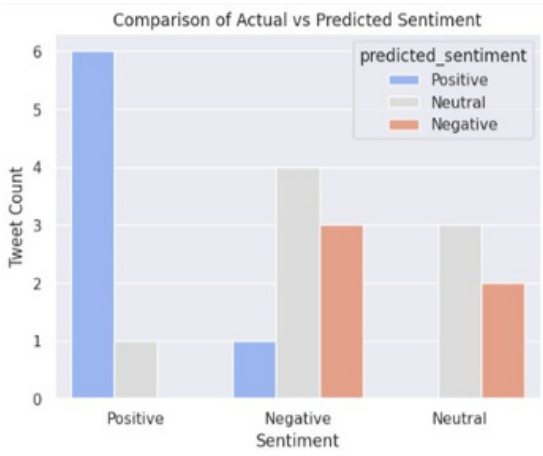
Code Steps:

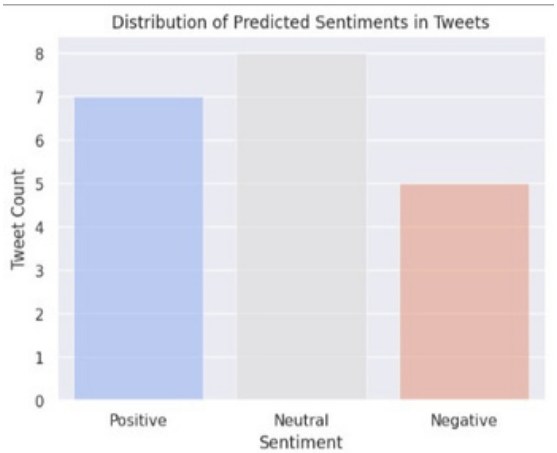
1. Authenticate with the Twitter API and fetch tweets.
2. Preprocess tweets by removing special characters, URLs, and stopwords.
3. Tokenize and normalize the text.
4. Apply TextBlob or VADER for sentiment classification.
5. Visualize results using Matplotlib or Seaborn.

```
import pandas as pd
import re
import nltk
import seaborn as sns
import matplotlib.pyplot as plt
from nltk.sentiment.vader import SentimentIntensityAnalyzer
data = {
    'tweet': [
        "I love artificial intelligence!", "AI is the future of technology.",
        "I hate how AI is being misused!", "AI is overrated.",
        "Artificial Intelligence will revolutionize industries.", "AI can be dangerous if not used responsibly.",
        "I feel neutral about AI.", "AI in healthcare is awesome!",
        "AI is taking over everything!", "I don't trust AI at all.",
        "AI can help solve many global challenges.", "I'm not sure about AI yet.",
        "I'm excited to see how AI will progress.", "AI doesn't live up to the hype.",
        "AI is fascinating but scary.", "I feel optimistic about AI.",
        "AI is going to change everything!", "I'm concerned about AI and privacy.",
        "AI is great, but it needs regulation.", "AI should be used with caution."
    ],
    'sentiment': [
        'Positive', 'Positive', 'Negative', 'Negative', 'Positive',
        'Negative', 'Neutral', 'Positive', 'Negative', 'Negative',
        'Positive', 'Neutral', 'Positive', 'Negative', 'Neutral',
        'Positive', 'Negative', 'Neutral', 'Negative', 'Neutral'
    ]
}
df = pd.DataFrame(data)
```

```
[22] def clean_tweet(text):
        text = re.sub(r'http\S+', '', text)  # Remove URLs
        text = re.sub(r'[^A-Za-z\s]', '', text)  # Remove special characters
        text = text.lower().strip()  # Normalize case and strip spaces
        return text

    df['cleaned_tweet'] = df['tweet'].apply(clean_tweet)

    nltk.download('vader_lexicon')
    sia = SentimentIntensityAnalyzer()

    sentiments = []
    for text in df['cleaned_tweet']:
        score = sia.polarity_scores(text)
        compound = score['compound']
        if compound >= 0.05:
            sentiments.append('Positive')
        elif compound <= -0.05:
            sentiments.append('Negative')
        else:
            sentiments.append('Neutral')

    df['predicted_sentiment'] = sentiments

    from sklearn.metrics import accuracy_score, classification_report

    accuracy = accuracy_score(df['sentiment'], df['predicted_sentiment'])
    print("\nModel Accuracy: ", accuracy)
    print("\nClassification Report:")
    print(classification_report(df['sentiment'], df['predicted_sentiment']))

    sns.set(style="darkgrid")
    plt.figure(figsize=(8, 6))

    sns.countplot(x='sentiment', data=df, palette='coolwarm', alpha=0.6, label="Actual Sentiment")
    plt.title("Distribution of Actual Sentiments in Tweets")
    plt.xlabel("Sentiment")
    plt.ylabel("Tweet Count")
    plt.show()

    sns.countplot(x='predicted_sentiment', data=df, palette='coolwarm', alpha=0.6, label="Predicted Sentiment")
    plt.title("Distribution of Predicted Sentiments in Tweets")
    plt.xlabel("Sentiment")
    plt.ylabel("Tweet Count")
    plt.show()

    sns.countplot(x="sentiment", hue="predicted_sentiment", data=df, palette='coolwarm')
    plt.title("Comparison of Actual vs Predicted Sentiment")
    plt.xlabel("Sentiment")
    plt.ylabel("Tweet Count")
    plt.show()
```

```
Model Accuracy:  0.6

Classification Report:
              precision    recall  f1-score   support

    Negative       0.60      0.38      0.46         8
     Neutral       0.38      0.60      0.46         5
    Positive       0.86      0.86      0.86         7

    accuracy                           0.60        20
   macro avg       0.61      0.61      0.59        20
weighted avg       0.63      0.60      0.60        20
```





Experiment No. 14

Aim: To perform sentiment analysis on product reviews and determine user satisfaction.

Apparatus Required:

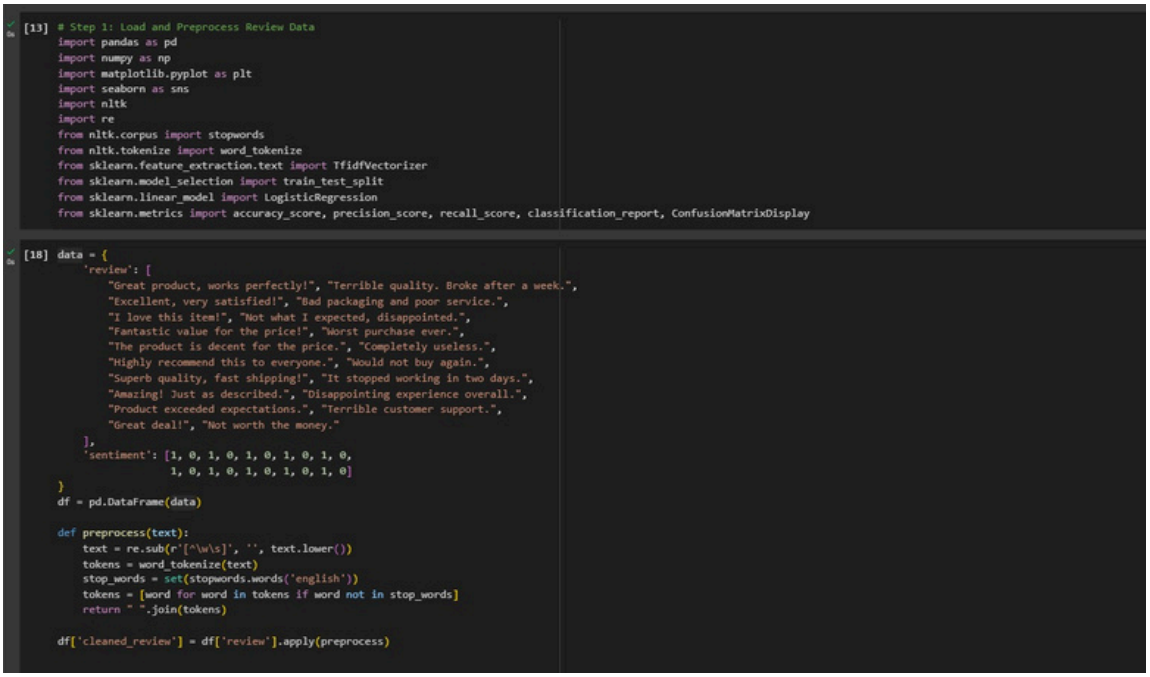
- Python
- Jupyter Notebook
- pandas, numpy, nltk, textblob, sklearn

Theory:

E-commerce platforms collect vast amounts of reviews that influence potential buyers. Sentiment analysis extracts useful insights by classifying reviews as positive, negative, or neutral. The text is preprocessed by removing noise, tokenizing, and converting it into numerical vectors using TF-IDF or Word2Vec. Machine learning models like Naïve Bayes or logistic regression can be used for classification.

Code Steps:

1. Load and preprocess review data.
2. Convert text into numerical vectors using TF-IDF or Word2Vec.
3. Train a machine learning model for classification.
4. Evaluate model performance using accuracy, precision, and recall.
5. Display results and visualize sentiment trends.

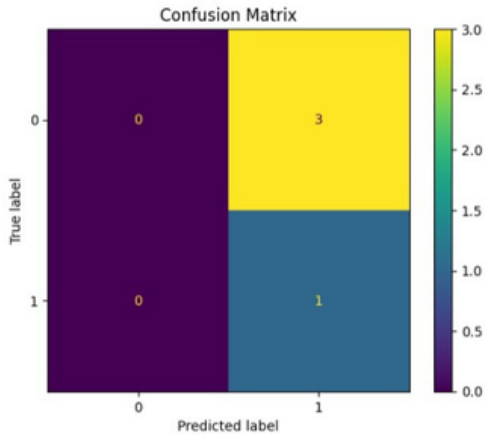


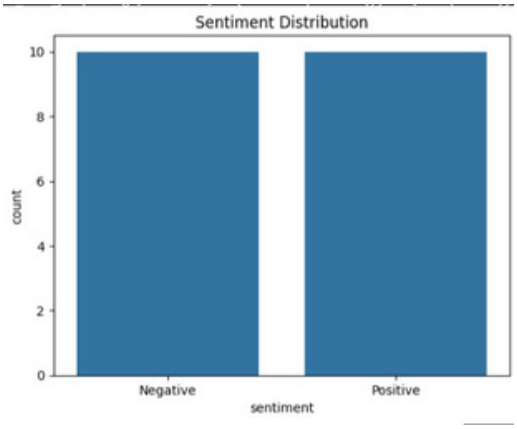
```
[13] # Step 1: Load and Preprocess Review Data
     import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
     import seaborn as sns
     import nltk
     import re
     from nltk.corpus import stopwords
     from nltk.tokenize import word_tokenize
     from sklearn.feature_extraction.text import TfidfVectorizer
     from sklearn.model_selection import train_test_split
     from sklearn.linear_model import LogisticRegression
     from sklearn.metrics import accuracy_score, precision_score, recall_score, classification_report, ConfusionMatrixDisplay
```

```
[18] data = {
         'review': [
             "Great product, works perfectly!", "Terrible quality. Broke after a week.",
             "Excellent, very satisfied!", "Bad packaging and poor service.",
             "I love this item!", "Not what I expected, disappointed.",
             "Fantastic value for the price!", "Worst purchase ever.",
             "The product is decent for the price.", "Completely useless.",
             "Highly recommend this to everyone.", "Would not buy again.",
             "Superb quality, fast shipping!", "It stopped working in two days.",
             "Amazing! Just as described.", "Disappointing experience overall.",
             "Product exceeded expectations.", "Terrible customer support.",
             "Great deal!", "Not worth the money."
         ],
         'sentiment': [1, 0, 1, 0, 1, 0, 1, 0, 1, 0,
                       1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
     }
     df = pd.DataFrame(data)

     def preprocess(text):
         text = re.sub(r'[^\w\s]', '', text.lower())
         tokens = word_tokenize(text)
         stop_words = set(stopwords.words('english'))
         tokens = [word for word in tokens if word not in stop_words]
         return " ".join(tokens)

     df['cleaned_review'] = df['review'].apply(preprocess)
```

```
[19] vectorizer = TfidfVectorizer()
     X = vectorizer.fit_transform(df['cleaned_review'])
     y = df['sentiment']

     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

     model = LogisticRegression()
     model.fit(X_train, y_train)

     y_pred = model.predict(X_test)

     accuracy = accuracy_score(y_test, y_pred)
     precision = precision_score(y_test, y_pred)
     recall = recall_score(y_test, y_pred)

     print("Classification Report:\n", classification_report(y_test, y_pred))
     print(f"Accuracy: {accuracy:.2f}")
     print(f"Precision: {precision:.2f}")
     print(f"Recall: {recall:.2f}")

     sns.countplot(x='sentiment', data=df)
     plt.title("Sentiment Distribution")
     plt.xticks([0, 1], ['Negative', 'Positive'])
     plt.show()

     disp = ConfusionMatrixDisplay.from_estimator(model, X_test, y_test)
     plt.title("Confusion Matrix")
     plt.show()
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         3
           1       0.25      1.00      0.40         1

    accuracy                           0.25         4
   macro avg       0.12      0.50      0.20         4
weighted avg       0.06      0.25      0.10         4

Accuracy: 0.25
Precision: 0.25
Recall: 1.00
```





Experiment No. 15

Aim: To improve product search using NLP-based techniques.

Apparatus Required:

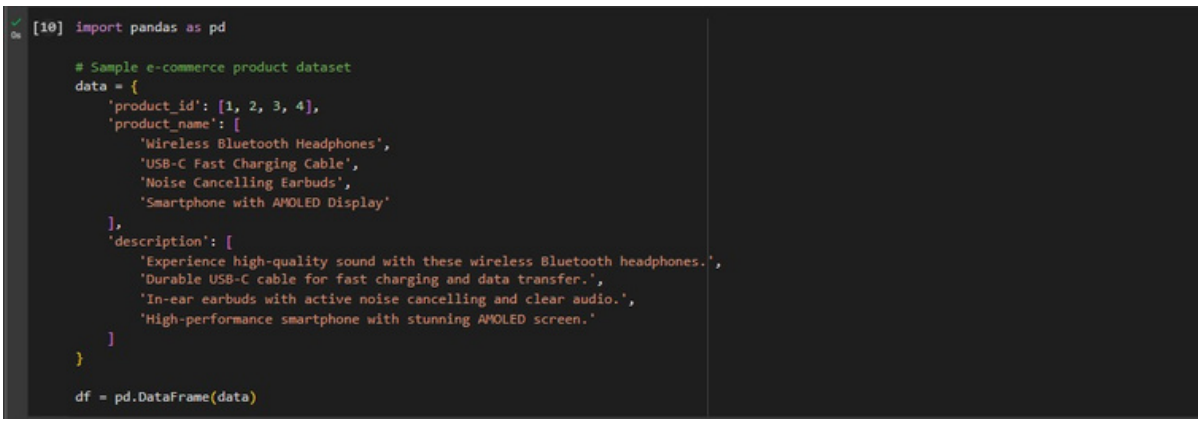
- Python
- Jupyter Notebook
- pandas, numpy, nltk, spacy, elasticsearch

Theory:

Traditional keyword-based searches in e-commerce may fail to understand user intent. NLP techniques, such as lemmatization, stemming, and named entity recognition (NER), help improve search results. Vectorization techniques like TF-IDF and word embeddings enhance relevance ranking. ElasticSearch further optimizes query handling and indexing.

Code Steps:

1. Load e-commerce product data.
2. Preprocess search queries by tokenizing and normalizing text.
3. Implement TF-IDF or word embeddings for semantic search.
4. Integrate with ElasticSearch for faster query execution.
5. Test the improved search accuracy against baseline models.



```
[10] import pandas as pd

    # Sample e-commerce product dataset
    data = {
        'product_id': [1, 2, 3, 4],
        'product_name': [
            'Wireless Bluetooth Headphones',
            'USB-C Fast Charging Cable',
            'Noise Cancelling Earbuds',
            'Smartphone with AMOLED Display'
        ],
        'description': [
            'Experience high-quality sound with these wireless Bluetooth headphones.',
            'Durable USB-C cable for fast charging and data transfer.',
            'In-ear earbuds with active noise cancelling and clear audio.',
            'High-performance smartphone with stunning AMOLED screen.'
        ]
    }

    df = pd.DataFrame(data)
```

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

nltk.download('punkt')
nltk.download('stopwords')

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

def preprocess(text):
    tokens = word_tokenize(text.lower())
    tokens = [word for word in tokens if word.isalnum() and word not in stop_words]
    return " ".join(tokens)

df['processed_description'] = df['description'].apply(preprocess)
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
[7] from sklearn.feature_extraction.text import TfidfVectorizer
    from sklearn.metrics.pairwise import cosine_similarity

    tfidf = TfidfVectorizer()
    tfidf_matrix = tfidf.fit_transform(df['processed_description'])
```

```
[8] def search(query, top_k=3):
        query_processed = preprocess(query)
        query_vec = tfidf.transform([query_processed])
        cosine_sim = cosine_similarity(query_vec, tfidf_matrix).flatten()
        top_indices = cosine_sim.argsort()[-top_k:][::-1]

        return df.iloc[top_indices][['product_name', 'description']]
```

```
[9] results = search("Bluetooth noise cancelling")
    print(results)
```

```
                    product_name  \
2       Noise Cancelling Earbuds
0  Wireless Bluetooth Headphones
3  Smartphone with AMOLED Display

                                         description
2  In-ear earbuds with active noise cancelling an...
0  Experience high-quality sound with these wirel...
3  High-performance smartphone with stunning AMOL...
```

Experiment No. 16

Aim: To build an NLP-based recommendation system for e-commerce platforms.

Apparatus Required:

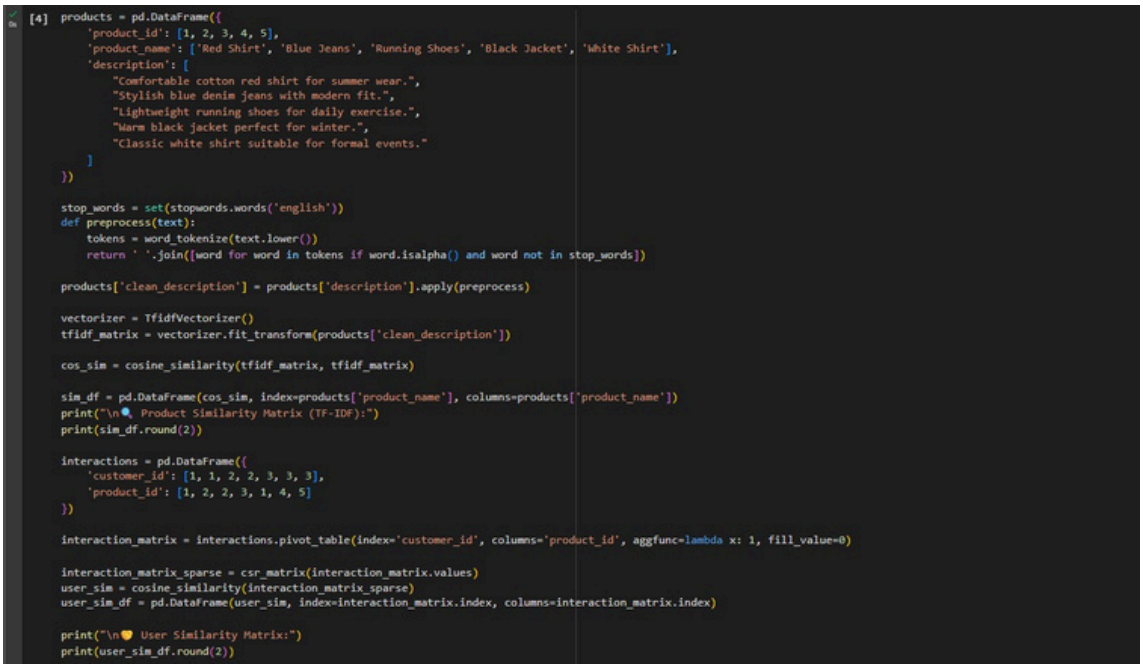
- Python
- Jupyter Notebook
- pandas, numpy, nltk, scikit-learn, gensim

Theory:

Recommender systems enhance user experience by suggesting relevant products. NLP extracts insights from user reviews, descriptions, and metadata. Two main approaches are content-based filtering (analyzing product text similarity) and collaborative filtering (based on user interactions). Word embeddings (Word2Vec, TF-IDF) help in finding product similarities.

Code Steps:

1. Collect and preprocess product data.
2. Use TF-IDF or Word2Vec to represent product descriptions.
3. Compute similarity scores for recommendations.
4. Implement collaborative filtering using customer interactions.
5. Generate recommendations and visualize results.



```
[4] products = pd.DataFrame({
        'product_id': [1, 2, 3, 4, 5],
        'product_name': ['Red Shirt', 'Blue Jeans', 'Running Shoes', 'Black Jacket', 'White Shirt'],
        'description': [
            "Comfortable cotton red shirt for summer wear.",
            "Stylish blue denim jeans with modern fit.",
            "Lightweight running shoes for daily exercise.",
            "Warm black jacket perfect for winter.",
            "Classic white shirt suitable for formal events."
        ]
    })

    stop_words = set(stopwords.words('english'))
    def preprocess(text):
        tokens = word_tokenize(text.lower())
        return ' '.join([word for word in tokens if word.isalpha() and word not in stop_words])

    products['clean_description'] = products['description'].apply(preprocess)

    vectorizer = TfidfVectorizer()
    tfidf_matrix = vectorizer.fit_transform(products['clean_description'])

    cos_sim = cosine_similarity(tfidf_matrix, tfidf_matrix)

    sim_df = pd.DataFrame(cos_sim, index=products['product_name'], columns=products['product_name'])
    print("\n🔍 Product Similarity Matrix (TF-IDF):")
    print(sim_df.round(2))

    interactions = pd.DataFrame({
        'customer_id': [1, 1, 2, 2, 3, 3, 3],
        'product_id': [1, 2, 2, 3, 1, 4, 5]
    })

    interaction_matrix = interactions.pivot_table(index='customer_id', columns='product_id', aggfunc=lambda x: 1, fill_value=0)

    interaction_matrix_sparse = csr_matrix(interaction_matrix.values)
    user_sim = cosine_similarity(interaction_matrix_sparse)
    user_sim_df = pd.DataFrame(user_sim, index=interaction_matrix.index, columns=interaction_matrix.index)

    print("\n🤝 User Similarity Matrix:")
    print(user_sim_df.round(2))
```

```python
def recommend_products(customer_id, top_n=2):
    sim_scores = user_sim_df[customer_id].sort_values(ascending=False)
    sim_scores = sim_scores.drop(customer_id)

    similar_users = sim_scores.index
    weighted_scores = np.zeros(interaction_matrix.shape[1])

    for sim_user in similar_users:
        sim_score = sim_scores[sim_user]
        weighted_scores += sim_score * interaction_matrix.loc[sim_user].values

    already_bought = interaction_matrix.loc[customer_id].values
    recommendation_scores = weighted_scores * (1 - already_bought)

    top_indices = recommendation_scores.argsort()[::-1][:top_n]
    recommended_product_ids = interaction_matrix.columns[top_indices]
    return products[products['product_id'].isin(recommended_product_ids)]

print("\n🎯 Recommendations for Customer 1:")
print(recommend_products(1))

plt.figure(figsize=(8, 6))
sns.heatmap(sim_df, annot=True, cmap='coolwarm')
plt.title("Product Similarity Heatmap (TF-IDF)")
plt.show()
```

```
🔍 Product Similarity Matrix (TF-IDF):
product_name   Red Shirt  Blue Jeans  Running Shoes  Black Jacket  White Shirt
product_name
Red Shirt           1.00         0.0            0.0           0.0         0.12
Blue Jeans          0.00         1.0            0.0           0.0         0.00
Running Shoes       0.00         0.0            1.0           0.0         0.00
Black Jacket        0.00         0.0            0.0           1.0         0.00
White Shirt         0.12         0.0            0.0           0.0         1.00

🤝 User Similarity Matrix:
customer_id     1    2     3
customer_id
1            1.00  0.5  0.41
2            0.50  1.0  0.00
3            0.41  0.0  1.00

🎯 Recommendations for Customer 1:
   product_id   product_name                                  description  \
2           3  Running Shoes    Lightweight running shoes for daily exercise.
4           5     White Shirt  Classic white shirt suitable for formal events.

                          clean_description
2    lightweight running shoes daily exercise
4  classic white shirt suitable formal events
```