SESSION - (2024-25)

NATURAL LANGUAGE PROCESSING

LAB MANUAL

DEPARTMENT OF COMPUTER ENGINEERING AND APPLICATIONS

(CSE-AIML)

GLA. University

Submitted By-

Tripti Gupta(2215500161)

Submitted To-

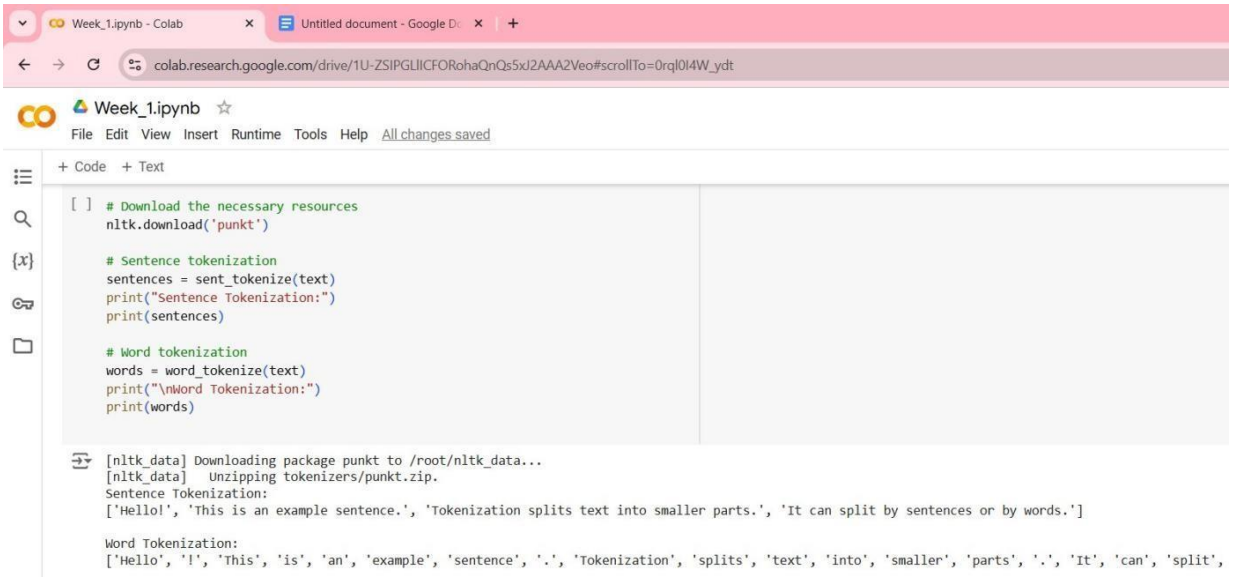
Mr.Ankur Mishra

Week -1

Aim: a) Write a python program to perform tokenization by word and sentence using nltk.

```
# Import the necessary modules from nltk
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize

# Sample text for tokenization
text = """Hello! This is an example sentence. Tokenization splits text
into smaller parts.
It can split by sentences or by words."""

# Download the necessary resources
nltk.download('punkt')

# Sentence tokenization
sentences = sent_tokenize(text)
print("Sentence Tokenization:")
print(sentences)

# Word tokenization
words = word_tokenize(text)
print("\nWord Tokenization:")
print(words)
```

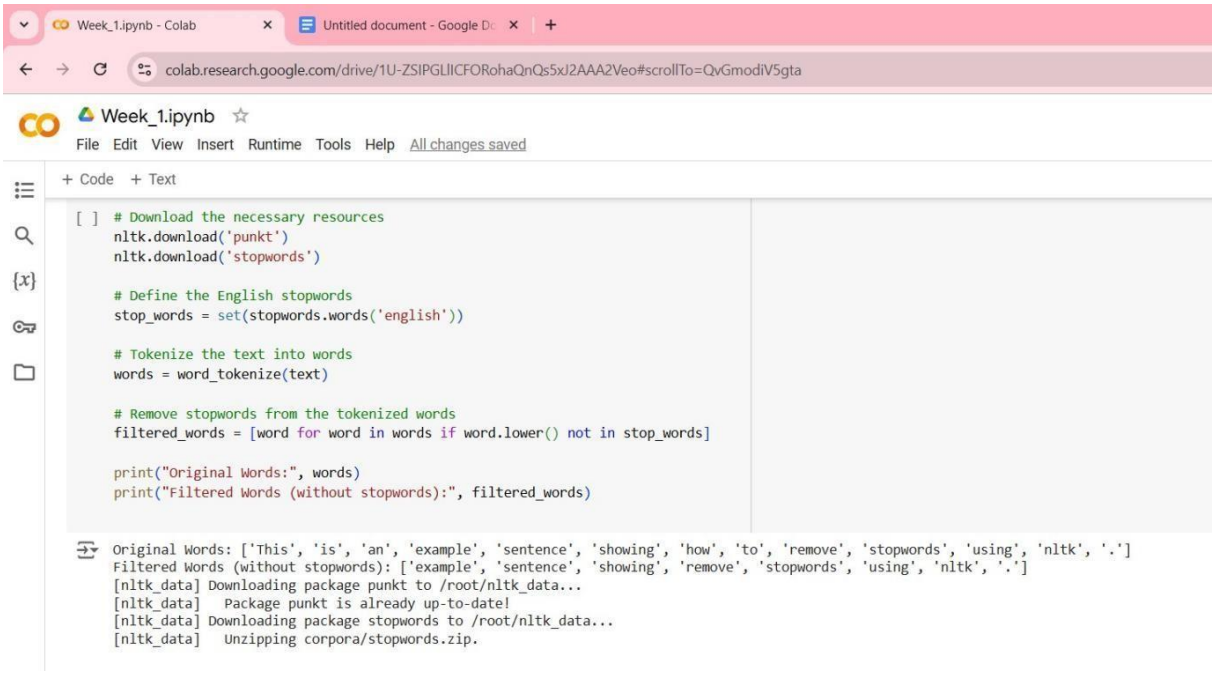
Output -



b) Write a python program to eliminate stopwords using nltk

```
# Import the necessary modules from nltk
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

# Sample text for stopword removal
text = "This is an example sentence showing how to remove stopwords
using nltk."

# Download the necessary resources
nltk.download('punkt')
nltk.download('stopwords')

# Define the English stopwords
stop_words = set(stopwords.words('english'))

# Tokenize the text into words
words = word_tokenize(text)

# Remove stopwords from the tokenized words
filtered_words = [word for word in words if word.lower() not in
stop_words]

print("Original Words:", words)
print("Filtered Words (without stopwords):", filtered_words)
```

Output-

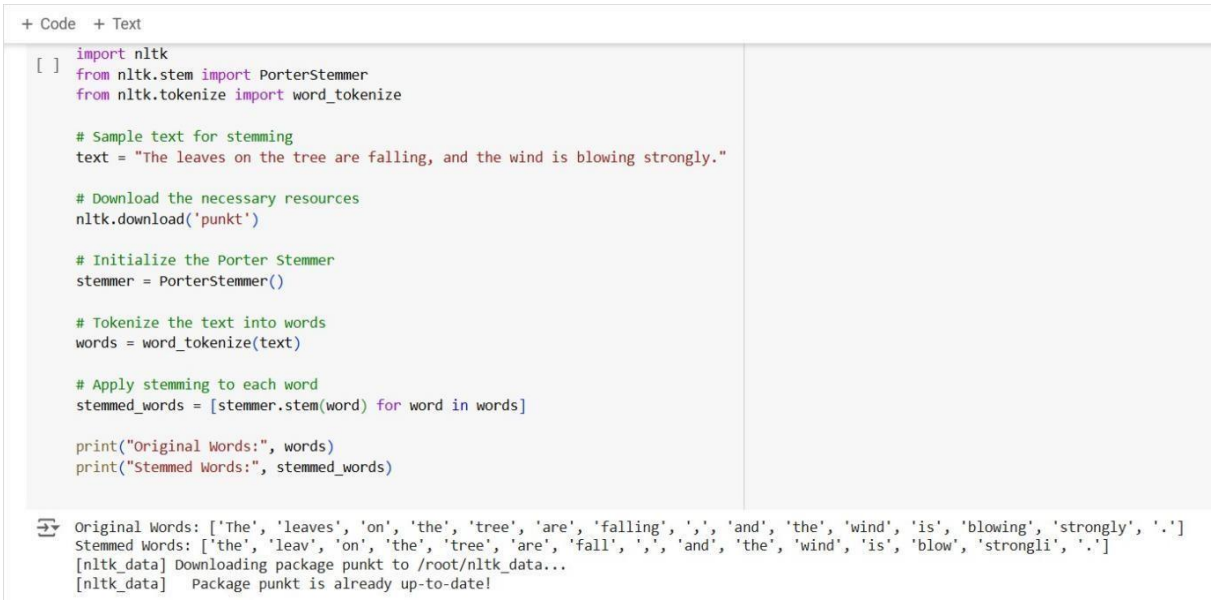


c.) Write a python program to perform stemming using nltk

```
# Import the necessary modules from nltk
import nltk
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize

# Sample text for stemming
text = "The leaves on the tree are falling, and the wind is blowing
strongly."

# Download the necessary resources
nltk.download('punkt')

# Initialize the Porter Stemmer
stemmer = PorterStemmer()

# Tokenize the text into words
words = word_tokenize(text)

# Apply stemming to each word
```

```
stemmed_words = [stemmer.stem(word) for word in words]

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

Output-



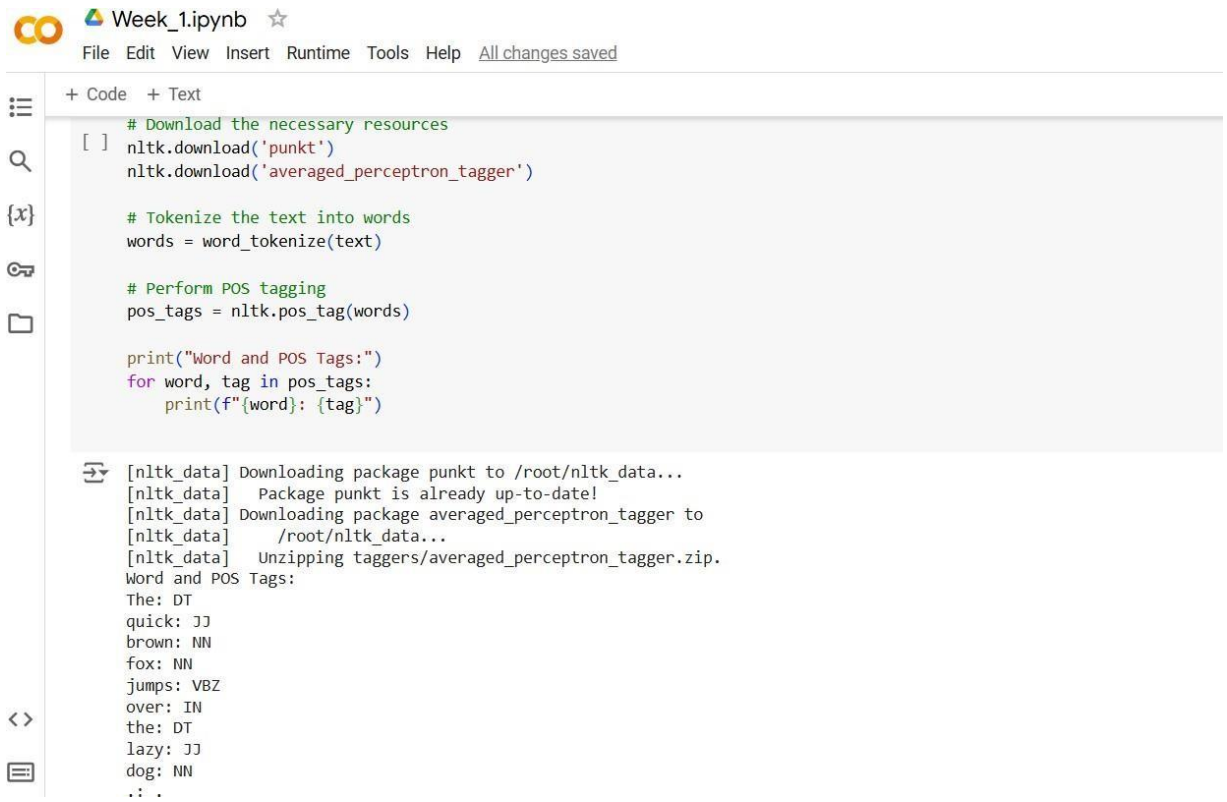
Week-2

a) Write a python program to perform Parts of Speech tagging using nltk

```
# Import the necessary modules from nltk
import nltk
from nltk.tokenize import word_tokenize

# Sample text for POS tagging
text = "The quick brown fox jumps over the lazy dog."

# Download the necessary resources
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

# Tokenize the text into words
```

```
words = word_tokenize(text)

# Perform POS tagging
pos_tags = nltk.pos_tag(words)

print("Word and POS Tags:")
for word, tag in pos_tags:
    print(f"{word}: {tag}")
```

Output-

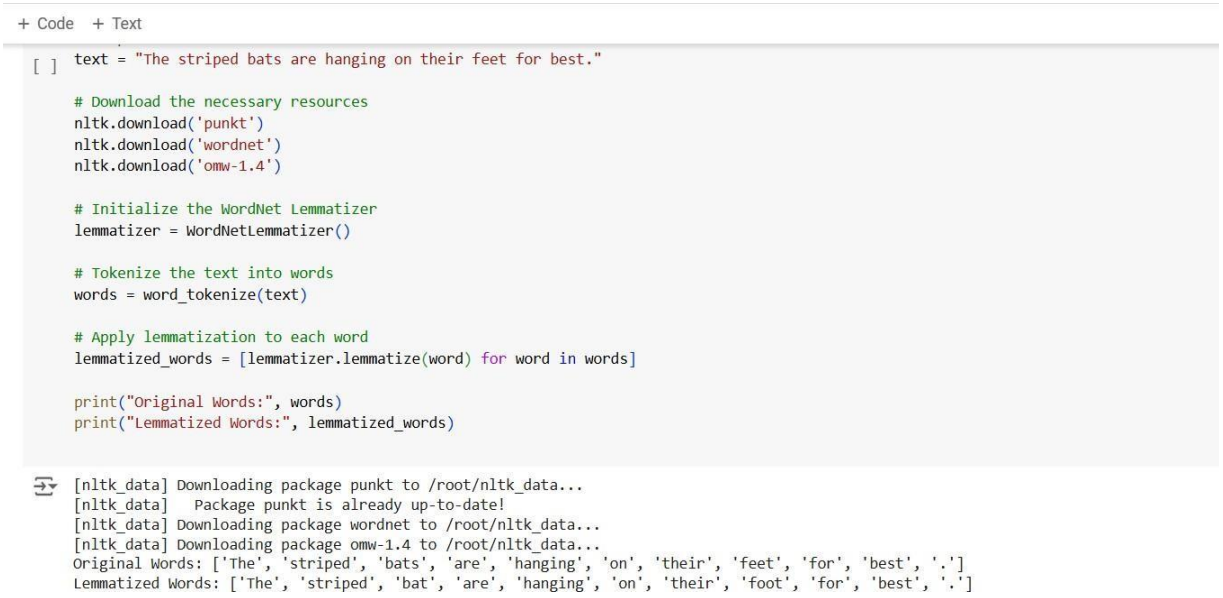


b) Write a python program to perform lemmatization using nltk.

```
# Import the necessary modules from nltk
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize

# Sample text for lemmatization
text = "The striped bats are hanging on their feet for best."

# Download the necessary resources
```

```
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('omw-1.4')

# Initialize the WordNet Lemmatizer
lemmatizer = WordNetLemmatizer()

# Tokenize the text into words
words = word_tokenize(text)

# Apply lemmatization to each word
lemmatized_words = [lemmatizer.lemmatize(word) for word in words]

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

Output-



Week - 3

a) Write a python program for chunking using nltk.

```
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.tag import pos_tag
```

```
def chunk_sentence(sentence):
    """Chunks a given sentence using NLTK's RegexpParser.

    Args:
        sentence: The sentence to be chunked.

    Returns:
        A list of chunked sentences.
    """

    # Tokenize the sentence into words
    words = word_tokenize(sentence)

    # Perform POS tagging
    pos_tags = pos_tag(words)

    # Define chunk grammar rules
    chunk_grammar = r"""
        NP: {<DT>?<JJ>*<NN>+}  # Noun Phrase
        VP: {<VB.*>}            # Verb Phrase
        PP: {<IN> <NP>}         # Prepositional Phrase
    """

    # Create a chunk parser
    chunk_parser = nltk.RegexpParser(chunk_grammar)

    # Parse the sentence
    chunked_sentence = chunk_parser.parse(pos_tags)

    return chunked_sentence

# Example usage
sentence = "The quick brown fox jumps over the lazy dog."
chunked_sentence = chunk_sentence(sentence)

print(chunked_sentence)
```

Output-

```
        # Create a chunk parser
        chunk_parser = nltk.RegexpParser(chunk_grammar)

        # Parse the sentence
        chunked_sentence = chunk_parser.parse(pos_tags)

        return chunked_sentence

# Example usage
sentence = "The quick brown fox jumps over the lazy dog."
chunked_sentence = chunk_sentence(sentence)

print(chunked_sentence)
```

```
(S
  (NP The/DT quick/JJ brown/NN fox/NN)
  (VP jumps/VBZ)
  (PP over/IN (NP the/DT lazy/JJ dog/NN))
  ./.)
```

B) Write a python program to perform Named Entity Recognition using nltk

```
# program for NER
import spacy
from spacy import displacy

#  Load the English language model
nlp = spacy.load("en_core_web_sm")

# Define your text
text = "Apple Inc. was founded by Steve Jobs and Steve Wozniak in California on April 1, 1976."


# Process the text with spaCy
doc = nlp(text)

displacy.render(doc, style="ent")
```

Output -

```
displacy.render(doc, style="ent")
```

Week-4

a) Write a python program to find Term Frequency and Inverse Document Frequency (TF-IDF)

```
# code to calculate TF
def compute_tf(doc):
    tf = {}
    total_words = len(doc)
    for word in doc:
        word_lower = word.lower()
        if word_lower in tf:
            tf[word_lower] += 1
        else:
            tf[word_lower] = 1
    # Normalize by total number of words in the document
    for word in tf:
        tf[word] /= total_words
    return tf
```

```
# Code to calculate IDF
def compute_idf(corpus):
    idf = {}
    total_docs = len(corpus)
    # Create a set of all words that appear in at least one document
    word_doc_count = {}
    for doc in corpus:
        words_in_doc = set([word.lower() for word in doc])
        for word in words_in_doc:
            if word not in word_doc_count:
                word_doc_count[word] = 1
            else:
                word_doc_count[word] += 1
    # Calculate IDF
    for word, count in word_doc_count.items():
        idf[word] = math.log(total_docs / (1 + count)) # Adding 1 to avoid division by zero
    return idf
```

```
# Code to calculate TF-IDF
def compute_tfidf(doc, tf, idf):
    tfidf = {}
    for word in tf:
        tfidf[word] = tf[word] * idf.get(word, 0)  # Use 0 for words not in IDF dictionary
    return tfidf
```

```
# Example usage
import math

# Sample corpus (list of documents)
corpus = [
    ["the", "sky", "is", "blue"],
    ["the", "sun", "is", "bright"],
    ["the", "sun", "in", "the", "sky", "is", "bright"],
    ["we", "can", "see", "the", "shining", "sun", "the", "bright", "sun"]
]

# Step 1: Compute TF for each document
tfs = [compute_tf(doc) for doc in corpus]

# Step 2: Compute IDF using the entire corpus
idf = compute_idf(corpus)

# Step 3: Compute TF-IDF for each document
tfidfs = [compute_tfidf(doc, tf, idf) for doc, tf in zip(corpus, tfs)]

# Output results
print("TF for each document:")
for i, tf in enumerate(tfs):
    print(f"Document {i+1} TF: {tf}")

print("\nIDF for the corpus:")
print(idf)

print("\nTF-IDF for each document:")
for i, tfidf in enumerate(tfidfs):
    print(f"Document {i+1} TF-IDF: {tfidf}")
```

Output -

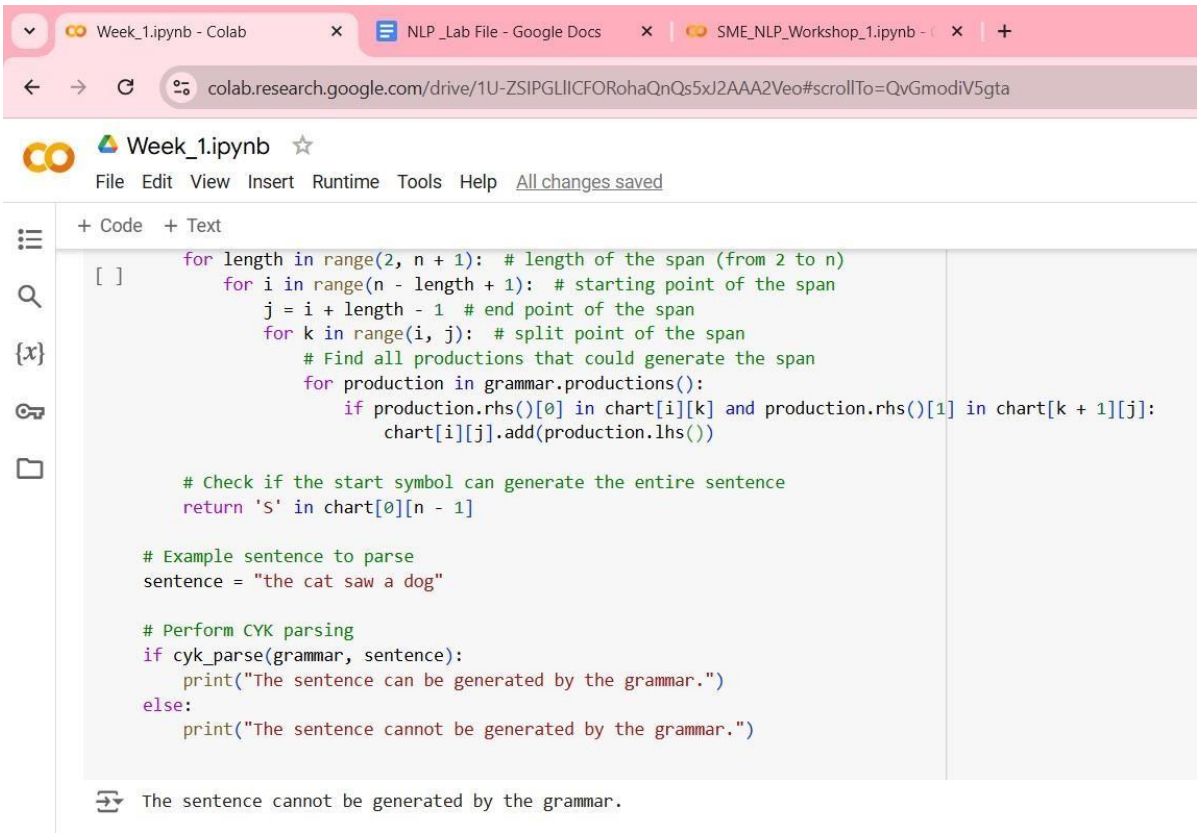
```
TF for each document:
Document 1 TF: {'the': 0.25, 'sky': 0.25, 'is': 0.25, 'blue': 0.25}
Document 2 TF: {'the': 0.25, 'sun': 0.25, 'is': 0.25, 'bright': 0.25}
Document 3 TF: {'the': 0.2857142857142857, 'sun': 0.14285714285714285, 'in': 0.14285714285714285, 'sky': 0.14285714285714285, 'is': 0.14285714285714285, '
Document 4 TF: {'we': 0.1111111111111111, 'can': 0.1111111111111111, 'see': 0.1111111111111111, 'the': 0.2222222222222222, 'shining': 0.1111111111111111,

IDF for the corpus:
{'is': 0.0, 'the': -0.2231435513142097, 'sky': 0.28768207245178085, 'blue': 0.6931471805599453, 'sun': 0.0, 'bright': 0.0, 'in': 0.6931471805599453, 'can'

TF-IDF for each document:
Document 1 TF-IDF: {'the': -0.05578588782855243, 'sky': 0.07192051811294521, 'is': 0.0, 'blue': 0.1732867951399863 2}
Document 2 TF-IDF: {'the': -0.05578588782855243, 'sun': 0.0, 'is': 0.0, 'bright': 0.0}
Document 3 TF-IDF: {'the': -0.06375530037548849, 'sun': 0.0, 'in': 0.09902102579427789, 'sky': 0.04109743892168297, 'is': 0.0, 'bright': 0.0}
Document 4 TF-IDF: {'we': 0.07701635339554948, 'can': 0.07701635339554948, 'see': 0.07701635339554948, 'the': -0.04958745584760216, 'shining': 0.077016353
```

b) Write a python program for CYK parsing (Cocke-Younger-Kasami Parsing) or Chart Parsing.

```
import nltk
from nltk import CFG

# Define a simple context-free grammar (CFG)
grammar = CFG.fromstring("""
  S -> NP VP
  VP -> V NP
  NP -> Det N
  V -> "saw" | "ate"
  Det -> "a" | "an" | "the"
  N -> "dog" | "cat" | "man"
""")

# Define the CYK parser function
def cyk_parse(grammar, sentence):
    # Tokenize the sentence
    sentence = sentence.split()

    # Initialize the chart (a list of lists of sets)
    n = len(sentence)
    chart = [[set() for _ in range(n)] for _ in range(n)]

    # Fill the chart with terminal symbols (words in the sentence)
    for j in range(n):
        for production in grammar.productions(rhs=sentence[j]):
            chart[j][j].add(production.lhs())

    # Fill the chart with non-terminal symbols for substrings of length
> 1
    for length in range(2, n + 1): # length of the span (from 2 to n)
        for i in range(n - length + 1):  # starting point of the span
            j = i + length - 1  # end point of the span
            for k in range(i, j):  # split point of the span
                # Find all productions that could generate the span
                for production in grammar.productions():
                    if production.rhs()[0] in chart[i][k] and
production.rhs()[1] in chart[k + 1][j]:
                        chart[i][j].add(production.lhs())

    # Check if the start symbol can generate the entire sentence
```

```
    return 'S' in chart[0][n - 1]

# Example sentence to parse
sentence = "the cat saw a dog"

# Perform CYK parsing
if cyk_parse(grammar, sentence):
    print("The sentence can be generated by the grammar.")
else:
    print("The sentence cannot be generated by the grammar.")
```

Output-



Week-5

a) Write a python program to find all unigrams, bigrams and trigrams present in the given corpus.

```
import nltk
from nltk.util import ngrams
from nltk.tokenize import word_tokenize
```

```
# Sample text (corpus)
corpus = "This is a simple example sentence for extracting unigrams,
bigrams, and trigrams."

# Tokenize the corpus into words
words = word_tokenize(corpus)

# Find unigrams (single words)
unigrams = list(ngrams(words, 1))

# Find bigrams (pairs of words)
bigrams = list(ngrams(words, 2))

# Find trigrams (triplets of words)
trigrams = list(ngrams(words, 3))

# Print the results
print("Unigrams:")
print(unigrams)

print("\nBigrams:")
print(bigrams)

print("\nTrigrams:")
print(trigrams)
```

Output-

b) Write a python program to find the probability of the given statement “This is my cat” by taking the an exmple corpus into consideration. 'This is a dog’, 'This is

a cat', 'I love
my cat',
'This is my name’

```
import nltk
from nltk import FreqDist
from nltk.util import bigrams, ngrams
from nltk.tokenize import word_tokenize

# Sample corpus
corpus = """
This is my cat. My cat is black. The cat is playing. This is a simple
sentence.
"""

# Tokenize the corpus into words
words = word_tokenize(corpus.lower()) # Lowercasing to make it
case-insensitive

# Unigram Model: Calculate the frequency distribution of words in the
corpus
unigram_freq = FreqDist(words)

# Bigram Model: Generate bigrams from the tokenized words
bigram_list = list(bigrams(words))
bigram_freq = FreqDist(bigram_list)

# Sentence for which we want to calculate the probability
sentence = "This is my cat"
sentence_tokens = word_tokenize(sentence.lower())

# Calculate Unigram Probability
def unigram_probability(sentence_tokens, unigram_freq):
    total_words = sum(unigram_freq.values())  # Total number of words
in the corpus
    prob = 1.0
    for word in sentence_tokens:
        prob *= unigram_freq[word] / total_words
    return prob

# Calculate Bigram Probability
def bigram_probability(sentence_tokens, bigram_freq, unigram_freq):
```

```
    prob = unigram_freq[sentence_tokens[0]] /
sum(unigram_freq.values())  # P(w1)
    for i in range(1, len(sentence_tokens)):
        prob *= bigram_freq[(sentence_tokens[i-1], sentence_tokens[i])]
/ unigram_freq[sentence_tokens[i-1]]
    return prob

# Calculate probabilities for the given sentence
unigram_prob = unigram_probability(sentence_tokens, unigram_freq)
bigram_prob = bigram_probability(sentence_tokens, bigram_freq,
unigram_freq)

# Print the results
print(f"Unigram Probability of '{sentence}': {unigram_prob}")
print(f"Bigram Probability of '{sentence}': {bigram_prob}")
```

Output-

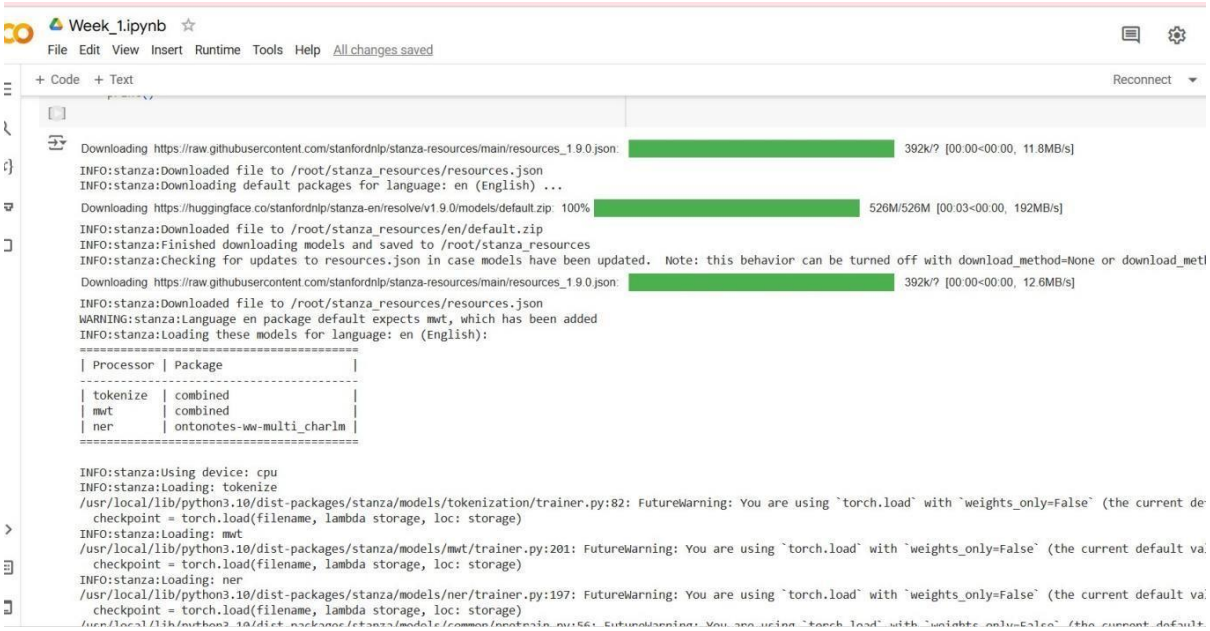
Week-6

Use the Stanford named Entity recognizer to extract entities from the documents. Use it programmatically and output for each document which named entities it contains and of Which type.

```
pip install stanza

import stanza

# Initialize the Stanford NER using stanza (which wraps Stanford's NER
models)
stanza.download('en') # This will download the English model
nlp = stanza.Pipeline('en', processors='tokenize,ner')

# Example list of documents
```

```
documents = [
    "Barack Obama was born in Hawaii. He was the 44th president of the United States.",
    "Apple Inc. is looking to expand its business in Europe. Tim Cook is the CEO of Apple.",
    "Elon Musk, the CEO of Tesla, plans to send humans to Mars by 2024."
]

# Function to extract Named Entities from a document
def extract_named_entities(doc):
    # Process the document through the NER pipeline
    doc = nlp(doc)

    # Extract and print named entities and their types
    entities = []
    for ent in doc.ents:
        entities.append((ent.text, ent.type))
    return entities

# Iterate over the documents and extract entities
for i, doc in enumerate(documents):
    print(f"Document {i+1}:")
    entities = extract_named_entities(doc)

    if entities:
        print("Named Entities and their Types:")
        for entity in entities:
            print(f"Entity: {entity[0]}, Type: {entity[1]}")
    else:
        print("No named entities found.")
    print()
```

Output-



Week-7

Choose any corpus available on the internet freely. For the corpus, for each document, count how many times each stop word occurs and find out which are the most frequently occurring stop words. Further, calculate the term frequency and inverse document frequency as The motivation behind this is basically to find out how important a document is to a given query. For e.g.: If the query is say: “The brown crow”. “The” is less important. “Brown” and “crow” are relatively more important. Since “the” is a more common word, its tf will be high. Hence we multiply it by idf, by knowing how common it is to reduce its weight.

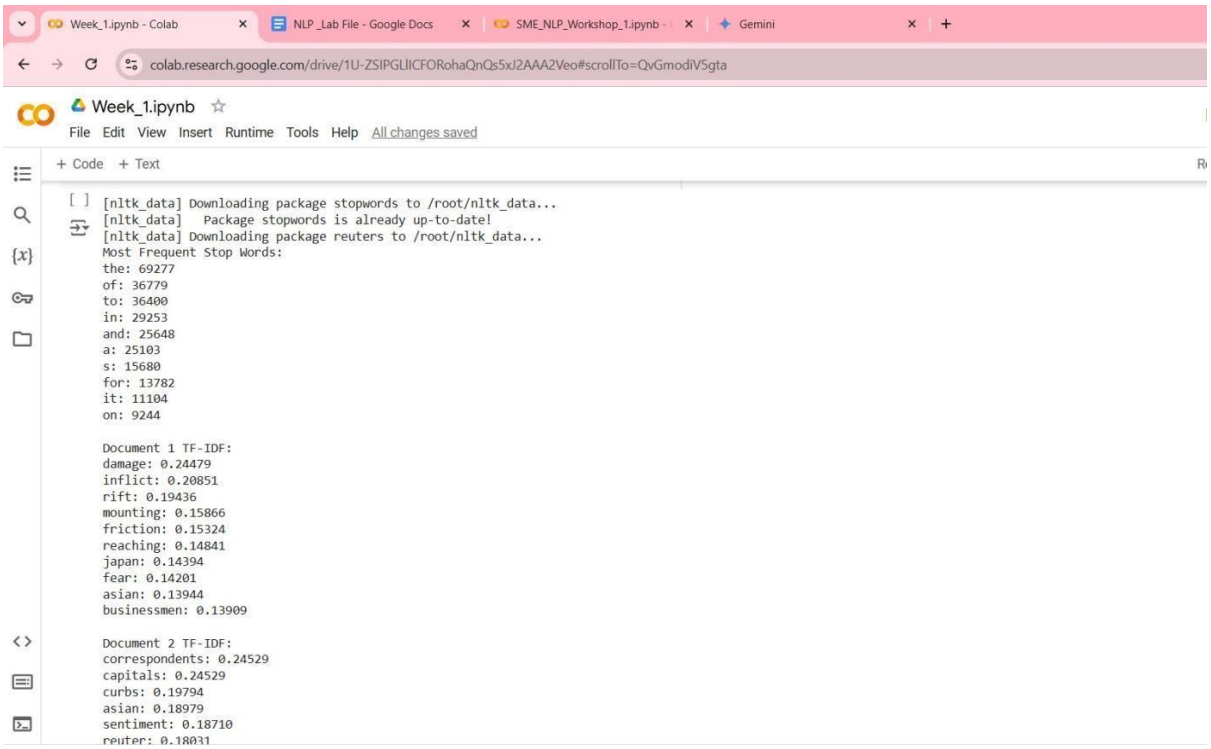
```
import nltk
from nltk.corpus import stopwords
from nltk.corpus import reuters
from nltk.probability import FreqDist
from nltk.text import TextCollection
import math

# Download required NLTK resources
nltk.download('stopwords')
nltk.download('reuters')

# Load the stop words
stop_words = set(stopwords.words('english'))
```

```
# Load the Reuters corpus
documents = reuters.sents()

# 1. Count how many times each stop word occurs in the documents
stop_word_counts = {}

for doc in documents:
    for word in doc:
        word_lower = word.lower()
        if word_lower in stop_words:
            if word_lower not in stop_word_counts:
                stop_word_counts[word_lower] = 1
            else:
                stop_word_counts[word_lower] += 1

# Sort stop words by frequency
sorted_stop_word_counts = sorted(stop_word_counts.items(), key=lambda
x: x[1], reverse=True)

# Print most frequent stop words
print("Most Frequent Stop Words:")
for word, count in sorted_stop_word_counts[:10]:
    print(f"{word}: {count}")

# 2. Calculate Term Frequency (TF)
def compute_tf(doc):
    tf = {}
    total_words = len(doc)
    for word in doc:
        word_lower = word.lower()
        if word_lower in tf:
            tf[word_lower] += 1
        else:
            tf[word_lower] = 1
    # Normalize by total number of words in the document
    for word in tf:
        tf[word] /= total_words
    return tf

# 3. Calculate Inverse Document Frequency (IDF)
def compute_idf(corpus):
    idf = {}
```

```
    total_docs = len(corpus)
    # Create a set of all words that appear in at least one document
    word_doc_count = {}
    for doc in corpus:
        words_in_doc = set([word.lower() for word in doc])
        for word in words_in_doc:
            if word not in word_doc_count:
                word_doc_count[word] = 1
            else:
                word_doc_count[word] += 1

    # Calculate IDF for each word
    for word, doc_count in word_doc_count.items():
        idf[word] = math.log(total_docs / (1 + doc_count))  # Smoothing
with +1

    return idf

# 4. Calculate TF-IDF
def compute_tfidf(doc, tf, idf):
    tfidf = {}
    for word in tf:
        tfidf[word] = tf[word] * idf.get(word, 0) # Use 0 for words
not in IDF dictionary
    return tfidf

# Select a few sample documents
sample_docs = documents[:5] # Use the first 5 documents as samples for
analysis

# Compute TF, IDF, and TF-IDF for the sample documents
corpus = documents # Full corpus for IDF computation
idf = compute_idf(corpus)

for idx, doc in enumerate(sample_docs):
    print(f"\nDocument {idx + 1} TF-IDF:")
    tf = compute_tf(doc)
    tfidf = compute_tfidf(doc, tf, idf)

    # Sort TF-IDF values by their score
    sorted_tfidf = sorted(tfidf.items(), key=lambda x: x[1],
reverse=True)
```

```
    # Print the top 10 TF-IDF scores for the document
    for word, score in sorted_tfidf[:10]:
        print(f"{word}: {score:.5f}")
```

Output-



Week-8

a. Write the python code to perform sentiment analysis using NLP

```
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer

# Download the VADER lexicon
nltk.download('vader_lexicon')

# Initialize the SentimentIntensityAnalyzer
sia = SentimentIntensityAnalyzer()

# Example texts
texts = [
    "I love this product! It's amazing and works perfectly.",
    "This is the worst experience I've ever had. So disappointed.",
    "I'm not sure how I feel about this. It's okay, I guess.",
```

```
    "Fantastic! I will definitely buy this again. Highly recommend!",
    "The movie was a bit long and boring. Could have been better."
]

# Perform sentiment analysis for each text
for text in texts:
    # Get the sentiment scores
    sentiment_score = sia.polarity_scores(text)

    # Determine the sentiment
    if sentiment_score['compound'] >= 0.05:
        sentiment = 'Positive'
    elif sentiment_score['compound'] <= -0.05:
        sentiment = 'Negative'
    else:
        sentiment = 'Neutral'

    # Print the result
    print(f"Text: {text}")
    print(f"Sentiment: {sentiment}")
    print(f"Sentiment Scores: {sentiment_score}")
    print()
```

Output-

```
[ ]         sentiment = 'Neutral'

        # Print the result
        print(f"Text: {text}")
        print(f"Sentiment: {sentiment}")
        print(f"Sentiment Scores: {sentiment_score}")
        print()

[nltk_data] Downloading package vader_lexicon to /root/nltk_data...
Text: I love this product! It's amazing and works perfectly.
Sentiment: Positive
Sentiment Scores: {'neg': 0.0, 'neu': 0.286, 'pos': 0.714, 'compound': 0.9259}

Text: This is the worst experience I've ever had. So disappointed.
Sentiment: Negative
Sentiment Scores: {'neg': 0.484, 'neu': 0.516, 'pos': 0.0, 'compound': -0.8173}

Text: I'm not sure how I feel about this. It's okay, I guess.
Sentiment: Neutral
Sentiment Scores: {'neg': 0.165, 'neu': 0.674, 'pos': 0.16, 'compound': -0.016}

Text: Fantastic! I will definitely buy this again. Highly recommend!
Sentiment: Positive
Sentiment Scores: {'neg': 0.0, 'neu': 0.341, 'pos': 0.659, 'compound': 0.865}

Text: The movie was a bit long and boring. Could have been better.
Sentiment: Positive
Sentiment Scores: {'neg': 0.162, 'neu': 0.634, 'pos': 0.204, 'compound': 0.1531}
```

Week-9

1. Write the python code to develop Spam Filter using NLP

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report
import string

# Download necessary resources
nltk.download('punkt')
nltk.download('stopwords')

# Sample Data - You can replace this with a dataset like 'SMS Spam
Collection Dataset'
# A list of (message, label) tuples
data = [
    ("Hey, how are you?", "ham"),
    ("Free cash prize, claim now!", "spam"),
    ("Call me when you get this message", "ham"),
    ("Limited time offer, win a lottery!", "spam"),
    ("Let's meet tomorrow", "ham"),
    ("Congratulations, you've won a free ticket", "spam"),
    ("Are we still meeting at 5?", "ham"),
    ("Earn money from home. Apply now", "spam"),
]

# Step 1: Preprocessing the text (lowercasing, removing punctuation,
stopwords, etc.)
def preprocess_text(text):
    text = text.lower()  # Convert to lowercase
    text = ''.join([char for char in text if char not in
string.punctuation]) # Remove punctuation
    words = word_tokenize(text) # Tokenize the text into words
    words = [word for word in words if word not in
stopwords.words('english')] # Remove stop words
    return ' '.join(words)

# Preprocess the data
messages, labels = zip(*data)
messages = [preprocess_text(msg) for msg in messages]
```

```
# Step 2: Convert text data into numerical features using
CountVectorizer (Bag of Words)
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(messages)

# Step 3: Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, labels,
test_size=0.3, random_state=42)

# Step 4: Train a Naive Bayes Classifier
model = MultinomialNB()
model.fit(X_train, y_train)

# Step 5: Predict the labels for the test set
y_pred = model.predict(X_test)

# Step 6: Evaluate the model
print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
print("Classification Report:")
print(classification_report(y_test, y_pred))

# Example usage: Predict if a new message is spam or ham
new_message = "You have won a free gift, claim it now!"
preprocessed_message = preprocess_text(new_message)
vectorized_message = vectorizer.transform([preprocessed_message])
prediction = model.predict(vectorized_message)

print(f"Message: '{new_message}'")
print(f"Prediction: {prediction[0]}") # spam or ham
```

Output-

```
[ ] print(f"Message: '{new_message}'")
    print(f"Prediction: {prediction[0]}")  # spam or ham

Accuracy: 0.3333333333333333
Classification Report:
              precision    recall  f1-score   support

         ham       0.33      1.00      0.50         1
        spam       0.00      0.00      0.00         2

    accuracy                           0.33         3
   macro avg       0.17      0.50      0.25         3
weighted avg       0.11      0.33      0.17         3

Message: 'You have won a free gift, claim it now!'
Prediction: ham
[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!
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with n
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with n
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with n
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

Week-10

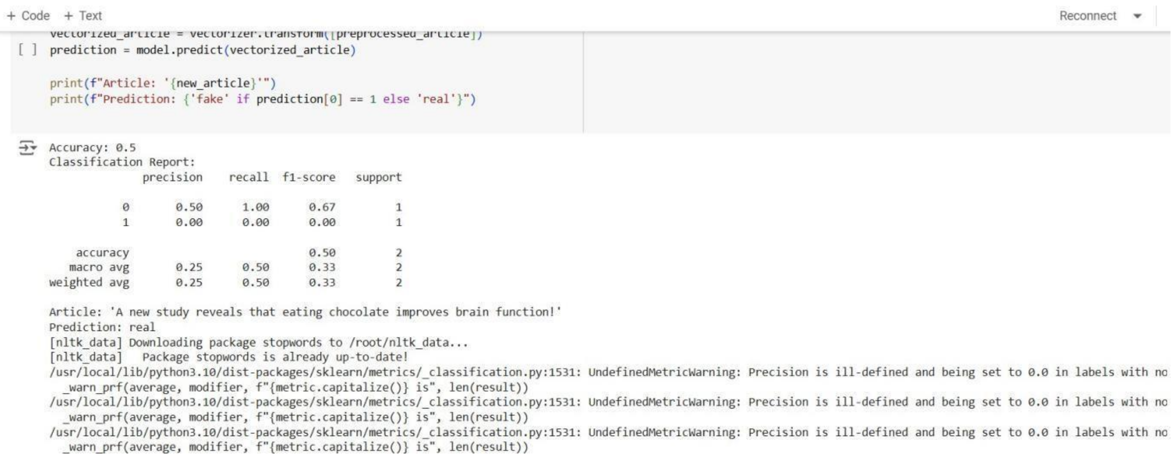
1. Write the python code to detect Fake News using NLP

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report
import string
import nltk

# Download necessary resources
nltk.download('stopwords')
from nltk.corpus import stopwords

# Sample Fake News Dataset (Replace this with a real dataset such as
"fake_news.csv")
# For this example, the data is structured as 'text' and 'label'
columns
data = {
    'text': [
        "Breaking: Scientists have discovered a new planet.",
        "New study finds that vaccines cause autism.",
        "Global warming is accelerating at an unprecedented rate.",
        "Aliens have made contact with Earth, according to new
reports.",
        "The government announces new tax relief measures for small
businesses."
    ],
    'label': ['real', 'fake', 'real', 'fake', 'real']
}

# Create DataFrame
df = pd.DataFrame(data)

# Step 1: Preprocessing the text (lowercasing, removing punctuation,
stopwords, etc.)
def preprocess_text(text):
    # Lowercase the text
    text = text.lower()
```

```
    # Remove punctuation
    text = ''.join([char for char in text if char not in
string.punctuation])

    # Tokenize and remove stopwords
    stop_words = set(stopwords.words('english'))
    words = text.split()
    words = [word for word in words if word not in stop_words]

    return ' '.join(words)

# Apply preprocessing
df['text'] = df['text'].apply(preprocess_text)

# Step 2: Convert text data into numerical features using TF-IDF
Vectorization
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(df['text'])

# Step 3: Convert labels to binary values (real -> 0, fake -> 1)
y = df['label'].map({'real': 0, 'fake': 1})

# Step 4: Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Step 5: Train a Logistic Regression classifier
model = LogisticRegression()
model.fit(X_train, y_train)

# Step 6: Predict the labels for the test set
y_pred = model.predict(X_test)

# Step 7: Evaluate the model
print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
print("Classification Report:")
print(classification_report(y_test, y_pred))

# Example usage: Predict if a new news article is fake or real
new_article = "A new study reveals that eating chocolate improves brain
function!"
preprocessed_article = preprocess_text(new_article)
vectorized_article = vectorizer.transform([preprocessed_article])
```

```
prediction = model.predict(vectorized_article)

print(f"Article: '{new_article}'")
print(f"Prediction: {'fake' if prediction[0] == 1 else 'real'}")
```

Output-



```
+ Code  + Text                                                          Reconnect

vectorized_article = vectorizer.transform([preprocessed_article])
[ ] prediction = model.predict(vectorized_article)

    print(f"Article: '{new_article}'")
    print(f"Prediction: {'fake' if prediction[0] == 1 else 'real'}")

Accuracy: 0.5
Classification Report:
              precision    recall  f1-score   support

           0       0.50      1.00      0.67         1
           1       0.00      0.00      0.00         1

    accuracy                           0.50         2
   macro avg       0.25      0.50      0.33         2
weighted avg       0.25      0.50      0.33         2

Article: 'A new study reveals that eating chocolate improves brain function!'
Prediction: real
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```