**PRACTICAL NO 1**

**Write a program to implement Tokenization of text**

**Code:**

print("131 SAKIB TAMBOLI")

import nltk

nltk.download()

data="Welcome to SIMS!!"

tokens=nltk.sent\_tokenize(data)

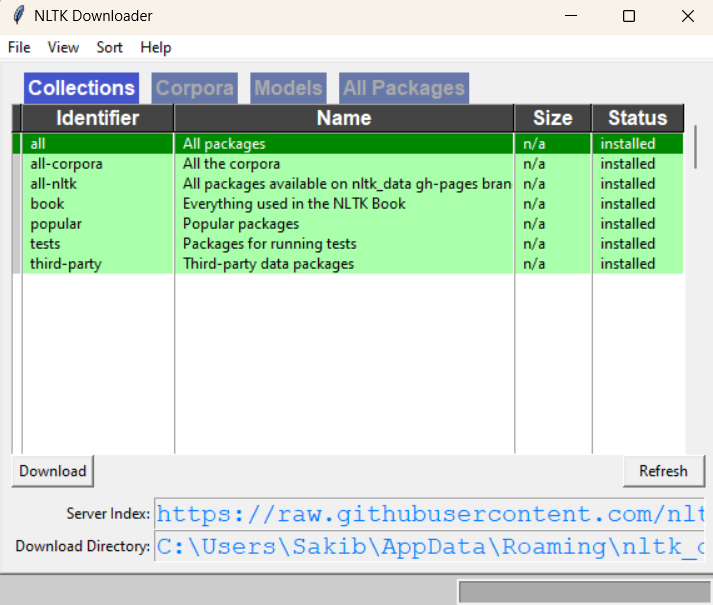
print(tokens)

tokens=nltk.word\_tokenize(data)

print(tokens)

**Output:**

****

****

**PRACTICAL NO 2**

**Write a program to implement Stop word removal.**

**Code:**

print("131 SAKIB TAMBOLI")

from nltk.corpus import stopwords

nltk.download('stopwords')

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

print(stopwords)

from nltk.tokenize import sent\_tokenize,word\_tokenize

from nltk.corpus import stopwords

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

data="All work and no play.All work and no play makes jack a dull boy"

tokens=nltk.word\_tokenize(data)

filtered\_Data=[]

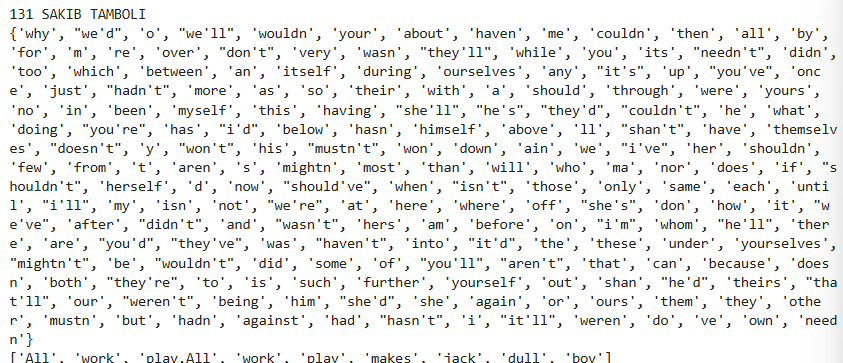
for w in tokens:

if w not in stopWords:

filtered\_Data.append(w)

print(filtered\_Data)

**Output:**

****

**PRACTICAL NO 3**

**Write a program to implement Stemming.**

**Code:**

print("131 SAKIB TAMBOLI")

from nltk.stem import PorterStemmer

port\_stemmer=PorterStemmer()

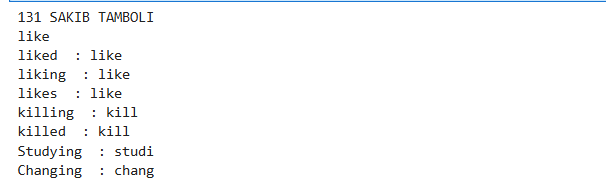
print(port\_stemmer.stem("Liked"))

data=["liked","liking","likes","killing","killed","Studying","Changing"]

for words in data:

print(words," :",port\_stemmer.stem(words))

**Output:**

****

**PRACTICAL NO 4**

**Write a program to implement Lemmatization.**

**Code:**

import nltk

from nltk.stem import WordNetLemmatizer

lemmati=WordNetLemmatizer()

print("131 Sakib Tamboli")

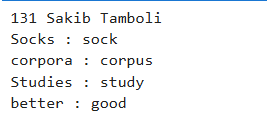
print("Socks :",lemmati.lemmatize("socks"))

print("corpora :",lemmati.lemmatize("corpora"))

print("Studies :",lemmati.lemmatize("studies"))

print("better :",lemmati.lemmatize("better",pos="a"))

**Output:**

****

**PRACTICAL NO 5**

**Write a program to implement the N-gram model.**

**Code:**

import nltk

nltk.download('punkt')

from nltk.util import ngrams

from nltk.tokenize import word\_tokenize

data="The little boy ran away"

#tokenize the text

token=nltk.word\_tokenize(data)

print("131 Sakib Tamboli")

Ngram=ngrams(token,3)

print("Trigram")

for gram in Ngram:

print(gram)

#BIGRAM

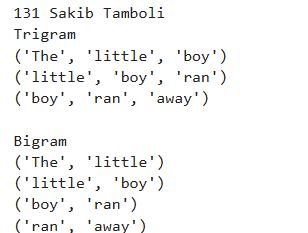
Ngram=ngrams(token,2)

print("\nBigram")

for gram in Ngram:

print(gram)

**Output:**

****

**PRACTICAL NO 6**

**Write a program to implement POS tagging.**

**Code:**

import nltk

nltk.download('punkt')

nltk.download('punk\_tab')

from nltk import pos\_tag

text="I enjoy coding and solving complex programming challenges."

tokens = text.split()

print ("Tokenized words: \n",tokens)

tagged\_words = pos\_tag(tokens)

print("\nPOS tagged words : \n", tagged\_words)

#Dispaly POS tags

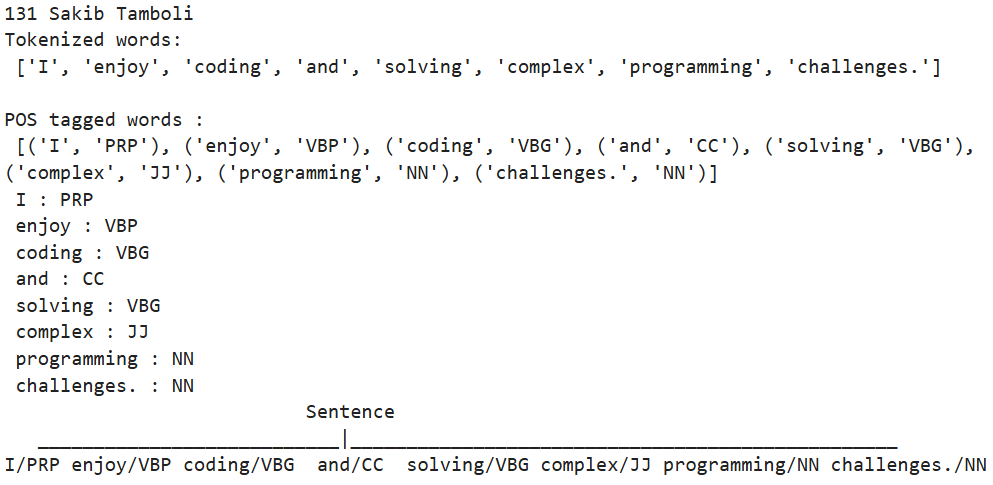
for word, tag in pos\_tags:

print(f" {word} : {tag}")

tree\_obj = nltk.Tree("Sentence", [(word, tag) for word, tag in pos\_tags])

tree\_obj.pretty\_print()

**Output:**

****

**PRACTICAL NO 7**

**Write a program to build a custom NER system**

**USING NLTK**

**Code:**

import nltk

from nltk import word\_tokenize,pos\_tag,ne\_chunk

nltk.download('punkt')

nltk.download('maxent\_ne\_chunker')

nltk.download('words')

#Sample text

text="I was born in Mumbai and went to Pune University for pursuing further studies"

#Tokenizing the text

tokens=word\_tokenize(text)

# Part of speech tagging

tagged\_tokens=pos\_tag(tokens)

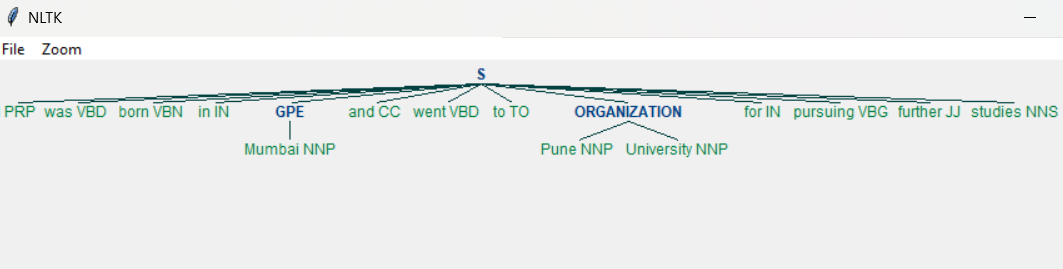
#Named Enitity Recognition (NE Chunking)

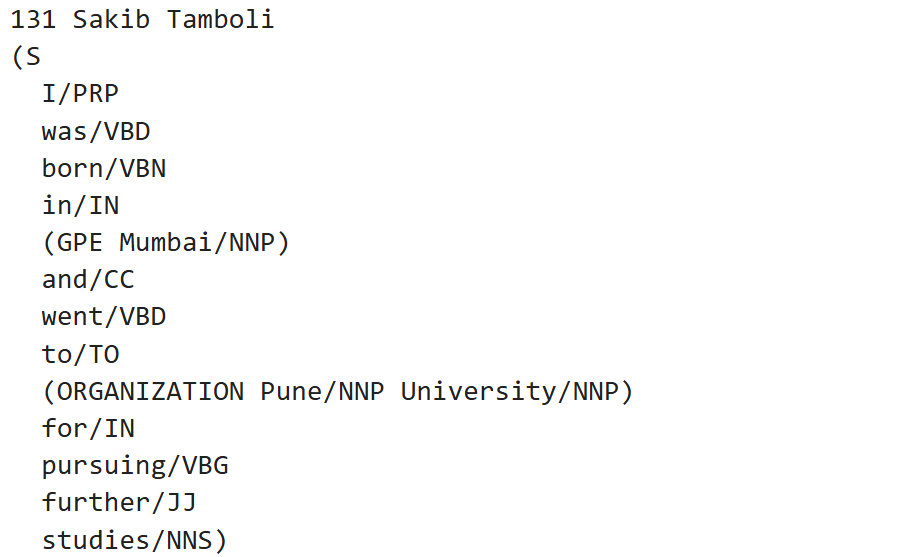
named\_entities=ne\_chunk(tagged\_tokens)

#Print the Named Entities

print(named\_entities)

named\_entities.draw()

**Output:**

****

**SPACY**

**Code:**

import spacy

# Load spaCy's English NER model

nlp = spacy.load("en\_core\_web\_sm")

# Sample text

#text = "I was born in Mumbai and went to Pune University to pursue further studies in August 2024."

text = "Jamshedji Tata founded Tata Steel on 26 August 1907 in India.He was very fluent in English.During the First World War (1914–1918), the Tata Group made rapid progress. Tata Motors stock is worth ₹800 today."

# Process the text using spaCy

doc = nlp(text)

# Print Named Entities

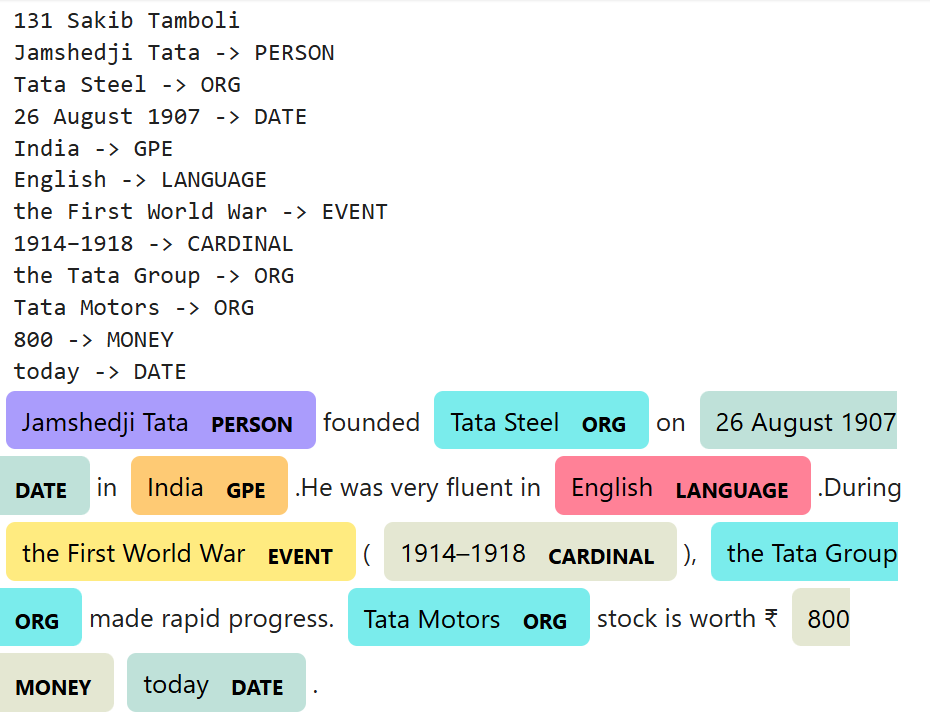
for ent in doc.ents:

print(ent.text, "->", ent.label\_)

# Visualizing Named Entities (Optional, works in Jupyter Notebook)

spacy.displacy.render(doc, style="ent", jupyter=True)

**Output:**

****

**PRACTICAL NO 8**

**Creating and comparing different text representations.**

**a)Write a program to create a bag of words(bow) text representation.**

**Code:**

# Bag of Words

import nltk

import numpy as np

# Ensure you have the necessary NLTK resources

nltk.download('punkt')

texts = [

"The cat sat on the mat",

"The dog sat on the log"

]

# Tokenize the texts

tokenized\_texts = [nltk.word\_tokenize(text.lower()) for text in texts]

# Create a vocabulary (set of all unique words)

vocabulary = sorted(set(word for text in tokenized\_texts for word in text))

print("Vocabulary:", vocabulary)

# Bag of Words (BoW) representation

def get\_bow\_representation(tokens, vocabulary):

return [tokens.count(word) for word in vocabulary]

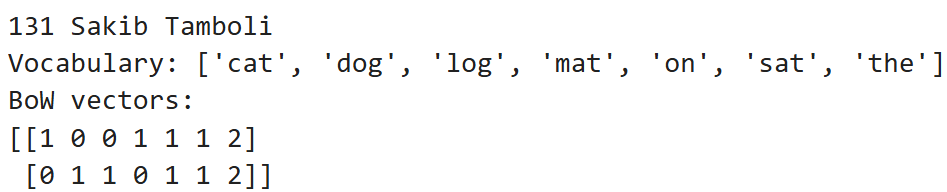
bow\_vectors = [get\_bow\_representation(text, vocabulary) for text in tokenized\_texts]

# Print BoW vectors

print("BoW vectors:")

print(np.array(bow\_vectors))

**Output:**

****

**b)Write a program to create tf\_idf text representations.**

**Code:**

import nltk

import numpy as np

from collections import Counter

from math import log

# Ensure you have the necessary NLTK resources

nltk.download('punkt')

texts = [

"The cat sat on the mat",

"The dog sat on the log"

]

# Tokenize the texts

tokenized\_texts = [nltk.word\_tokenize(text.lower()) for text in texts]

# Create a vocabulary (set of all unique words)

vocabulary = set(word for text in tokenized\_texts for word in text)

print("Vocabulary:", vocabulary)

# Function to compute Term Frequency (TF)

def get\_tf(tokens, vocabulary):

tf\_vector = [tokens.count(word) for word in vocabulary]

print("\nTF vectors:")

print(tf\_vector)

return tf\_vector

# Function to compute Inverse Document Frequency (IDF)

def get\_idf(vocabulary, docs):

num\_docs = len(docs)

idf\_vector = []

for word in vocabulary:

# Count the number of documents containing the word

num\_docs\_with\_word = sum(1 for doc in docs if word in doc)

# Calculate IDF as log(num\_docs / (1 + num\_docs\_with\_word)) to avoid division by zero

idf\_value = log(num\_docs / (1 + num\_docs\_with\_word)) + 1 # Adding 1 to smooth

idf\_vector.append(idf\_value)

return idf\_vector # Fix indentation to return after loop completes

# Function to compute TF-IDF

def get\_tfidf(tokens, vocabulary, idf\_vector):

tf\_vector = get\_tf(tokens, vocabulary)

tfidf\_vector = [tf \* idf for tf, idf in zip(tf\_vector, idf\_vector)]

return tfidf\_vector

# Calculate IDF for the entire corpus

idf\_vector = get\_idf(vocabulary, tokenized\_texts)

print("\nIDF vectors:")

print(idf\_vector)

# Compute TF-IDF for each document

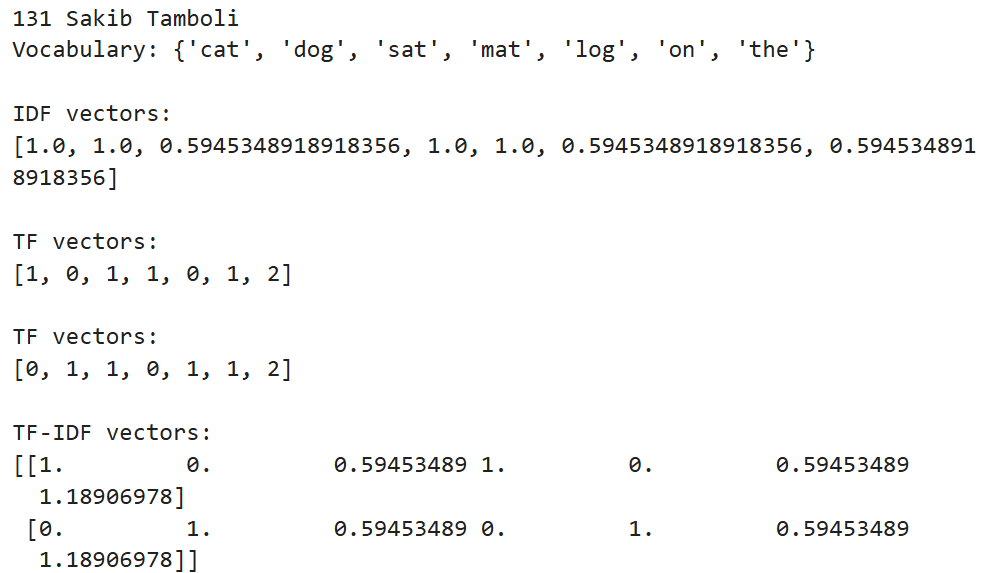
tfidf\_vectors = [get\_tfidf(text, vocabulary, idf\_vector) for text in tokenized\_texts]

# Print TF-IDF vectors

print("\nTF-IDF vectors:")

print(np.array(tfidf\_vectors))

**Output:**

****

**c) Write a program to compare two vectors of bow using cosine similarity.**

**Code:**

import nltk

import numpy as np

from sklearn.metrics.pairwise import cosine\_similarity

# Ensure you have the necessary NLTK resources

nltk.download('punkt')

texts = [

"The cat sat on the mat",

"The dog sat on the log"

]

# Tokenize the texts

tokenized\_texts = [nltk.word\_tokenize(text.lower()) for text in texts]

# Create a vocabulary (set of all unique words)

vocabulary = set(word for text in tokenized\_texts for word in text)

print(vocabulary)

# Bag of Words (BoW) representation

def get\_bow\_representation(tokens, vocabulary):

return [tokens.count(word) for word in vocabulary]

bow\_vectors = [get\_bow\_representation(text, vocabulary) for text in tokenized\_texts]

# Print BoW vectors

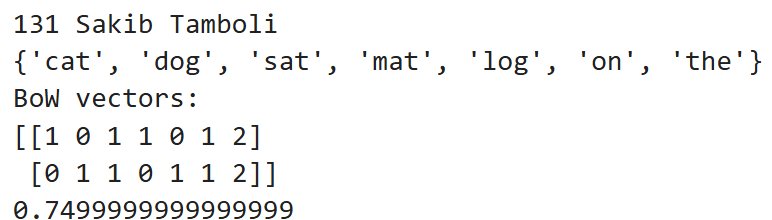
print("BoW vectors:")

print(np.array(bow\_vectors))

bow\_similarity = cosine\_similarity([bow\_vectors[0]], [bow\_vectors[1]])[0][0]

print(bow\_similarity)

**Output:**



**d)Write a program to compare a bow vector with a tf-idf vector using cosine similarity.**

**Code:**

import nltk

import numpy as np

from collections import Counter

from math import log

from sklearn.metrics.pairwise import cosine\_similarity

# Ensure you have the necessary NLTK resources

nltk.download('punkt')

texts = [

"The cat sat on the mat",

"The dog sat on the log"

]

# Tokenize the texts

tokenized\_texts = [nltk.word\_tokenize(text.lower()) for text in texts]

# Create a vocabulary (set of all unique words)

vocabulary = set(word for text in tokenized\_texts for word in text)

print("Vocabulary:", vocabulary)

# Bag of Words (BoW) representation

def get\_bow\_representation(tokens, vocabulary):

return [tokens.count(word) for word in vocabulary]

bow\_vectors = [get\_bow\_representation(text, vocabulary) for text in tokenized\_texts]

# Function to compute Term Frequency (TF)

def get\_tf(tokens, vocabulary):

return [tokens.count(word) for word in vocabulary]

def get\_idf(vocabulary, docs):

num\_docs = len(docs)

idf\_vector = []

for word in vocabulary:

# Count the number of documents containing the word

num\_docs\_with\_word = sum(1 for doc in docs if word in doc)

# Calculate IDF as log(num\_docs / (1 + num\_docs\_with\_word)) to avoid division by zero

idf\_value = log(num\_docs / (1 + num\_docs\_with\_word)) + 1 # Adding 1 to smooth

idf\_vector.append(idf\_value) # ✅ Collect all values

return idf\_vector # ✅ Return the full vector \*\*after\*\* the loop

# Function to compute TF-IDF

def get\_tfidf(tokens, vocabulary, idf\_vector):

tf\_vector = get\_tf(tokens, vocabulary)

tfidf\_vector = [tf \* idf for tf, idf in zip(tf\_vector, idf\_vector)]

return tfidf\_vector

# Calculate IDF for the entire corpus

idf\_vector = get\_idf(vocabulary, tokenized\_texts)

# print("\nIDF vectors:")

# print(idf\_vector)

# Compute TF-IDF for each document

tfidf\_vectors = [get\_tfidf(text, vocabulary, idf\_vector) for text in tokenized\_texts]

# Compute cosine similarity between BoW and TF-IDF vectors for doc1

bow\_similarity = cosine\_similarity([bow\_vectors[0]], [tfidf\_vectors[0]])[0][0]

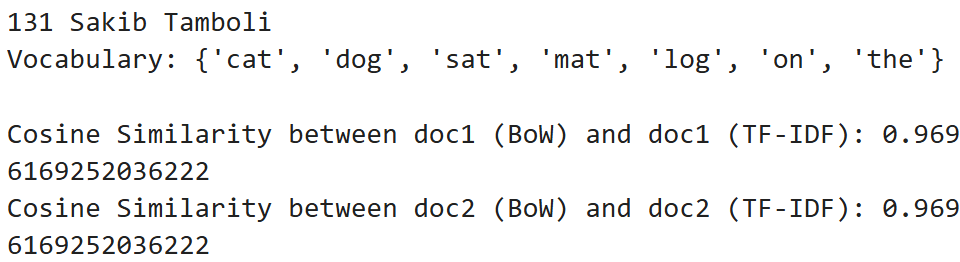
print("\nCosine Similarity between doc1 (BoW) and doc1 (TF-IDF):", bow\_similarity)

# Compute cosine similarity between BoW and TF-IDF vectors for doc2

bow\_similarity = cosine\_similarity([bow\_vectors[1]], [tfidf\_vectors[1]])[0][0]

print("Cosine Similarity between doc2 (BoW) and doc2 (TF-IDF):", bow\_similarity)

**Output:**

****

**PRACTICAL NO 9**

**Write a Program for Training and use word embedding Word2vec/GloVe.**

**Word2vec**

**Code:**

from gensim.models import Word2Vec

from nltk.tokenize import word\_tokenize

import nltk

# Download the punkttokenizer models from NLTK

nltk.download('punkt')

# Function to train Word2Vec model

def train\_word\_embeddings(sentences):

# Tokenize sentences using NLTK word\_tokenize and convert to lowercase

tokenized\_sentences = [word\_tokenize(sentence.lower()) for sentence in sentences]

# Train Word2Vec model

model = Word2Vec(sentences=tokenized\_sentences, vector\_size=100, window=5, min\_count=1, workers=4)

return model

# Function to use trained Word2Vec model and find similar words

def use\_word\_embeddings(model, word, top\_n=5):

try:

# Get the top N similar words to the input word

similar\_words = model.wv.most\_similar(word, topn=top\_n)

print(f"Words most similar to '{word}':")

for w, score in similar\_words:

print(f"{w}: {score:.4f}")

except KeyError:

print(f"'{word}' not in vocabulary")

# Example usage

sentences = [

"The quick brown fox jumps over the lazy dog",

"A fox is a cunning animal",

"The dog barks at night",

"Foxes and dogs are different species"

]

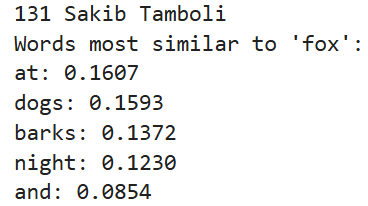
# Train Word2Vec model using the provided sentences

model = train\_word\_embeddings(sentences)

# Use the trained model to find words similar to "fox"

use\_word\_embeddings(model, "fox")

**Output:**

****

**Glove:**

**Code:**

pip install numpy scipy gensim tqdm

import re

import nltk

from collections import Counter

from itertools import product

import numpy as np

from scipy.sparse import coo\_matrix

nltk.download('punkt')

# Sample text corpus

text\_corpus = [

"Deep learning is a subset of machine learning.",

"Word embeddings capture semantic meaning.",

"Neural networks are used in NLP tasks.",

]

# Preprocessing: Tokenization and Lowercasing

def preprocess\_text(text):

text = text.lower()

text = re.sub(r'[^a-z\s]', '', text) # Remove punctuation

return nltk.word\_tokenize(text)

# Tokenize all sentences

tokenized\_corpus = [preprocess\_text(sentence) for sentence in text\_corpus]

# Define context window size

WINDOW\_SIZE = 2

# Count word co-occurrences

word\_counts = Counter(word for sentence in tokenized\_corpus for word in sentence)

vocab = list(word\_counts.keys())

word\_to\_id = {word: i for i, word in enumerate(vocab)}

id\_to\_word = {i: word for word, i in word\_to\_id.items()}

# Create co-occurrence matrix

co\_occurrence = Counter()

for sentence in tokenized\_corpus:

for i, word in enumerate(sentence):

word\_id = word\_to\_id[word]

for j in range(max(0, i - WINDOW\_SIZE), min(len(sentence), i + WINDOW\_SIZE + 1)):

if i != j: # Skip self-pairing

co\_occurrence[(word\_id, word\_to\_id[sentence[j]])] += 1

# Convert to sparse matrix

rows, cols, data = zip(\*[(i, j, count) for (i, j), count in co\_occurrence.items()])

X = coo\_matrix((data, (rows, cols)), shape=(len(vocab), len(vocab)))

print("Vocabulary Size:", len(vocab))

print("Sample Co-occurrence Matrix:", X.toarray())

import scipy.sparse.linalg

EMBEDDING\_DIM = min(50, X.shape[0] - 1) # Ensure k is smaller than the matrix size

# Compute the Positive Pointwise Mutual Information (PPMI) matrix

def ppmi\_matrix(X):

total\_sum = X.sum()

sum\_over\_words = np.array(X.sum(axis=0)).flatten()

sum\_over\_contexts = np.array(X.sum(axis=1)).flatten()

expected\_counts = np.outer(sum\_over\_contexts, sum\_over\_words) / total\_sum

nonzero\_indices = X.toarray() > 0 # Avoid division by zero

ppmi = np.log((X.toarray() / expected\_counts) + 1) \* nonzero\_indices

return np.nan\_to\_num(ppmi) # Replace NaN values with zero

# Compute PPMI and apply Singular Value Decomposition (SVD)

ppmi\_X = ppmi\_matrix(X)

U, S, Vt = scipy.sparse.linalg.svds(ppmi\_X, k=EMBEDDING\_DIM)

# Extract word embeddings

word\_vectors = U @ np.diag(np.sqrt(S)) # Take the square root of singular values

# Store word embeddings

word\_embeddings = {id\_to\_word[i]: word\_vectors[i] for i in range(len(vocab))}

from sklearn.metrics.pairwise import cosine\_similarity

def find\_similar\_words(word, top\_n=5):

if word not in word\_embeddings:

return "Word not in vocabulary"

word\_vector = word\_embeddings[word].reshape(1, -1)

similarities = {other\_word: cosine\_similarity(word\_vector, word\_embeddings[other\_word].reshape(1, -1))[0][0]

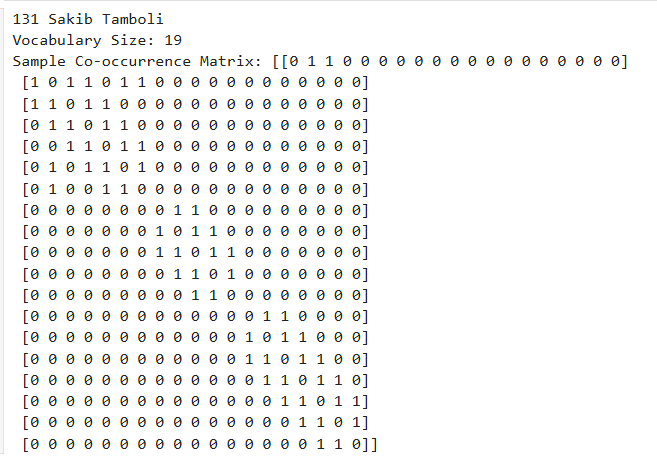
for other\_word in vocab if other\_word != word}

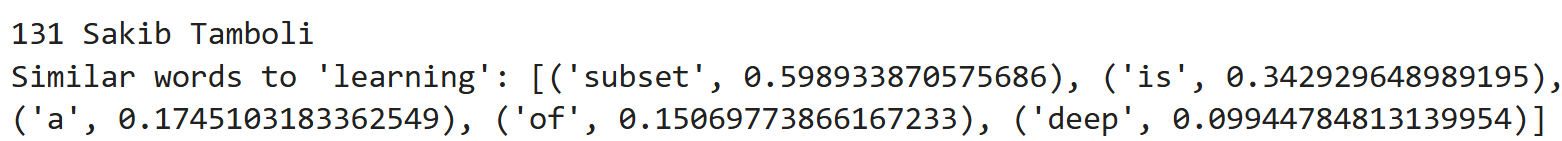
return sorted(similarities.items(), key=lambda x: x[1], reverse=True)[:top\_n]

# Example: Find similar words to "learning"

print("Similar words to 'learning':", find\_similar\_words("learning"))

**Output:**





**PRACTICAL NO 10**

**Write a Program to Implement a text classifier using NaiveBayes with scikit-learn.**

**Code:**

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.naive\_bayes import MultinomialNB

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import classification\_report

def train\_text\_classifier(X, y):

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a CountVectorizer

vectorizer = CountVectorizer()

X\_train\_vectorized = vectorizer.fit\_transform(X\_train)

X\_test\_vectorized = vectorizer.transform(X\_test)

# Train a Naive Bayes classifier

classifier = MultinomialNB()

classifier.fit(X\_train\_vectorized, y\_train)

# Make predictions

y\_pred = classifier.predict(X\_test\_vectorized)

# Print classification report

print(classification\_report(y\_test, y\_pred))

return vectorizer, classifier

def classify\_text(text, vectorizer, classifier):

text\_vectorized = vectorizer.transform([text])

prediction = classifier.predict(text\_vectorized)

return prediction[0]

# Example usage

X = [

"I love this movie, it's amazing!",

"This book is terrible, I couldn't finish it.",

"The food at this restaurant is delicious.",

"The service here is awful, I'm never coming back.",

"What a great experience, highly recommended!",

]

y = ["positive", "negative", "positive", "negative", "positive"]

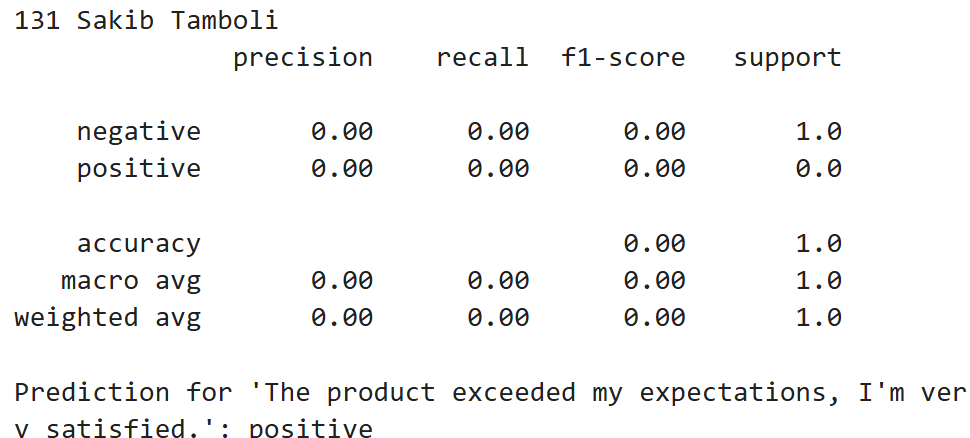
vectorizer, classifier = train\_text\_classifier(X, y)

new\_text = "The product exceeded my expectations, I'm very satisfied."

prediction = classify\_text(new\_text, vectorizer, classifier)

print(f"Prediction for '{new\_text}': {prediction}")

**Output:**

****

**PRACTICAL NO 11**

**Write a Program to Build a sentiment analysis system..**

**Code:**

import nltk

from nltk.sentiment import SentimentIntensityAnalyzer

import pandas as pd

nltk.download('vader\_lexicon')

def analyze\_sentiment(text):

sia = SentimentIntensityAnalyzer()

sentiment\_scores = sia.polarity\_scores(text)

if sentiment\_scores['compound'] >= 0.1:

sentiment = "Positive"

elif sentiment\_scores['compound'] <= -0.1:

sentiment = "Negative"

else:

sentiment = "Neutral"

return sentiment, sentiment\_scores

def analyze\_sentiments(texts):

results = []

for text in texts:

sentiment, scores = analyze\_sentiment(text)

results.append({

'text': text,

'sentiment': sentiment,

'pos\_score': scores['pos'],

'neg\_score': scores['neg'],

'neu\_score': scores['neu'],

'compound\_score': scores['compound']

})

return pd.DataFrame(results)

# Example usage

texts = [

"I absolutely love this product! It's amazing!",

"This is the worst experience I've ever had.",

"The movie was okay, nothing special.",

"I'm feeling pretty neutral about the whole situation.",

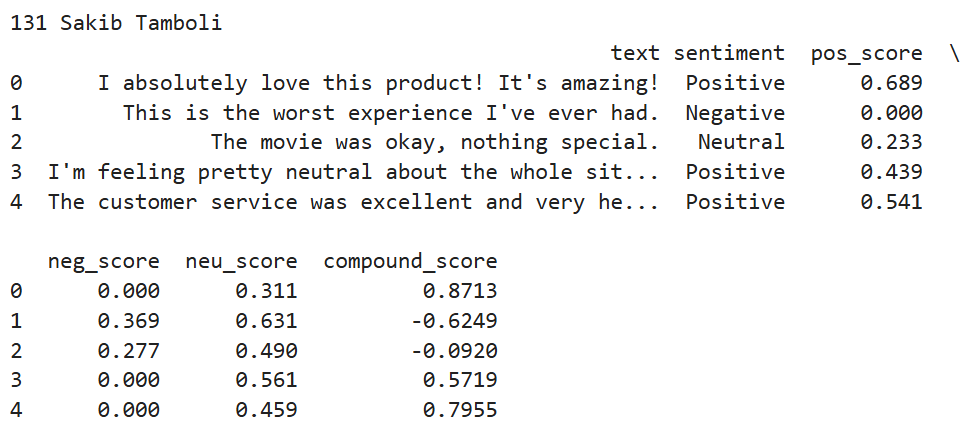
"The customer service was excellent and very helpful!"

]

results\_df = analyze\_sentiments(texts)

print(results\_df)

**Output:**



**PRACTICAL NO 12**

**Write a Program to create a text summarization tool.**

**Code:**

#!pip install transformers

#!pip install torch

from transformers import pipeline

def summarize\_text(text, max\_length=150, min\_length=50):

summarizer = pipeline("summarization", model="facebook/bart-large-cnn")

summary = summarizer(text, max\_length=max\_length, min\_length=min\_length, do\_sample=False)

return summary[0]['summary\_text']

# Example usage

long\_text = """

Climate change is one of the most pressing issues facing our planet today.

It refers to long-term shifts in temperatures and weather patterns, mainly caused by human activities, especially the burning of fossil fuels.

These activities release greenhouse gases into the atmosphere, trapping heat and causing the Earth's average temperature to rise.

The consequences of climate change are far-reaching and include more frequent and severe weather events, rising sea levels, and disruptions to ecosystems. To address this global challenge, countries and organizations worldwide are working on strategies to reduce greenhouse gas emissions and transition to cleaner energy sources. Individual actions, such as reducing energy consumption and adopting sustainable practices, also play a crucial role in mitigating the effects of climate change.

"""

summary = summarize\_text(long\_text)

print("Abstractive text summarization")

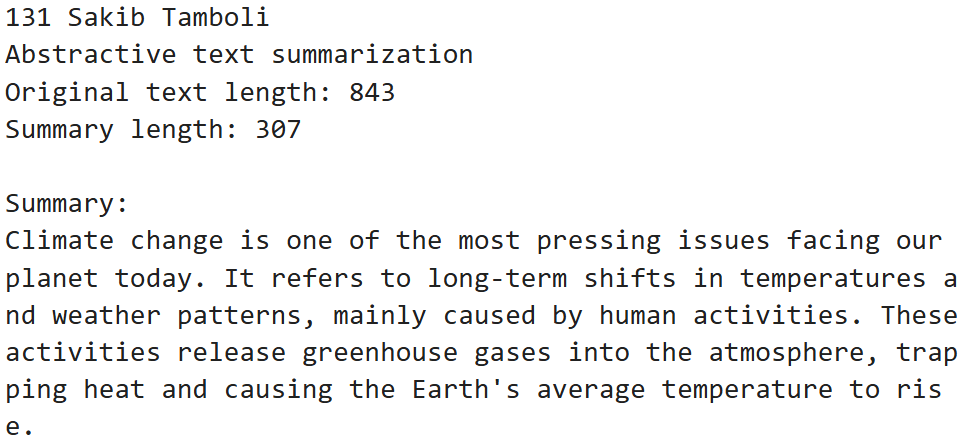
print("Original text length:", len(long\_text))

print("Summary length:", len(summary))

print("\nSummary:")

print(summary)

**Output:**



**Code:**

#!pip install sumy

from sumy.parsers.plaintext import PlaintextParser

from sumy.nlp.tokenizers import Tokenizer

from sumy.summarizers.lex\_rank import LexRankSummarizer

# Sample text

text = """

Climate change is one of the most pressing issues facing our planet today. It refers to long-term shifts in temperatures and weather patterns, mainly caused by human activities, especially the burning of fossil fuels. These activities release greenhouse gases into the atmosphere, trapping heat and causing the Earth's average temperature to rise. The consequences of climate change are far-reaching and include more frequent and severe weather events, rising sea levels, and disruptions to ecosystems. To address this global challenge, countries and organizations worldwide are working on strategies to reduce greenhouse gas emissions and transition to cleaner energy sources. Individual actions, such as reducing energy consumption and adopting sustainable practices, also play a crucial role in mitigating the effects of climate change.

"""

# Create parser and summarizer

parser = PlaintextParser.from\_string(text, Tokenizer("english"))

summarizer = LexRankSummarizer()

# Get top 2 sentences as summary

summary = summarizer(parser.document, sentences\_count=2)

# Print results

summary\_text = " ".join(str(sentence) for sentence in summary)

print("131 Sakib Tamboli")

print("Extractive text summarization")

print("\nOriginal text length:", len(text))

print("Summary length:", len(summary\_text))

print("Summary:", summary\_text)

**Output:**

