Practical 1

Aim: Write a program to implement Tokenization of text

Program:

```
import nltk
# nltk.download()

data = "My name is Dattaram Santosh Kolte. I am 21 years old. I live in Ghansoli."
# splits the words in the text data
tokens = nltk.word_tokenize(data)
print(tokens)

# splits the sentences in the text data
tokens = nltk.sent_tokenize(data)
print(tokens)
```

Output:

['My', 'name', 'is', 'Dattaram', 'Santosh', 'Kolte', '.', 'I', 'am', '21', 'years', 'old', '.', 'I', 'live', 'in', 'Ghansoli', '.']
['My name is Dattaram Santosh Kolte.', 'I am 21 years old.', 'I live in Ghansoli.']

Practical 2

Aim: Write a program to implement Stop word removal.

Program:

```
from nltk.corpus import stopwords as sw
import nltk
# nltk.download('stopwords')

stopwords = set(sw.words('english'))
# print(stopwords)

data="All work and no play makes jack a dull boy"
tokens = nltk.word_tokenize(data)
data_without_stopwords = []

for word in tokens:
  if word not in stopwords:
    data_without_stopwords.append(word)

print(data_without_stopwords)
```

Output:

['All', 'work', 'play', 'makes', 'jack', 'dull', 'boy']

Practical 3

Aim: Write a program to implement Stemming.

Program:

```
from nltk.stem import PorterStemmer

ps = PorterStemmer()
print(ps.stem("Socks"))
words = ["program", "programs", "programer", "programing", "programers"]

for w in words:
   print(w, " : ", ps.stem(w))
```

Output:

sock
program : program
programs : program
programer : program
programing : program
programers : program

Practical 4

Aim: Write a program to implement Lemmatization.

Program:

```
from nltk.stem import WordNetLemmatizer

lm = WordNetLemmatizer()

print("Socks :",lm.lemmatize("socks"))
print("Better",lm.lemmatize("better",pos="a"))
print("Am",lm.lemmatize("am",pos="v"))

words = ["cats","cacti","radii","feet","speech",'runner']

for w in words:
   print(w,":",lm.lemmatize(w))
```

Output:

Socks : sock
Better good
Am be
cats : cat
cacti : cactus
radii : radius
feet : foot
speech : speech
runner : runner

Practical 5

Aim: Write a program to implement the N-gram model.

Program:

```
import nltk
nltk.download('punkt_tab')

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)
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:

```
Trigram
('The', 'little', 'boy')
('little', 'boy', 'ran')
('boy', 'ran', 'away')

Bigram
('The', 'little')
('little', 'boy')
('boy', 'ran')
('ran', 'away')
```

Practical 6

Aim: Write a program to implement POS tagging.

Program:

```
import spacy

nlp = spacy.load('en_core_web_sm')
doc = nlp("Don't be afraid to give up the good to go for the grea")

POS_count = doc.count_by(spacy.attrs.POS)
print((POS_count))

for k,v in sorted(POS_count.items()):
  print(f"{k}, {doc.vocab[k].text} : {v}")
```

Output:

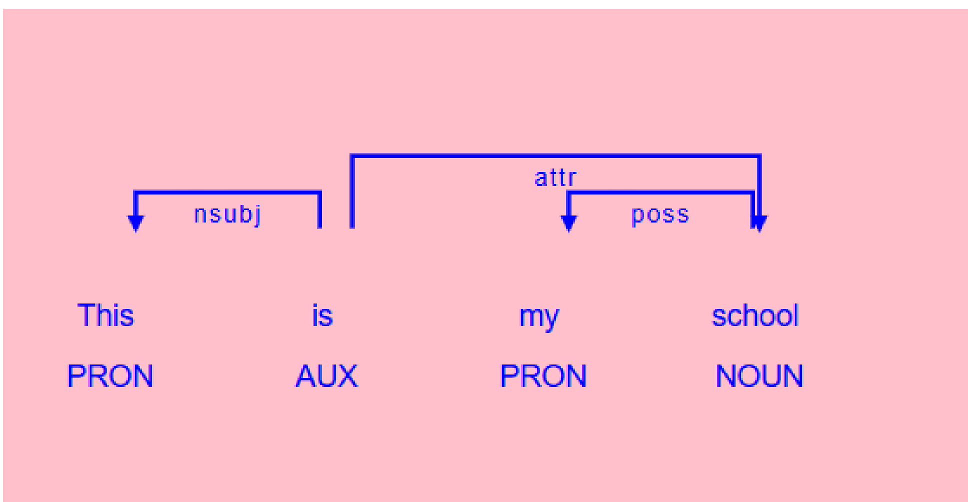
{87: 2, 94: 3, 84: 1, 100: 2, 85: 2, 90: 2, 92: 2}
84, ADJ : 1
85, ADP : 2
87, AUX : 2
90, DET : 2
92, NOUN : 2
94, PART : 3
100, VERB : 2

Program:

```
import spacy
from spacy import displacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("This is my school")

POS_count = doc.count_by(spacy.attrs.POS)
for k,v in sorted(POS_count.items()):
  print(f"{k}, {doc.vocab[k].text} : {v}")

option = {'color':'blue', 'bg':'pink', 'compact':'True', 'distance':100}
displacy.render(doc,style='dep',options=option)
```

Output:

87, AUX : 1
92, NOUN : 1
95, PRON : 2



Program: *by NLTK*

```
import nltk
from nltk import pos_tag, word_tokenize
from nltk.draw import tree

# Sample text
text = "This is my school"

# Tokenization
tokens = word_tokenize(text) #list of tokens
print(tokens)

# Part-of-Speech Tagging
pos_tags = pos_tag(tokens) #returns a list of tupels
print(pos_tags)
# Display POS tags
for word, tag in pos_tags:
   print(f"{word} : {tag}")

# Drawing the POS dependency tree
tree_obj = nltk.Tree('Sentence', [(word, tag) for word, tag in pos_tags])
tree_obj.pretty_print()
```

Output:

```
['This', 'is', 'my', 'school']
[('This', 'DT'), ('is', 'VBZ'), ('my', 'PRP$'), ('school', 'NN')]
This : DT
is : VBZ
my : PRP$
school : NN
      Sentence
   _______|________________
This/DT  is/VBZ  my/PRP$ school/NN
```

Practical 7

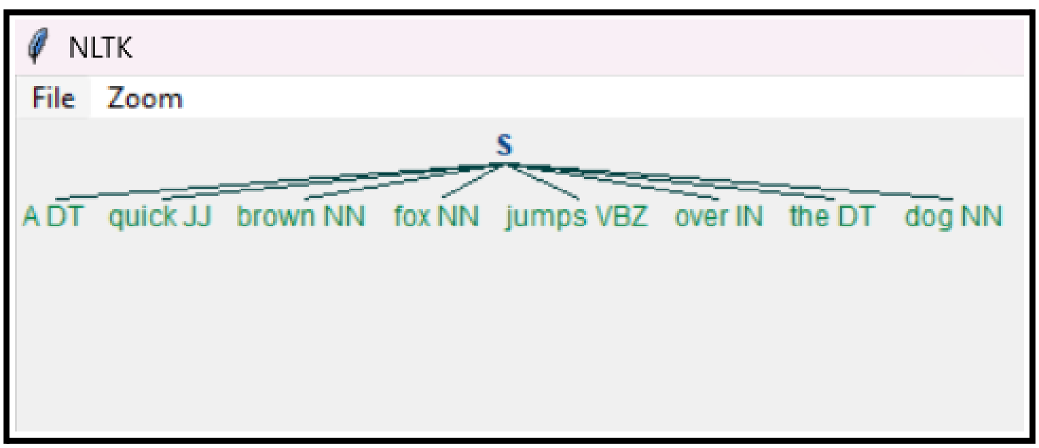
Aim: Write a program to build a custom NER system.

Program: *by NLTK*

```
import nltk
from nltk import word_tokenize,pos_tag,ne_chunk

nltk.download('punkt_tab')
nltk.download('averaged_perceptron_tagger_eng')
nltk.download('maxent_ne_chunker_tab')

#Sample text
text="A quick brown fox jumps over the dog"

#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:

(S A/DT quick/JJ brown/NN fox/NN jumps/VBZ over/IN the/DT dog/NN)



Program: *by Spacy*

```
import spacy

# Load spaCy's English NER model
nlp = spacy.load("en_core_web_sm")

# Sample text
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:

```
Jamshedji Tata -> PERSON
Tata Steel -> ORG
26 August 1907 -> DATE
India -> GPE
English -> LANGUAGE
the First World War -> EVENT
1914-1918 -> CARDINAL
the Tata Group -> ORG
Tata Motors -> ORG
800 -> MONEY
today -> DATE
```

Jamshedji Tata PERSON founded Tata Steel ORG on 26 August 1907 DATE in India GPE .He was very fluent in English LANGUAGE .During the First World War EVENT (1914–1918 CARDINAL), the Tata Group ORG made rapid progress. Tata Motors ORG stock is worth ₹ 800 MONEY today DATE .

Practical 8

Aim: Creating and comparing different text representations.

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

Program:

```
import nltk
import numpy as np
 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:

```
Vocabulary: ['cat', 'dog', 'log', 'mat', 'on', 'sat', 'the']
BoW vectors:
[[1 0 0 1 1 1 2]
 [0 1 1 0 1 1 2]]
```

b)Write a program to create tf_idf text representations.

Program:

```
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:

```
Vocabulary: {'dog', 'sat', 'on', 'the', 'mat', 'cat', 'log'}

IDF vectors:
[1.0, 0.5945348918918356, 0.5945348918918356, 0.5945348918918356, 1.0, 1.0, 1.0]

TF vectors:
[0, 1, 1, 2, 1, 1, 0]

TF vectors:
[1, 1, 1, 2, 0, 0, 1]

TF-IDF vectors:
[[0.         0.59453489 0.59453489 1.18906978 1.         1.
  0.        ]
 [1.         0.59453489 0.59453489 1.18906978 0.         0.
  1.        ]]
```

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

Program:

```
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:

```
{'dog', 'sat', 'on', 'the', 'mat', 'cat', 'log'}
BoW vectors:
[[0 1 1 2 1 1 0]
 [1 1 1 2 0 0 1]]
0.7499999999999999
```

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

Program:

```
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:

Vocabulary: {'dog', 'sat', 'on', 'the', 'mat', 'cat', 'log'}

Cosine Similarity between doc1 (BoW) and doc1 (TF-IDF): 0.9696169252036222
Cosine Similarity between doc2 (BoW) and doc2 (TF-IDF): 0.9696169252036222

Practical 9

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

Program: *Word2vec*

```
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:

Words most similar to 'fox':
at: 0.1607
dogs: 0.1593
barks: 0.1372
night: 0.1230
and: 0.0854

Program: *Glove*

```
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:

```
Vocabulary Size: 19
Sample Co-occurrence Matrix: [[0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 1 0 1 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0]]

Similar words to 'learning': [('subset', 0.5989338705756867), ('is', 0.34292964898919537), ('a', 0.17451031833625497), ('of', 0.1506977386616728), ('deep', 0.09944784813140051)]
```

Practical 10

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

Program:

```
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:

```
              precision    recall  f1-score   support

    negative       0.00      0.00      0.00       1.0
    positive       0.00      0.00      0.00       0.0

    accuracy                           0.00       1.0
   macro avg       0.00      0.00      0.00       1.0
weighted avg       0.00      0.00      0.00       1.0

Prediction for 'The product exceeded my expectations, I'm very satisfied.': positive
```

Practical 11

Aim: Write a Program to Build a sentiment analysis system..

Program:

```
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:

```
text sentiment  pos_score  \
0     I absolutely love this product! It's amazing!  Positive      0.689
1       This is the worst experience I've ever had.  Negative      0.000
2             The movie was okay, nothing special.   Neutral      0.233
3  I'm feeling pretty neutral about the whole sit...  Positive      0.439
4  The customer service was excellent and very he...  Positive      0.541

   neg_score  neu_score  compound_score
0      0.000      0.311          0.8713
1      0.369      0.631         -0.6249
2      0.277      0.490         -0.0920
3      0.000      0.561          0.5719
4      0.000      0.459          0.7955
```

Practical 12

Aim: Write a Program to create a text summarization tool.

Program:

```
#!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 = """ Human communication has come a long way since ancient
civilizations first developed methods to share information. From rudimentary
signals to complex digital conversations, the transformation of how people
connect is a testament to both innovation and necessity.In ancient times, early
humans relied on basic forms of communication such as cave paintings, symbols,
and hand gestures. As societies grew, more structured methods emerged,
including the use of smoke signals and drum beats to transmit simple messages
over long distances. These early techniques were effective in specific contexts but
lacked depth and complexity.The invention of writing marked a significant leap
forward. The earliest forms of written communication appeared around 3100 BCE
in Mesopotamia, where the Sumerians developed cuneiform scripts to record
trade transactions and historical events. Similarly, Egyptian hieroglyphs offered a
more visual approach to documenting information. Writing enabled people to
store knowledge, share stories, and communicate across time and space."""

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:

Device set to use cpu
Abstractive text summarization
Original text length: 1059
Summary length: 332

Summary:
In ancient times, early humans relied on basic forms of communication such as cave paintings, symbols, and hand gestures. As societies grew, more structured methods emerged, including the use of smoke signals and drum beats to transmit simple messages over long distances. The invention of writing marked a significant leap forward.

Program:

```
!pip install sumy
import nltk
nltk.download('punkt_tab')

from sumy.parsers.plaintext import PlaintextParser
from sumy.nlp.tokenizers import Tokenizer
from sumy.summarizers.lex_rank import LexRankSummarizer

# Sample text
text = """Human communication has come a long way since ancient civilizations
first developed methods to share information. From rudimentary signals to
complex digital conversations, the transformation of how people connect is a
testament to both innovation and necessity.In ancient times, early humans relied
on basic forms of communication such as cave paintings, symbols, and hand
gestures. As societies grew, more structured methods emerged, including the use
of smoke signals and drum beats to transmit simple messages over long distances.
These early techniques were effective in specific contexts but lacked depth and
complexity.The invention of writing marked a significant leap forward. The earliest
forms of written communication appeared around 3100 BCE in Mesopotamia,
where the Sumerians developed cuneiform scripts to record trade transactions
and historical events. Similarly, Egyptian hieroglyphs offered a more visual
approach to documenting information. Writing enabled people to store
knowledge, share stories, and communicate across time and space."""

# 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("Extractive text summarization")
print("\nOriginal text length:", len(text))
print("Summary length:", len(summary_text))
print("Summary:", summary_text)
```

Output:

Extractive text summarization

Original text length: 1058
Summary length: 383
Summary: Human communication has come a long way since ancient civilizations first developed methods to share information. From rudimentary signals to complex digital conversations, the transformation of how people connect is a testament to both innovation and necessity.In ancient times, early humans relied on basic forms of communication such as cave paintings, symbols, and hand gestures.