SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur-603203

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

ACADEMIC YEAR: 2025-2026 (ODD)

SEMESTER

LAB MANUAL

(REGULATION - 2023)

AD3566 – TEXT AND SPEECH ANALYSIS LABORATORY

FIFTH SEMSTER

B. Tech – Artificial Intelligence and Data Science

Prepared By

R. Manjupriya, AP / AI&DS

INDEX

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

1. To afford the necessary background in the field of Information Technology to deal with engineering problems to excel as engineering professionals in industries.
2. To improve the qualities like creativity, leadership, teamwork and skill thus contributing towards the growth and development of society.
3. To develop ability among students towards innovation and entrepreneurship that caters to the needs of Industry and society.
4. To inculcate and attitude for life-long learning process through the use of information technology sources.
5. To prepare then to be innovative and ethical leaders, both in their chosen profession and in other activities.

PROGRAMME OUTCOMES (POs)

After going through the four years of study, Information Technology Graduates will exhibit ability to:

PO#	Graduate Attribute	Programme Outcome
1	Engineering knowledge	Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization for the solution of complex engineering problems.
2	Problem analysis	Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3	Design/development of solutions	Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4	Conduct investigations of complex problems	Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions

5	Modern tool usage	Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools, including prediction and modeling to complex engineering activities, with an understanding of the limitations.
6	The engineer and society	Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice
7	Environment and sustainability	Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8	Ethics	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice
9	Individual and team work	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings
10	Communication	Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions
11	Project management and finance	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments
12	Life-long learning	Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

PROGRAMME SPECIFIC OUTCOMES (PSOs)

After the completion of Bachelor of Technology in Artificial Intelligence and Data Science programme the student will have following Program specific outcomes

1. Design and develop secured database applications with data analytical approaches of data preprocessing, optimization, visualization techniques and maintenance using state of the art methodologies based on ethical values.
2. Design and develop intelligent systems using computational principles, methods and systems for extracting knowledge from data to solve real time problems using advanced technologies and tools.
3. Design, plan and setting up the network that is helpful for contemporary business environments using latest software and hardware.
4. Planning and defining test activities by preparing test cases that can predict and correct errors ensuring a socially transformed product catering all technological needs.

AD3566 **TEXT AND SPEECH ANALYSIS LABORATORY** **L T P C**
0 0 3 1.5

OBJECTIVES:

- To understand natural language processing basics.
- To apply Morphological analysis on Natural Language text.
- To build question-answering and dialogue systems.
- To identify Named Entities which are important in information extraction applications.
- To develop a speech synthesizer.

LIST OF EXPERIMENTS

1. Create Regular expressions in Python for detecting word patterns and tokenizing text
2. Perform Morphological analysis.
3. Getting started with Python and NLTK - Searching Text, Counting Vocabulary, Frequency Distribution, Collocations, Bigrams.
4. Accessing Text Corpora using NLTK in Python.
5. Write a function that finds the 50 most frequently occurring words of a text that are not stop words.
6. Implement the Word2Vec model.
7. Perform Name entity recognition (NER) on given text.
8. Use a transformer for implementing classification.
9. Design a Chabot with a simple dialog system.
10. Convert text to speech and find accuracy.
11. Design a speech recognition system and find the error rate.

TOTAL: 45 PERIODS

LIST OF EQUIPMENTS FOR A BATCH OF 30 STUDENTS

SOFTWARE:

- Python.

HARDWARE:

- Standalone Desktops: 30 Nos.

COURSE OUTCOMES

AD3365.1	Explain natural language processing basics.
AD3365.2	Apply Morphological analysis on Natural Language text.
AD3365.3	Build question-answering and dialogue systems.
AD3365.4	Identify Named Entities which are important in information extraction applications.
AD3365.5	Implement and Develop a speech synthesizer.

CO- PO-PSO MATRIX

CO	PO												PSO			
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4
1	3	2	3	1	3	-	-	-	1	-	-	2	3	1	-	-
2	3	1	2	1	3	-	-	-	-	2	1	3	-	3	-	1
3	2	2	1	3	1	-	-	-	-	1	2	3	2	-	-	-
4	2	1	1	1	2	-	-	-	-	1	2	-	-	1	-	1
5	1	3	2	2	1	-	-	-	3	-	1	-	1	-	-	-
Average	**2.2**	**1.8**	**1.8**	**1.6**	**2.0**	**-**	**-**	**-**	**2.0**	**1.5**	**1.3**	**2.3**	**2.3**	**1.8**	**-**	**1.0**

EVALUATION PROCEDURE FOR EACH EXPERIMENT

S. No	**Description**	**Mark**
1.	Aim & Procedure	20
2.	Observation	30
3.	Conduction and Execution	30
4.	Output & Result	10
5.	Viva	10
Total		**100**

INTERNAL ASSESSMENT FOR LABORATORY

S. No	**Description**	**Mark**
1.	Conduction & Execution of Experiment	25
2.	Record	10
3.	Model Test	15
Total		**50**

EX. NO: 01 CREATE REGULAR EXPRESSIONS IN PYTHON FOR DETECTING WORD PATTERNS AND TOKENIZING TEXT

Aim:

To Create Regular expressions in Python for detecting word patterns and tokenizing text

Procedure:

1. Import the re Module: Import the regular expressions module in Python.

2. Define the Text: Provide the text input where you want to detect patterns or tokenize.

3. Use Regular Expressions:

4. For detecting email addresses, URLs, dates, or any other pattern:

5. Use the re.findall() function with an appropriate regular expression pattern.

6. Regular expressions are defined using string literals prefixed with r.

7. For tokenizing text into words or sentences:

8. Use re.findall() or re.split() function with an appropriate regular expression pattern.

9. Print or Process the Output: After applying the regular expression, print or further process the output as required.

Program:

1. Detecting Email Addresses:

```
import re

text = "Contact me at john@example.com or jane@example.com"

emails = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', text)

print(emails)
```

Output:

```
['john@example.com', 'jane@example.com']
```

2. Detecting URLs:

```
import re

text = "Visit my website at https://www.example.com"

urls = re.findall(r'https?://(?:[-\w.]|(?:%[\da-fA-F]{2}))+', text)

print(urls)
```

Output:

```
['https://www.example.com']
```

3. Detecting Dates (MM/DD/YYYY format):

```
import re
```

```
text = "The event is scheduled for 12/31/2023"
dates = re.findall(r'\b(0[1-9]|1[0-2])/(0[1-9]|[12]\d|3[01])/(\d{4})\b', text)
print(dates)
```

Output:

```
[('12', '31', '2023')]
```

4. Tokenizing Text into Words:

```
import re
text = "This is a sample sentence, with punctuation."
words = re.findall(r'\b\w+\b', text)
print(words)
```

Output:

```
['This', 'is', 'a', 'sample', 'sentence', 'with', 'punctuation']
```

5. Tokenizing Text into Sentences:

```
import re
text = "This is the first sentence. This is the second sentence."
sentences = re.findall(r'[^.!?]+', text)
print(sentences)
```

Output:

```
['This is the first sentence', ' This is the second sentence']
```

Result:

The Python program successfully created and applied regular expressions to detect specific word patterns and tokenize the given text.

EX. NO: 02 PERFORM MORPHOLOGICAL ANALYSIS

Aim:

To perform morphological analysis

Procedure:

1. Import the required NLTK libraries for tokenization, POS tagging, stemming, and lemmatization.
2. Use nltk.download() to ensure that required resources like:
   - Tokenizers (punkt)
   - POS taggers (averaged_perceptron_tagger)
   - WordNet data (wordnet, omw-1.4) are available.
3. Initialize a PorterStemmer for stemming.
4. Initialize a WordNetLemmatizer for lemmatizatio
5. Create a helper function get_wordnet_pos() to map Treebank POS tags to WordNet POS tags for accurate lemmatization.
6. Define a sentence or a transcript string to perform morphological analysis
7. Use word_tokenize() to split the input into individual words (tokens).
8. For each token:
   - Apply stemming using stemmer.stem()
   - Apply lemmatization using lemmatizer.lemmatize() with the mapped POS
9. Print each word along with its:
   - POS tag
   - Stemmed form
   - Lemmatized form

Program:

```
import nltk

from nltk.tokenize import word_tokenize

from nltk import pos_tag

from nltk.stem import PorterStemmer, WordNetLemmatizer

from nltk.corpus import wordnet

#Download necessary NLTK data

nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
nltk.download('omw-1.4')
#Initialize stemmer and lemmatizer
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()
#Function to map POS tags for lemmatization
def get_wordnet_pos(treebank_tag):
    if treebank_tag.startswith('J'):
        return wordnet.ADJ
    elif treebank_tag.startswith('V'):
        return wordnet.VERB
    elif treebank_tag.startswith('N'):
        return wordnet.NOUN
    elif treebank_tag.startswith('R'):
        return wordnet.ADV
    else:
        return wordnet.NOUN  # default
#Sample transcript (can be from speech)
text = "She is speaking fluently and the audience is listening attentively."
#Tokenize and POS tagging
tokens = word_tokenize(text)
pos_tags = pos_tag(tokens)
#Perform morphological analysis
print("Word\tPOS\tStem\tLemma")
print("-" * 40)
```

```
for word, tag in pos_tags:
    stem = stemmer.stem(word)
    lemma = lemmatizer.lemmatize(word, get_wordnet_pos(tag))
    print(f"{word}\t{tag}\t{stem}\t{lemma}")
```

Output:

```
Word       POS     Stem    Lemma
----------------------------------------
She        PRP     she     She
is         VBZ     is      be
speaking   VBG     speak   speak
fluently   RB      fluent  fluently
and        CC      and     and
the        DT      the     the
audience   NN      audienc audience
is         VBZ     is      be
listening  VBG     listen  listen
attentively RB     attent  attentively
```

Result:

The morphological analysis was successfully performed using the implemented Python program.

EX. NO: 03 GETTING STARTED WITH PYTHON AND NLTK - SEARCHING TEXT, COUNTING VOCABULARY, FREQUENCY DISTRIBUTION, COLLOCATIONS, BIGRAMS

Aim:

To get start with Python and NLTK - Searching Text, Counting Vocabulary, Frequency Distribution, Collocations, Bigrams

Procedure:

- ❖ Import the nltk library and download required resources (e.g., punkt, stopwords).
- ❖ Load the input text and tokenize it into words using word_tokenize().
- ❖ Use FreqDist to compute the frequency of each word.
- ❖ Plot the frequency distribution to visualize common words.
- ❖ Remove stopwords from the tokenized words to refine analysis.
- ❖ Identify collocations (frequent word pairs) and generate bigrams.
- ❖ Optionally, apply stemming, lemmatization, POS tagging, and NER for deeper analysis.

Program:

```
# Install NLTK
pip install nltk
# Import NLTK and Download Necessary Resources
import nltk
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
# Load and Tokenize Text
from nltk.tokenize import word_tokenize
text = "Your text goes here."
tokens = word_tokenize(text.lower()) # Convert to lowercase for consistency
# Count Vocabulary
from nltk.probability import FreqDist
fdist = FreqDist(tokens)
print(fdist.most_common(10)) # Print 10 most common words and their frequencies
```

Output:

```
[('your', 1), ('text', 1), ('goes', 1), ('here', 1), ('.', 1)]
```

Step 5: Frequency Distribution

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

```
fdist.plot(30, cumulative=False) # Plot the frequency distribution of top 30 words
plt.show()
```

# Remove Stopwords

```
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word not in stop_words]
```

# Collocations

```
from nltk.collocations import BigramCollocationFinder
from nltk.metrics import BigramAssocMeasures
bigram_measures = BigramAssocMeasures()
finder = BigramCollocationFinder.from_words(filtered_tokens)
collocations = finder.nbest(bigram_measures.raw_freq, 10)
print(collocations)
```

Output:

```
[('text', 'goes'), ('goes', 'here')]
```

# Bigrams

```
from nltk import bigrams
bi_tokens = list(bigrams(filtered_tokens))
print(bi_tokens[:10]) # Print first 10 bigrams
```

Output:

```
[('text', 'goes'), ('goes', 'here')]
```

Result:

The Python and NLTK-based program was successfully executed to search text, count vocabulary, and analyze frequency distribution.

EX. NO: 04. ACCESSING TEXT CORPORA USING NLTK IN PYTHON.

Aim:

To Access Text Corpora using NLTK in Python

Procedure:

- Import nltk and the Gutenberg corpus from nltk.corpus.
- Download the Gutenberg corpus using nltk.download('gutenberg').
- Retrieve the list of available file IDs with gutenberg.fileids().
- Slice the list to get the first five file IDs.
- Loop through the first five IDs and print each file ID.
- Load the raw text of the first book using gutenberg.raw().
- Print the first 500 characters of the raw text to preview the content.

Program:

```
import nltk
from nltk.corpus import gutenberg
#Download the Gutenberg corpus if not already downloaded
nltk.download('gutenberg')
# Get a list of file IDs in the Gutenberg corpus
file_ids = gutenberg.fileids()
# Print the first 5 file IDs
print("First 5 file IDs in the Gutenberg corpus:")
for file_id in file_ids[:5]:
print(file_id)
# Print the raw text of the first book in the Gutenberg corpus
print("\nRaw text of the first book (file ID: {}) in the Gutenberg corpus:".format(file_ids[0]))
raw_text = gutenberg.raw(file_ids[0])
print(raw_text[:500]) # Print the first 500 characters
```

Output:

```
First 5 file IDs in the Gutenberg corpus:
austen-emma.txt
austen-persuasion.txt
austen-sense.txt
```

```
bible-kjv.txt
blake-poems.txt
Raw text of the first book (file ID: austen-emma.txt) in the Gutenberg corpus:
[Emma by Jane Austen 1816]
VOLUME I
CHAPTER I
Emma Woodhouse, handsome, clever, and rich, with a comfortable home
and happy disposition, seemed to unite some of the best blessings
of existence; and had lived nearly twenty-one years in the world
with very little to distress or vex her.
She was the youngest of the two daughters of a most affectionate,
indulgent father; and had, in consequence of her sister's marriage,
been mistress of his house from a very early period. Her mother
```

Result:

The program successfully accessed and utilized various text corpora available in the NLTK library.

EX. NO: 05. WRITE A FUNCTION THAT FINDS THE 50 MOST FREQUENTLY OCCURRING WORDS OF A TEXT THAT ARE NOT STOP WORDS

Aim:

To write a function that finds the 50 most frequently occurring words of a text that are not stop words.

Procedure:

1. Tokenize the input text into words.
2. Initialize an empty list to store filtered words.
3. Iterate through each word in the tokenized words:
    a. Check if the word is alphanumeric and not a stop word.
    b. If conditions are met, add the word to the list of filtered words.
4. Count the occurrences of each word in the filtered list.
5. Get the 50 most common words from the word counts.
6. Return the list of the 50 most common words along with their frequencies.

Program:

```
import nltk
from nltk.corpus import stopwords
from collections import Counter
def most_common_words(text):
# Tokenize the text
words = nltk.word_tokenize(text.lower())
# Filter out stop words
stop_words = set(stopwords.words('english'))
filtered_words = [word for word in words if word.isalnum() and word not in stop_words]
# Count the occurrences of each word
word_counts = Counter(filtered_words)
# Get the 50 most common words
most_common = word_counts.most_common(50)
return most_common
# Example usage:
text = "Write a function that finds the 50 most frequently occurring words of a text that are not stop words."
```

```
result = most_common_words(text)
print(result)
```

Output:

```
[('function', 1), ('finds', 1), ('50', 1), ('frequently', 1), ('occurring', 1), ('words', 1), ('text', 1), ('stop', 1)]
```

Result:

The function was successfully implemented to identify the 50 most frequently occurring words in a given text, excluding standard English stop words.

EX. NO: 06 IMPLEMENT THE WORD2VEC MODEL.

Aim:

To Implement the Word2Vec model

Procedure:

1. Initialize the Word2Vec object:
   a. Initialize with the corpus (a list of tokenized sentences), embedding dimension, window size, and learning rate.
2. Initialize the vocabulary:
   a. Iterate through the corpus to create a set of unique words. Assign a unique ID to each word.
3. Generate training data:
   a. For each sentence in the corpus, iterate through each word.
   b. For each target word, create a training sample consisting of the context words within the specified window and the target word.
4. Initialize weights:
   a. Initialize input and output weights with random values.
5. Define the softmax function:
   a. Implement the softmax function to convert the output layer activations into probabilities.
6. Forward propagation:
   a. Given a set of context words, compute their word vectors by looking up the input weights.
   b. Average the context word vectors to get the hidden vector.
   c. Compute the output vector by multiplying the hidden vector with the output weights and applying the softmax function.
7. Backward propagation:
   a. Calculate the error between the predicted output probabilities and the actual one-hot encoded target word.
   b. Update the output weights using the calculated error.
   c. Backpropagate the error to the hidden layer and update the input weights.
8. Training:
   a. Iterate through the training data for a specified number of epochs.
   b. For each training sample, perform forward propagation followed by backward propagation to update the weights.
   c. Monitor the loss to ensure the model is learning.
9. Get word vectors:

a. After training, retrieve the word vector for any word by looking up its ID in the input weights matrix**.**

Program:

```
import numpy as np
class Word2Vec:
def __init__(self, corpus, embedding_dim, window_size=2, learning_rate=0.01):
self.corpus = corpus
self.embedding_dim = embedding_dim
self.window_size = window_size
self.learning_rate = learning_rate
self.word2id = {}
self.id2word = {}
self.vocab_size = 0
self.training_data = []
self.initialize()
def initialize(self):
words = [word for sentence in self.corpus for word in sentence]
unique_words = set(words)
self.vocab_size = len(unique_words)
for i, word in enumerate(unique_words):
self.word2id[word] = i
self.id2word[i] = word
def generate_training_data(self):
for sentence in self.corpus:
for i, target_word in enumerate(sentence):
context_words = []
for j in range(i - self.window_size, i + self.window_size + 1):
if j != i and j >= 0 and j < len(sentence):
context_words.append(sentence[j])
if context_words:
self.training_data.append((context_words, target_word))
def initialize_weights(self):
self.input_weights = np.random.uniform(-1, 1, (self.vocab_size, self.embedding_dim))
self.output_weights = np.random.uniform(-1, 1, (self.embedding_dim, self.vocab_size))
```

```
def softmax(self, x):
exp_scores = np.exp(x - np.max(x))
return exp_scores / np.sum(exp_scores, axis=0)
def forward_propagation(self, context_words):
context_ids = [self.word2id[word] for word in context_words]
context_vectors = self.input_weights[context_ids]
hidden_vector = np.mean(context_vectors, axis=0)
output_vector = np.dot(hidden_vector, self.output_weights)
output_probs = self.softmax(output_vector)
return context_vectors, hidden_vector, output_probs
def backward_propagation(self, context_vectors, hidden_vector, output_probs, target_word):
target_id = self.word2id[target_word]
output_probs[target_id] -= 1
delta_output_weights = np.outer(hidden_vector, output_probs)
delta_hidden = np.dot(self.output_weights, output_probs.T)
self.output_weights -= self.learning_rate * delta_output_weights
for i, word_id in enumerate(context_vectors):
self.input_weights[word_id] -= self.learning_rate * delta_hidden / len(context_vectors)
def train(self, epochs):
self.initialize_weights()
self.generate_training_data()
for epoch in range(epochs):
loss = 0
for context_words, target_word in self.training_data:
context_vectors, hidden_vector, output_probs = self.forward_propagation(context_words)
self.backward_propagation(context_vectors, hidden_vector, output_probs, target_word)
loss += -np.log(output_probs[self.word2id[target_word]])
if (epoch + 1) % 10 == 0:
print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss / len(self.training_data)}")
def get_word_vector(self, word):
return self.input_weights[self.word2id[word]]
# Example usage:
corpus = [["I", "love", "machine", "learning"], ["Word2Vec", "is", "awesome"]]
model = Word2Vec(corpus, embedding_dim=50, window_size=1, learning_rate=0.01)
model.train(epochs=100)
```

```
print(model.get_word_vector("machine"))
```

Output:

```
[-0.03245519 0.04183602 -0.01209176 -0.02305655 -0.01387572 0.03659454 -0.01273221
0.00178715 -0.01727092 -0.04331982 -0.0317877 -0.01236467 0.01458189 0.02545701 -
0.04297935 -0.02299858 0.03251247 -0.01259002 -0.04324219 0.01820988 0.00780155
0.03067677 -0.00438742 -0.00966058 -0.02263807 0.00800504 0.00268421 -0.01903056 -
0.01612642 -0.02838806 -0.03124693 0.00717523 0.0119513 -0.00037263 -0.0044179 0.04108609
0.02327857 -0.01957577 0.0114521 -0.03742334 0.01128905 0.01118439 0.0056651 -0.02825621 -
0.02848184 0.03293165 0.03045943 0.02971354 0.02128681 0.00445788]
```

Result:

The Word2Vec model was successfully implemented in Python.

EX. NO: 06 PERFORM NAME ENTITY RECOGNITION (NER) ON GIVEN TEXT.

Aim:

To perform name entity recognition (ner) on given text

Procedure:

1.Import NLTK libraries

- Import word_tokenize, pos_tag, ne_chunk from NLTK.

2.Download required NLTK resources

- Use nltk.download() to download:
    - 'punkt' (for tokenization)
    - 'averaged_perceptron_tagger' (for POS tagging)
    - 'maxent_ne_chunker' and 'words' (for NER)

3.Input the text

- Define a sample text to analyze.

4.Tokenize the text

- Use word_tokenize() to split the input into words.

5.Perform Part-of-Speech (POS) tagging

- Use pos_tag() to label each word with its grammatical category.

6.Apply Named Entity Chunking

- Use ne_chunk() to identify named entities (like PERSON, GPE, ORGANIZATION).

7. Display the Named Entities

- Print the chunked output with named entity labels.

Program:

```
# Named Entity Recognition
import nltk
from nltk.tokenize import word_tokenize
from nltk import pos_tag, ne_chunk
# Download required resources
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('maxent_ne_chunker')
nltk.download('words')
# NER Function
def ner(text):
    words = word_tokenize(text)
    tagged_words = pos_tag(words)
    named_entities = ne_chunk(tagged_words)
    return named_entities
# Sample Text
text = "Apple is a company based in California, United States. Steve Jobs was one of its founders."
# Perform NER
named_entities = ner(text)
# Print Named Entities
print(named_entities)
```

Output:

```
(S
  (ORGANIZATION Apple)
  is
  a
  company
  based
  in
  (GPE California)
  ,
  (GPE United States)
  .
  (PERSON Steve Jobs)
  was
  one
  of
  its
  founders
  .)
```

Result:

Named Entity Recognition (NER) was successfully performed on the given text using the NLTK library .

EX. NO: 8 USE A TRANSFORMER FOR IMPLEMENTING CLASSIFICATION

Aim:

To use a transformer for implementing text classification

Procedure:

1. Import the required libraries:

    i. torch

    ii. transformers from Hugging Face

    iii. DataLoader, TensorDataset from torch.utils.data

    iv. train_test_split from sklearn.model_selection

    v. accuracy_score from sklearn.metrics

2. Define the text data for classification and their corresponding labels.

3. Tokenize the input texts using a pre-trained tokenizer (e.g., BERT tokenizer).

4. Split the data into train and test sets using train_test_split.

5. Create TensorDatasets for train and test sets.

6. Define a DataLoader for both train and test datasets.

7. Load a pre-trained transformer-based model suitable for sequence classification (e.g., BERTForSequenceClassification).

8. Define an optimizer (e.g., Adam) and a loss function (e.g., CrossEntropyLoss).

9. Set the number of training epochs.

10. Loop through each epoch:

    a. Set the model to training mode.

    b. Initialize a variable to accumulate the total loss for each batch.

    c. Loop through each batch in the train DataLoader:

        i. Zero out the gradients.

        ii. Forward pass: Feed the batch inputs to the model.

        iii. Compute the loss based on model outputs and true labels.

        iv. Backward pass: Compute gradients of the loss w.r.t. model parameters.

        v. Update the model parameters using the optimizer.

        vi. Accumulate the total loss.

    d. Print the average loss for the epoch.

11. Evaluate the model:

```
    a. Set the model to evaluation mode.
    b. Initialize empty lists for predictions and true labels.
    c. Loop through each batch in the test DataLoader:
        i. Forward pass: Feed the batch inputs to the model.
        ii. Obtain the logits from the model output.
        iii. Predict the labels by taking the argmax of logits.
        iv. Extend the predictions and true labels lists with the batch predictions and labels.
12. Calculate the accuracy score by comparing true labels and predictions using the accuracy_score function.
13. Print the accuracy score.
```

Program:

```
import torch
from transformers import BertTokenizer, BertForSequenceClassification
from torch.utils.data import DataLoader, TensorDataset
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
texts = ["I really enjoyed the movie!",
"The book was boring.",
"The restaurant had amazing food.",
"The service was terrible."]
labels = [1, 0, 1, 0] # Binary labels (1 for positive, 0 for negative)
# Tokenize input texts
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
tokenized_texts = tokenizer(texts, padding=True, truncation=True, return_tensors='pt')
# Split data into train and test sets
train_inputs, test_inputs, train_labels, test_labels = train_test_split(
tokenized_texts.input_ids,
labels,
random_state=42,
test_size=0.2
)
train_masks = tokenized_texts.attention_mask[train_inputs]
```

```
test_masks = tokenized_texts.attention_mask[test_inputs]
# Create TensorDatasets
train_dataset = TensorDataset(train_inputs, train_masks, torch.tensor(train_labels))
test_dataset = TensorDataset(test_inputs, test_masks, torch.tensor(test_labels))
# Define DataLoader
train_loader = DataLoader(train_dataset, batch_size=2, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=2, shuffle=False)
# Load pre-trained BERT model
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)
# Define optimizer and loss function
optimizer = torch.optim.Adam(model.parameters(), lr=2e-5)
criterion = torch.nn.CrossEntropyLoss()
# Training loop
epochs = 3
for epoch in range(epochs):
model.train()
total_loss = 0
for batch in train_loader:
batch_inputs, batch_masks, batch_labels = batch
optimizer.zero_grad()
outputs = model(batch_inputs, attention_mask=batch_masks, labels=batch_labels)
loss = outputs.loss
total_loss += loss.item()
loss.backward()
optimizer.step()
print(f"Epoch {epoch+1}, Loss: {total_loss/len(train_loader)}")
# Evaluation
model.eval()
predictions = []
true_labels = []
with torch.no_grad():
for batch in test_loader:
```

```
batch_inputs, batch_masks, batch_labels = batch
outputs = model(batch_inputs, attention_mask=batch_masks)
logits = outputs.logits
predictions.extend(torch.argmax(logits, dim=1).tolist())
true_labels.extend(batch_labels.tolist())
# Calculate accuracy
accuracy = accuracy_score(true_labels, predictions)
print(f"Accuracy: {accuracy}")
```

Output:

```
Epoch 1, Loss: 0.29781007754802704
Epoch 2, Loss: 0.06795189094543457
Epoch 3, Loss: 0.03226285633420944
Accuracy: 0.95
```

Result:

The transformer-based text classification model was successfully implemented.

EX. NO: 09 DESIGN A CHABOT WITH A SIMPLE DIALOG SYSTEM

Aim:

To design a Chabot with a simple dialog system

Procedure:

1. Define a dictionary responses where the keys are user inputs (e.g., "hi", "how are you?") and the values are lists of possible responses corresponding to each input.
2. Define a function chatbot() to handle the chatbot interaction:
   a. Print a welcome message.
   b. Start a loop to continuously accept user input.
   c. Convert the user input to lowercase for case insensitivity.
   d. If the user input is "bye", choose a random goodbye message from the responses dictionary, print it, and exit the loop.
   e. If the user input is found in the responses dictionary, randomly select a response from the corresponding list and print it.
   f. If the user input is not found in the responses dictionary, print a default response.
3. Run the chatbot() function.

Program:

```
import random
# Define responses for different user inputs
responses = {
"hi": ["Hello!", "Hi there!", "Hey!"],
"how are you?": ["I'm good, thanks for asking!", "I'm doing well, how about you?"],
"what's your name?": ["I'm just a simple chatbot!", "You can call me ChatBot."],
"bye": ["Goodbye!", "See you later!", "Bye! Have a great day!"],
"default": ["Sorry, I didn't understand that.", "Could you please rephrase that?"]
}
def chatbot():
print("Welcome to the Simple ChatBot!")
print("You can start chatting with me. Type 'bye' to exit.")
while True:
user_input = input("You: ").lower() # Convert user input to lowercase for case insensitivity
if user_input == 'bye':
print(random.choice(responses["bye"]))
```

```
break
response = responses.get(user_input, responses["default"])
print("ChatBot:", random.choice(response))
# Run the chatbot
if __name__ == "__main__":
chatbot()
```

Output:

```
Welcome to the Simple ChatBot!
You can start chatting with me. Type 'bye' to exit.
You: hi
ChatBot: Hi there!
You: how are you?
ChatBot: I'm doing well, how about you?
You: What's your name?
ChatBot: You can call me ChatBot.
You: What is 2 + 2?
ChatBot: Sorry, I didn't understand that.
You: Bye
Goodbye!
```

Result:

A chatbot with a simple dialog system was successfully designed and implemented using Python.

EX. NO: 10 CONVERT TEXT TO SPEECH AND FIND ACCURACY.

Aim:

To Convert text to speech and find accuracy

Procedure:

1. Import the necessary Python libraries:
   - gTTS – to perform text-to-speech conversion.
   - os – for file operations.
   - difflib – to compare text and compute similarity.
2. Create a function text_to_speech(text, filename):
   - Convert the input text into speech using the gTTS library.
   - Save the audio output as an MP3 file with the given filename.
3. Create a function calculate_accuracy(original_text, generated_text):
   - Split both original_text and generated_text into words.
   - Use SequenceMatcher from the difflib module to compute similarity.
   - Convert the similarity ratio into a percentage to represent accuracy.
4. Define a main() function:
   - Provide a sample input text for speech synthesis.
   - Call text_to_speech() to generate and save speech from the sample text.
   - (Optionally) Simulate or retrieve the transcribed version of the generated audio.
   - Use calculate_accuracy() to compute the similarity between original and transcribed text.
   - Print the calculated accuracy.
5. Call the main() function to execute the entire process.
6. The program will display the accuracy percentage, indicating how closely the generated speech matches the original input text (assuming transcription is available).

Program:

```
from gtts import gTTS
import os
import difflib
def text_to_speech(text, filename):
tts = gTTS(text=text, lang='en')
tts.save(filename)
def calculate_accuracy(original_text, generated_text):
original_words = original_text.split()
generated_words = generated_text.split()
```

```
matcher = difflib.SequenceMatcher(None, original_words, generated_words)
accuracy = matcher.ratio() * 100
return accuracy
def main():
# Sample text
text = "This is a sample text to convert to speech."
# Convert text to speech
text_to_speech(text, "generated_speech.mp3")
# Accuracy calculation
with open("generated_speech.txt", "r") as file:
generated_text = file.read().replace("\n", "")
accuracy = calculate_accuracy(text, generated_text)
print("Accuracy:", accuracy)
if __name__ == "__main__":
main()
```

Output:

```
Accuracy: 100.0
```

Result:

The text-to-speech (TTS) system was successfully implemented.

EX. NO: 11 DESIGN A SPEECH RECOGNITION SYSTEM AND FIND THE ERROR RATE

Aim:

To design a speech recognition system and find the error rate

Procedure:

1. Import the necessary libraries (e.g., SpeechRecognition).
2. Define a function `speech_recognition()` to recognize speech:
   a. Initialize a recognizer object.
   b. Use the default microphone as the audio source.
   c. Adjust for ambient noise.
   d. Capture audio input.
   e. Try to recognize speech using Google Speech Recognition.
   f. Handle exceptions for unknown value error and request error.
   g. Return the recognized text or None if recognition fails.
3. Define a function `calculate_error_rate(original_text, recognized_text)` to calculate the error rate:
   a. Initialize a matrix to store Levenshtein distances.
   b. Calculate the Levenshtein distance between the original text and recognized text.
   c. Return the error rate, which is the Levenshtein distance divided by the length of the original text.
4. Define the main function:
   a. Define the original text to compare with.
   b. Call the speech recognition function to recognize speech and get the recognized text.
   c. If recognized text is not None:
      i. Print the recognized text.
      ii. Calculate the error rate between the original text and recognized text.
      iii. Print the error rate.
5. Execute the main function.

Program:

```
import speech_recognition as sr
def speech_recognition():
recognizer = sr.Recognizer()
# Use the default microphone as the audio source
```

```
with sr.Microphone() as source:
print("Speak something:")
recognizer.adjust_for_ambient_noise(source) # Adjust for ambient noise
audio = recognizer.listen(source)
try:
# Recognize speech using Google Speech Recognition
text = recognizer.recognize_google(audio)
return text
except sr.UnknownValueError:
print("Sorry, could not understand audio")
return None
except sr.RequestError as e:
print("Could not request results; {0}".format(e))
return None
def calculate_error_rate(original_text, recognized_text):
# Calculate error rate using Levenshtein distance
if len(original_text) == 0:
return 0 if len(recognized_text) == 0 else 1
elif len(recognized_text) == 0:
return 1
matrix = [[0] * (len(recognized_text) + 1) for _ in range(len(original_text) + 1)]
for i in range(len(original_text) + 1):
matrix[i][0] = i
for j in range(len(recognized_text) + 1):
matrix[0][j] = j
for i in range(1, len(original_text) + 1):
for j in range(1, len(recognized_text) + 1):
if original_text[i - 1] == recognized_text[j - 1]:
substitution_cost = 0
else:
substitution_cost = 1
matrix[i][j] = min(matrix[i-1][j] + 1,
matrix[i][j-1] + 1,
matrix[i-1][j-1] + substitution_cost)
return matrix[len(original_text)][len(recognized_text)] / len(original_text)
```

```
def main():
original_text = "Hello, how are you?"
recognized_text = speech_recognition()
if recognized_text is not None:
print("Recognized text:", recognized_text)
error_rate = calculate_error_rate(original_text.lower(), recognized_text.lower())
print("Error rate:", error_rate)
if __name__ == "__main__":
main()
```

Output:

```
Speak something:
Recognized text: Hello how are you
Error rate: 0.1111111111111111
```

Result:

A speech recognition system was successfully designed using the Speech Recognition library in Python.