**Prac 1**

import pandas as pd

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

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.naive\_bayes import MultinomialNB

from sklearn.metrics import accuracy\_score, classification\_report

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

import string

# Download necessary NLTK resources

nltk.download('stopwords')

nltk.download('punkt')

# Sample dataset

data = {

"text": [

"I love this product! It's amazing and works perfectly.",

"The quality is terrible, I'm very disappointed.",

"Good value for money, happy with my purchase.",

"Awful experience, would not recommend it to anyone.",

"Decent, but could be improved in some areas.",

],

"label": ["positive", "negative", "positive", "negative", "neutral"]

}

# Create DataFrame

df = pd.DataFrame(data)

print("Original Dataset:\n", df)

# Text Preprocessing Function

def preprocess\_text(text):

text = text.lower() # Convert to lowercase

tokens = word\_tokenize(text) # Tokenize text

tokens = [word for word in tokens if word not in string.punctuation] # Remove punctuation

stop\_words = set(stopwords.words('english'))

tokens = [word for word in tokens if word not in stop\_words] # Remove stopwords

return ' '.join(tokens)

# Apply preprocessing

df['cleaned\_text'] = df['text'].apply(preprocess\_text)

# Convert text data to numerical representation

vectorizer = CountVectorizer()

X = vectorizer.fit\_transform(df['cleaned\_text'])

y = df['label']

# Split into training and testing sets

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

# Train Naive Bayes model

model = MultinomialNB()

model.fit(X\_train, y\_train)

# Make predictions

y\_pred = model.predict(X\_test)

# Evaluate model performance

print("\nModel Accuracy:", accuracy\_score(y\_test, y\_pred))

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred))

# Sentiment Prediction Function

def predict\_sentiment(text):

processed\_text = preprocess\_text(text) # Preprocess input text

vectorized\_text = vectorizer.transform([processed\_text]) # Transform text

return model.predict(vectorized\_text)[0] # Predict sentiment

# Test with new input

new\_text = "This is the best product I've ever bought!"

print("\nPredicted Sentiment:", predict\_sentiment(new\_text))

**Prac 2**

import torch

from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer

model\_name = "microsoft/DialoGPT-medium"

tokenizer = AutoTokenizer.from\_pretrained(model\_name)

model = AutoModelForCausalLM.from\_pretrained(model\_name)

# Function to generate chatbot responses

def chatbot\_response(prompt, chat\_history\_ids=None):

input\_ids = tokenizer.encode(prompt + tokenizer.eos\_token, return\_tensors="pt")

# Append new user input to the chat history

bot\_input\_ids = (

torch.cat([chat\_history\_ids, input\_ids], dim=-1) if chat\_history\_ids is not None else input\_ids

)

# Generate response

chat\_history\_ids = model.generate(

bot\_input\_ids,

max\_length=1000,

pad\_token\_id=tokenizer.eos\_token\_id,

top\_k=50,

top\_p=0.95,

temperature=0.7,

do\_sample=True,

)

# Decode and return response

response = tokenizer.decode(chat\_history\_ids[:, bot\_input\_ids.shape[-1]:][0], skip\_special\_tokens=True)

return response, chat\_history\_ids

# Start chatbot interaction

print("Chatbot: Hello! I am a chatbot. How can I help you today?")

chat\_history = None # To maintain conversation context

while True:

user\_input = input("You: ")

# Exit condition

if user\_input.lower() in ["exit", "quit", "bye"]:

print("Chatbot: Goodbye! Have a great day!")

break

# Generate chatbot response

response, chat\_history = chatbot\_response(user\_input, chat\_history)

print(f"Chatbot: {response}")

Prac 3

# Aim: Implementing a computer vision project, such as object detection or image segmentation.

from ultralytics import YOLO

import cv2

import matplotlib.pyplot as plt

# Load the YOLOv8 model

model = YOLO(r'C:\Users\student\Downloads\yolov8n.pt') # Use raw string for the path

# Path to the image

image\_path = r'C:\Users\student\Desktop\lily.jpg' # Use raw string for the path

# Read and process the image

image = cv2.imread(image\_path)

image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB) # Convert BGR to RGB

# Make predictions on the image

results = model.predict(image, conf=0.5)

# Annotate the image with predictions

annotated\_image = results[0].plot()

# Display the annotated image

plt.imshow(annotated\_image)

plt.axis('off') # Hide axes

plt.show()

Prac 4

# Aim: Generate realistic image using Generative AI

import tensorflow as tf

from tensorflow.keras import layers

import numpy as np

import matplotlib.pyplot as plt

# Define the generator model

def build\_generator():

model = tf.keras.Sequential([

layers.Dense(128, activation='relu', input\_shape=(100,)), # Fully connected layer

layers.Reshape((4, 4, 8)), # Reshape to 4x4x8

layers.Conv2DTranspose(64, (4, 4), strides=(2, 2), padding='same', activation='relu'), # Upsample to 8x8x64

layers.Conv2DTranspose(1, (4, 4), strides=(7, 7), padding='same', activation='sigmoid') # Final image (28x28x1)

])

return model

# Generate random noise

generator = build\_generator()

noise = tf.random.normal([1, 100]) # 1 sample, noise of shape (100,)

generated\_image = generator(noise) # Generate image from noise

# Visualize the generated image

plt.imshow(generated\_image[0, :, :, 0], cmap='gray') # Display first channel (grayscale)

plt.axis('off') # Hide axes for better visualization

plt.show()

Prac 5

# Aim: Building a deep learning model for time series forecasting or anomaly detection.

# Step-01: Install required libraries

# Uncomment the line below to install the required libraries

# !pip install numpy pandas matplotlib tensorflow

# Step-02: Code

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense, Dropout

# Generate synthetic time series data (Sine wave with noise)

time = np.arange(0, 100, 0.1)

data = np.sin(time) + np.random.normal(0, 0.1, len(time))

# Plot the synthetic data

plt.plot(time, data)

plt.title("Synthetic Time Series Data")

plt.xlabel("Time")

plt.ylabel("Value")

plt.show()

# Convert to a DataFrame

df = pd.DataFrame(data, columns=["value"])

# Normalize the data

scaler = MinMaxScaler(feature\_range=(0, 1))

df["value"] = scaler.fit\_transform(df[["value"]])

# Function to create sequences for LSTM input

def create\_sequences(data, sequence\_length):

sequences = []

labels = []

for i in range(len(data) - sequence\_length):

seq = data[i:i + sequence\_length]

label = data[i + sequence\_length]

sequences.append(seq)

labels.append(label)

return np.array(sequences), np.array(labels)

# Define sequence length

sequence\_length = 50

data\_values = df["value"].values

# Create sequences

X, y = create\_sequences(data\_values, sequence\_length)

# Reshape for LSTM input (samples, timesteps, features)

X = X.reshape(X.shape[0], X.shape[1], 1)

# Define the LSTM model

model = Sequential([

LSTM(50, activation="relu", input\_shape=(sequence\_length, 1), return\_sequences=True),

Dropout(0.2),

LSTM(50, activation="relu", return\_sequences=False),

Dropout(0.2),

Dense(1)

])

# Compile the model

model.compile(optimizer="adam", loss="mean\_squared\_error")

# Train the model

history = model.fit(X, y, epochs=20, batch\_size=32, validation\_split=0.2)

# Plot training and validation loss

plt.plot(history.history["loss"], label="Training Loss")

plt.plot(history.history["val\_loss"], label="Validation Loss")

plt.legend()

plt.title("Training and Validation Loss")

plt.show()

prac 6

# Aim: Use Python libraries such as GPT-2 or textgenrnn to train generative models on a corpus of text data and generate new text based on the patterns it has learned.

from transformers import GPT2LMHeadModel, GPT2Tokenizer

# Load the GPT-2 model and tokenizer

model\_name = "gpt2"

tokenizer = GPT2Tokenizer.from\_pretrained(model\_name)

model = GPT2LMHeadModel.from\_pretrained(model\_name)

# Define the text generation function

def generate\_text(prompt, max\_length=50, temperature=0.7, top\_p=0.9):

input\_ids = tokenizer.encode(prompt, return\_tensors="pt")

# Generate text using the GPT-2 model

output = model.generate(

input\_ids,

max\_length=max\_length,

temperature=temperature,

top\_p=top\_p,

do\_sample=True,

pad\_token\_id=tokenizer.eos\_token\_id

)

# Decode the output and return generated text

return tokenizer.decode(output[0], skip\_special\_tokens=True)

# Example prompt

prompt = "My College is at"

# Generate text based on the prompt

generated\_text = generate\_text(prompt, max\_length=100)

# Print the generated text

print("Generated Text:\n")

print(generated\_text)

prac 7

# Python code: Using RNN with LSTM for text classification

import tensorflow as tf

import numpy as np

from tensorflow.keras import layers, models

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

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

from sklearn.metrics import accuracy\_score

# Sample data (For a full implementation, replace this with a larger dataset)

texts = [

"I love this movie", "This movie is terrible",

"Amazing plot and great acting", "I hate this movie",

"The movie was okay, not great", "Fantastic experience, very enjoyable",

"Not worth watching", "I would watch it again, highly recommend",

"I dislike the plot but the acting was good", "A masterpiece in cinema"

]

labels = [1, 0, 1, 0, 0, 1, 0, 1, 0, 1] # 1: Positive, 0: Negative

# Tokenize the text data

tokenizer = Tokenizer(num\_words=10000)

tokenizer.fit\_on\_texts(texts)

X = tokenizer.texts\_to\_sequences(texts)

# Pad sequences to ensure they are of the same length

X = pad\_sequences(X, maxlen=10)

# Train-test split

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

# Convert to numpy arrays (if not already)

X\_train = np.array(X\_train)

X\_test = np.array(X\_test)

y\_train = np.array(y\_train)

y\_test = np.array(y\_test)

# Define the LSTM-based RNN model

model\_rnn = models.Sequential([

layers.Embedding(input\_dim=10000, output\_dim=64, input\_length=10),

layers.LSTM(64, return\_sequences=False),

layers.Dense(64, activation='relu'),

layers.Dense(1, activation='sigmoid') # Binary classification (positive/negative)

])

# Compile the model

model\_rnn.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Train the RNN model (LSTM)

model\_rnn.fit(X\_train, y\_train, epochs=5, batch\_size=2, validation\_data=(X\_test, y\_test))

# Evaluate the model

y\_pred\_rnn = (model\_rnn.predict(X\_test) > 0.5).astype(int)

# Print the accuracy of the model

print("RNN (LSTM) Accuracy:", accuracy\_score(y\_test, y\_pred\_rnn))

prac 8

import numpy as np

import random

import matplotlib.pyplot as plt

# Define the environment size (grid dimensions)

grid\_size = (5, 5) # 5x5 grid

goal\_state = (4, 4) # goal is at the bottom-right corner

start\_state = (0, 0) # start at the top-left corner

# Define the actions

actions = ['up', 'down', 'left', 'right']

action\_map = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}

# Q-learning parameters

learning\_rate = 0.1 # Alpha

discount\_factor = 0.9 # Gamma

epsilon = 0.1 # Epsilon for epsilon-greedy policy

num\_episodes = 1000 # Number of episodes for training

max\_steps\_per\_episode = 100 # Maximum steps per episode

# Initialize Q-table (state-action value table)

q\_table = np.zeros((grid\_size[0], grid\_size[1], len(actions)))

# Reward function: -1 for each step, +100 for reaching the goal

def reward\_function(state):

if state == goal\_state:

return 100 # reward for reaching the goal

return -1 # penalty for each step

# Choose action based on epsilon-greedy policy

def choose\_action(state):

if random.uniform(0, 1) < epsilon: # Exploration: random action

return random.choice(actions)

else: # Exploitation: choose the best action based on Q-table

state\_x, state\_y = state

q\_values = q\_table[state\_x, state\_y, :]

max\_q\_value = np.max(q\_values)

max\_actions = [actions[i] for i in range(len(actions)) if q\_values[i] == max\_q\_value]

return random.choice(max\_actions)

# Take action and get the next state

def take\_action(state, action):

state\_x, state\_y = state

move = action\_map[action]

new\_x = max(0, min(grid\_size[0] - 1, state\_x + move[0])) # Ensure we don't go out of bounds

new\_y = max(0, min(grid\_size[1] - 1, state\_y + move[1])) # Ensure we don't go out of bounds

return (new\_x, new\_y)

# Q-learning algorithm

def train\_q\_learning():

for episode in range(num\_episodes):

state = start\_state

total\_reward = 0

for step in range(max\_steps\_per\_episode):

action = choose\_action(state)

next\_state = take\_action(state, action)

reward = reward\_function(next\_state)

# Update Q-value using the Q-learning formula

state\_x, state\_y = state

next\_state\_x, next\_state\_y = next\_state

action\_index = actions.index(action)

# Bellman equation: Q(s, a) = Q(s, a) + alpha \* [reward + gamma \* max(Q(s', a')) - Q(s, a)]

max\_future\_q = np.max(q\_table[next\_state\_x, next\_state\_y, :])

q\_table[state\_x, state\_y, action\_index] += learning\_rate \* (reward + discount\_factor \* max\_future\_q - q\_table[state\_x, state\_y, action\_index])

state = next\_state

total\_reward += reward

# If the agent reaches the goal, break the episode

if state == goal\_state:

break

if (episode + 1) % 100 == 0:

print(f"Episode {episode + 1} completed")

# Visualization of learned policy (after training)

def visualize\_policy():

policy\_grid = np.full(grid\_size, '', dtype=object)

for x in range(grid\_size[0]):

for y in range(grid\_size[1]):

best\_action\_index = np.argmax(q\_table[x, y, :])

best\_action = actions[best\_action\_index]

policy\_grid[x, y] = best\_action[0].upper() # Show the first letter of the best action

# Plotting the policy grid

plt.figure(figsize=(6, 6))

plt.imshow(np.zeros(grid\_size), cmap="Blues", interpolation='none')

for x in range(grid\_size[0]):

for y in range(grid\_size[1]):

plt.text(y, x, policy\_grid[x, y], ha='center', va='center', color='black', fontsize=12)

plt.title("Learned Policy (Q-learning)")

plt.xticks(range(grid\_size[1]))

plt.yticks(range(grid\_size[0]))

plt.gca().invert\_yaxis()

plt.show()

# Train the agent using Q-learning

train\_q\_learning()

# Visualize the learned policy

visualize\_policy()