**PRACTICAL JOURNAL**

in

**Advanced Artificial Intelligence**

**&**

**Machine Learning**

Submitted to

**Laxman Devram Sonawane College, Kalyan (W) 421301**

*in partial fulfillment for the award of the degree of*

**Master of Science in Information Technology**

(Affiliated to Mumbai University)

*Submitted by*

**TANIA TAZIM JAMDAR**

Under the guidance of

**Mrs. Sabina Ansari (Advanced Artificial Intelligence)**

**&**

**Dr. Priyanka Pawar ( Machine Learning )**

Department of Information Technology

Kalyan, Maharashtra

Academic Year 2024-25

**PRACTICAL JOURNAL**

in

**Advance Artificial Intelligence**

Submitted to

**Laxman Devram Sonawane College, Kalyan (W) 421301**

in partial fulfilment for the award of the degree of

 **Master of Science in Information Technology**

(Affiliated to Mumbai University)

*Submitted by*

**TANIA TAZIM JAMDAR**

Under the guidance of

**Mrs. Sabina Ansari**

Department of Information Technology

Kalyan, Maharashtra

Academic Year 2024-25



The Kalyan Wholesale Merchants Education Society’s

**Laxman Devram Sonawane College,**

**Kalyan (W) 421301**

**Department of Information Technology**

**Masters of Science – Part II**

**Certificate**

This is to certify that **Mr. Tania Tazim Jamdar**, Seat number**\_\_\_\_\_\_\_\_\_\_\_\_\_**, studying in Masters of Science in Information Technology Part II , Semester II has satisfactorily completed the practical of “**Advanced Artificial Intelligence** ” as prescribed by University of Mumbai, during the academic year 2024-25.

Subject In-charge Coordinator In-charge ExternalExaminer

College Seal

**ADVANCED ARTIFICIAL INTELLIGENCE**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **SR No.** | **Practical** | **Sign** |
| 1 | Implementing advanced deep learning algorithms such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs) using Python libraries like TensorFlow or PyTorch. |  |
| 2 | Building a natural language processing (NLP) model for sentiment analysis or text classification. |  |
| 3 | Creating a chatbot using advanced techniques like transformer models |  |
| 4 | Developing a recommendation system using collaborative filtering or deep learning approaches. |  |
| 5 | Implementing a computer vision project, such as object detection or image segmentation |  |
| 6 | Training a generative adversarial network (GAN) for generating realistic images |  |
| 7 | Applying reinforcement learning algorithms to solve complex decision-making problems. |  |
| 8 | Utilizing transfer learning to improve model performance on limited datasets. |  |
| 9 | Building a deep learning model for time series forecasting or anomaly detection. |  |
| 10 | Implementing a machine learning pipeline for automated feature engineering and model selection. |  |
| 11 | Using advanced optimization techniques like evolutionary algorithms or Bayesian optimization for hyperparameter tuning |  |
| 12 | Deploying a machine learning model in a production environment using containerization and cloud services. |  |
| 13 | 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. |  |
| 14 | Experiment with neural networks like GANs (Generative Adversarial Networks) using Python libraries like TensorFlow or PyTorch to generate new images based on a dataset of images. |  |

**Practical 1**

**Aim :** Implementing advanced deep learning algorithms such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs) using Python libraries like TensorFlow or PyTorch

**Code :**

import tensorflow as tf

from tensorflow.keras import layers, models

import matplotlib.pyplot as plt

**# Load and preprocess the CIFAR-10 dataset**

(x\_train, y\_train), (x\_test, y\_test) = tf.keras.datasets.cifar10.load\_data()

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0  # Normalize pixel values to [0, 1]

**# Build the CNN model**

model = models.Sequential([

    layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(32, 32, 3)),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.Flatten(),

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

    layers.Dense(10, activation='softmax')  # 10 classes for CIFAR-10

])

**# Compile the model**

model.compile(optimizer='adam',

              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

**# Train the model**

history = model.fit(x\_train, y\_train, epochs=10, validation\_data=(x\_test, y\_test))

**# Evaluate the model**

test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=2)

print(f'\nTest accuracy: {test\_acc}')

**# Plot training & validation accuracy values**

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.title('Model accuracy')

plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend()

plt.show()

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

import matplotlib.pyplot as plt

**# Define transformations for the training and testing data**

transform = transforms.Compose([

    transforms.ToTensor(),

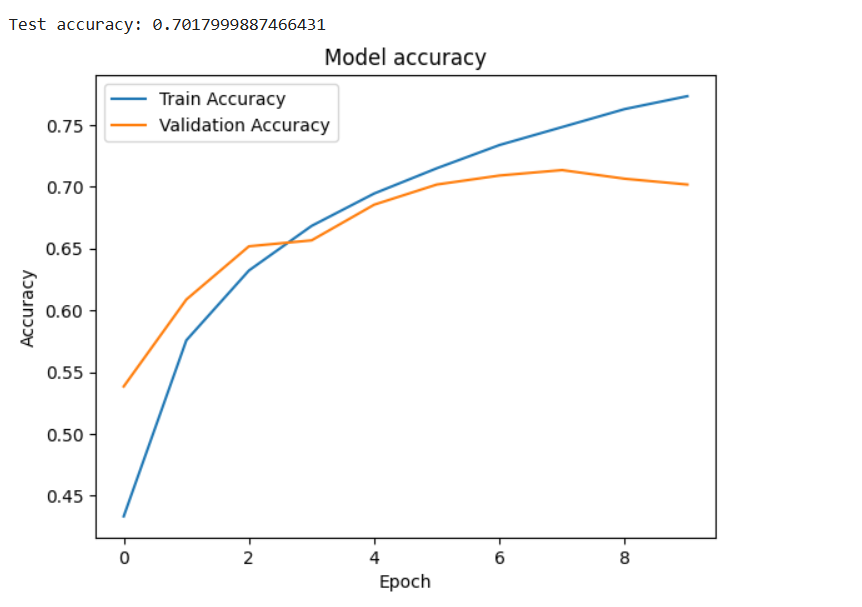
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),

])

**# Load the CIFAR-10 dataset**

trainset = torchvision

**OUTPUT :**

****

**Practical 2**

**Aim :** Building a natural language processing (NLP) model for sentiment analysis or text classification.

**Code :**

!pip install tensorflow matplotlib

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.preprocessing.text import Tokenizer

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

from tensorflow.keras.datasets import imdb

import matplotlib.pyplot as plt

**# Step 1: Load the IMDB dataset**

num\_words = 10000  # Use the top 10,000 words in the vocabulary

(x\_train, y\_train), (x\_test, y\_test) = imdb.load\_data(num\_words=num\_words)

**# Step 2: Explore the dataset**

print(f"Number of training samples: {len(x\_train)}")

print(f"Number of test samples: {len(x\_test)}")

print(f"Sample review (tokenized): {x\_train[0]}")

print(f"Label (0 = negative, 1 = positive): {y\_train[0]}")

**# Step 3: Decode a sample review**

word\_index = imdb.get\_word\_index()

reverse\_word\_index = {value: key for key, value in word\_index.items()}

decoded\_review = " ".join([reverse\_word\_index.get(i - 3, "?") for i in x\_train[0]])

print(f"Decoded review: {decoded\_review}")

**# Step 4: Pad sequences**

maxlen = 200  # Limit each review to 200 words

x\_train = pad\_sequences(x\_train, maxlen=maxlen)

x\_test = pad\_sequences(x\_test, maxlen=maxlen)

**# Step 5: Define the model**

model = models.Sequential([

    layers.Embedding(input\_dim=num\_words, output\_dim=32, input\_length=maxlen),

    layers.LSTM(32),  # Use an LSTM layer for capturing sequential dependencies

    layers.Dense(1, activation='sigmoid')  # Output layer for binary classification

])

**# Step 6: Compile the model**

model.compile(optimizer='adam',

              loss='binary\_crossentropy',

              metrics=['accuracy'])

**# Step 7: Display the model architecture**

model.summary()

**# Step 8: Train the model**

history = model.fit(x\_train, y\_train, epochs=5, batch\_size=64, validation\_split=0.2)

**# Step 9: Evaluate the model**

test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=2)

print(f"Test Accuracy: {test\_acc}")

**# Step 10: Plot training history**

plt.figure(figsize=(12, 4))

**# Accuracy plot**

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()

plt.title('Model Accuracy')

**# Loss plot**

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'], label='Train Loss')

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

plt.xlabel('Epoch')

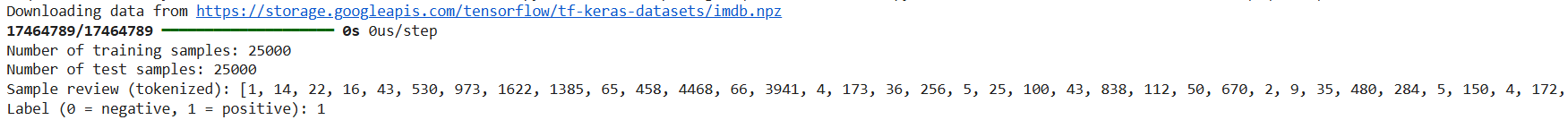
plt.ylabel('Loss')

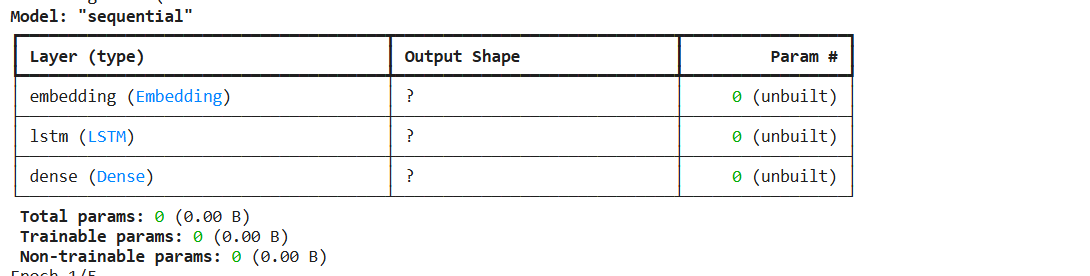
plt.legend()

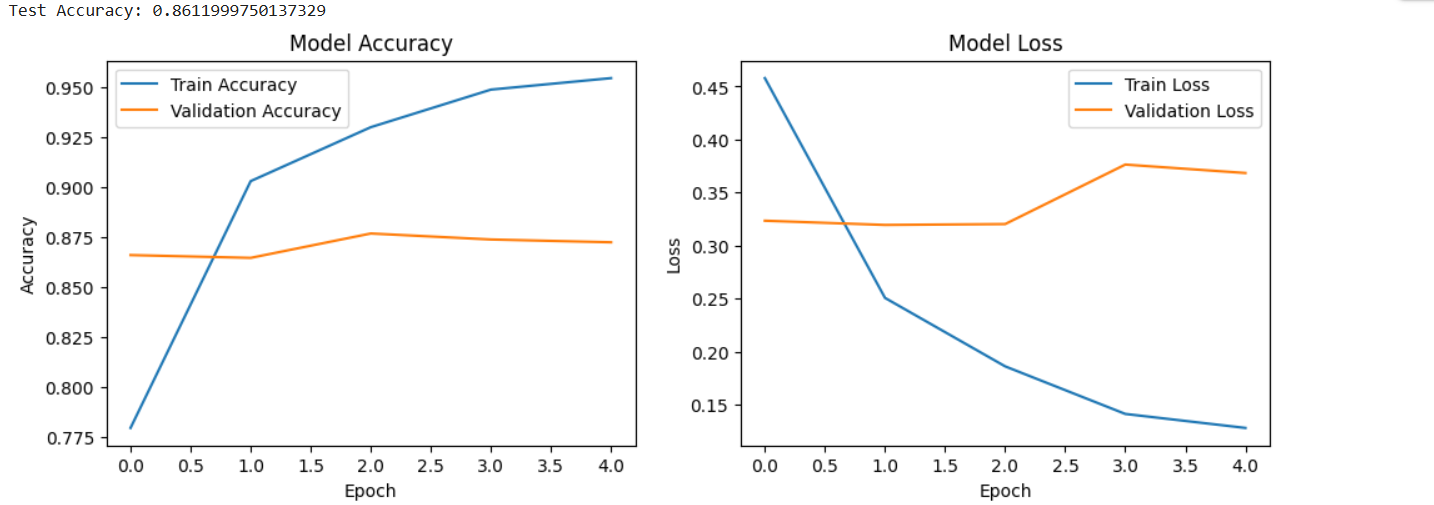
plt.title('Model Loss')

plt.show()

**OUTPUT :**

****

****

****

**Practical 3**

**Aim :** Creating a chatbot using advanced techniques like transformer models.

**Code :**

!pip install transformers torch

from transformers import AutoModelForCausalLM, AutoTokenizer

import torch

**# Step 1: Load Pre-trained Model and Tokenizer**

print("Loading the DialoGPT model...")

model\_name = "microsoft/DialoGPT-medium"

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

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

**# Step 2: Initialize Chat History**

chat\_history\_ids = None

step = 0

**# Step 3: Chat with the User**

print("Chatbot is ready! Type 'exit' to end the chat.\n")

while True:

    # User input

    user\_input = input("You: ")

    if user\_input.lower() == 'exit':

        print("Chatbot: Goodbye!")

        break

**# Encode the user input and add it to the chat history**

    new\_input\_ids = tokenizer.encode(user\_input + tokenizer.eos\_token, return\_tensors='pt')

    chat\_history\_ids = torch.cat([chat\_history\_ids, new\_input\_ids], dim=-1) if step > 0 else new\_input\_ids

**# Generate a response using the model**

    response\_ids = model.generate(chat\_history\_ids, max\_length=1000, pad\_token\_id=tokenizer.eos\_token\_id)

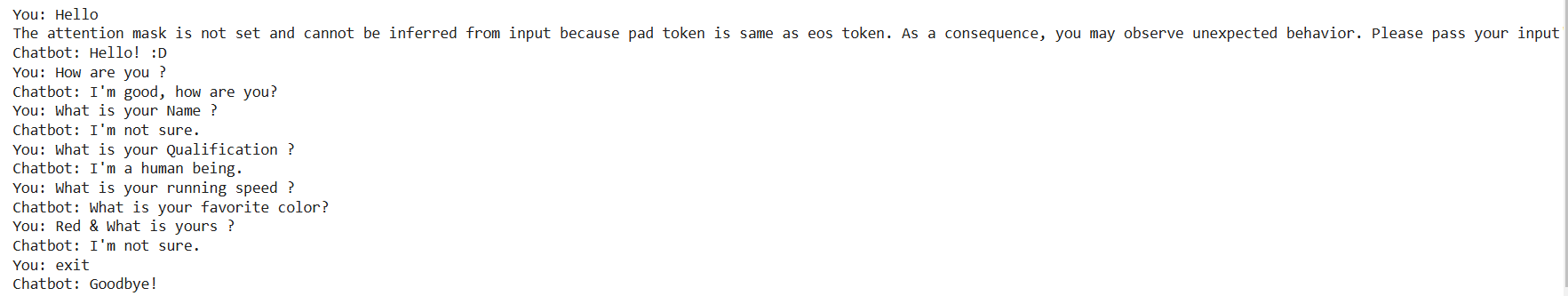
    response = tokenizer.decode(response\_ids[:, chat\_history\_ids.shape[-1]:][0], skip\_special\_tokens=True)

**# Display the response**

    print(f"Chatbot: {response}")

    step += 1

**Output :**

****

**Practical 4**

**Aim** : Developing a recommendation system using collaborative filtering or deep learning approaches.

**Code :**

**Step 1: Install Required Libraries**

Run the following command to install the necessary libraries:

pip install tensorflow numpy pandas matplotlib

**Step 2: Download the Dataset**

Download the MovieLens 100K dataset from grouplens.org/datasets/movielens. Extract the dataset into a folder.

Alternatively, the code below assumes that the u.data file is in the ml-100k folder.

**Step 3: Python Code for the Recommendation System**

import pandas as pd

import numpy as np

import tensorflow as tf

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

import matplotlib.pyplot as plt

**# Step 1: Load and preprocess the dataset**

file\_path = "ml-100k/u.data"

column\_names = ['user\_id', 'item\_id', 'rating', 'timestamp']

data = pd.read\_csv(file\_path, sep='\t', names=column\_names)

**# Normalize user IDs and item IDs to start from 0**

data['user\_id'] -= 1

data['item\_id'] -= 1

**# Extract the number of users and items**

num\_users = data['user\_id'].max() + 1

num\_items = data['item\_id'].max() + 1

print(f"Number of users: {num\_users}, Number of items: {num\_items}")

**# Step 2: Split data into training and testing**

train\_data, test\_data = train\_test\_split(data, test\_size=0.2, random\_state=42)

**# Step 3: Create TensorFlow datasets**

def create\_tf\_dataset(df):

users = tf.constant(df['user\_id'].values, dtype=tf.int32)

items = tf.constant(df['item\_id'].values, dtype=tf.int32)

ratings = tf.constant(df['rating'].values, dtype=tf.float32)

return tf.data.Dataset.from\_tensor\_slices(((users, items), ratings)).shuffle(1024).batch(32)

train\_dataset = create\_tf\_dataset(train\_data)

test\_dataset = create\_tf\_dataset(test\_data)

**# Step 4: Define the Recommendation Model**

class MatrixFactorizationModel(tf.keras.Model):

def \_\_init\_\_(self, num\_users, num\_items, embedding\_dim=50):

super().\_\_init\_\_()

self.user\_embedding = tf.keras.layers.Embedding(num\_users, embedding\_dim)

self.item\_embedding = tf.keras.layers.Embedding(num\_items, embedding\_dim)

def call(self, inputs):

user\_vector = self.user\_embedding(inputs[0])

item\_vector = self.item\_embedding(inputs[1])

dot\_product = tf.reduce\_sum(user\_vector \* item\_vector, axis=1)

return dot\_product

model = MatrixFactorizationModel(num\_users, num\_items)

**# Step 5: Compile the model**

model.compile(optimizer=tf.keras.optimizers.Adam(learning\_rate=0.01), loss='mse', metrics=['mae'])

**# Step 6: Train the model**

history = model.fit(train\_dataset, validation\_data=test\_dataset, epochs=10)

**# Step 7: Evaluate the model**

test\_loss, test\_mae = model.evaluate(test\_dataset)

print(f"Test Loss: {test\_loss:.4f}, Test MAE: {test\_mae:.4f}")

**# Step 8: Plot the training history**

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)

plt.plot(history.history['loss'], label='Train Loss')

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

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend()

plt.title('Loss over Epochs')

plt.subplot(1, 2, 2)

plt.plot(history.history['mae'], label='Train MAE')

plt.plot(history.history['val\_mae'], label='Validation MAE')

plt.xlabel('Epoch')

plt.ylabel('Mean Absolute Error')

plt.legend()

plt.title('MAE over Epochs')

plt.show()

**# Step 9: Make recommendations**

def recommend(user\_id, top\_k=5):

user\_vector = tf.constant([user\_id] \* num\_items, dtype=tf.int32)

item\_vector = tf.constant(list(range(num\_items)), dtype=tf.int32)

predictions = model.predict((user\_vector, item\_vector))

top\_items = np.argsort(-predictions)[:top\_k]

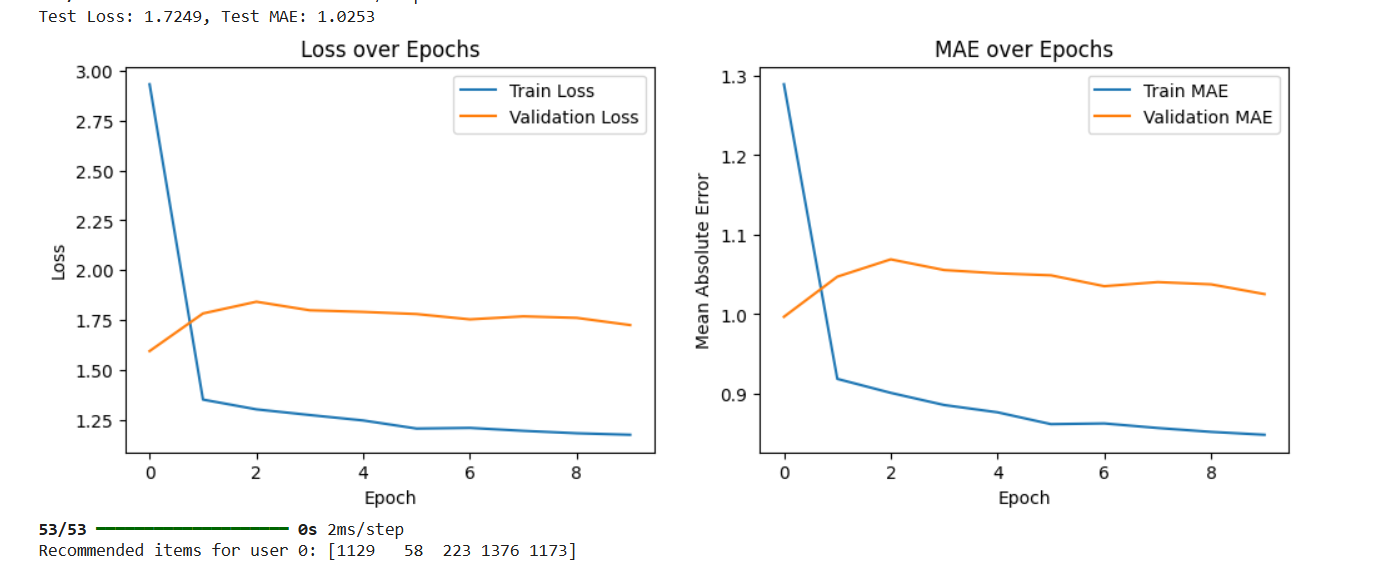
return top\_items

user\_id = 0 # Example user

recommended\_items = recommend(user\_id)

print(f"Recommended items for user {user\_id}: {recommended\_items}"

**Output :**

****

**Practical 5**

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

**Code :**

!pip install torch torchvision numpy matplotlib opencv-python ultralytics

import cv2

import numpy as np

import matplotlib.pyplot as plt

from ultralytics import YOLO  # YOLOv5 library from ultralytics

**# Step 1: Load the YOLOv5 Model**

print("Loading YOLOv5 model...")

model = YOLO("yolov5s.pt")  # Use the small version of YOLOv5 pre-trained on COCO dataset

**# Step 2: Load an Image for Object Detection**

image\_path = "/example.jpg"  # Replace with your image file path

image = cv2.imread(image\_path)

image\_rgb = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

**# Step 3: Perform Object Detection**

print("Performing object detection...")

results = model.predict(image\_rgb)

**# Step 4: Visualize Results**

annotated\_image = results[0].plot()  # Annotated image with bounding boxes and labels

plt.figure(figsize=(10, 10))

plt.imshow(cv2.cvtColor(annotated\_image, cv2.COLOR\_BGR2RGB))

plt.axis("off")

plt.title("Object Detection Results")

plt.show()

**# Step 5: Save the Annotated Image**

output\_path = "output.jpg"

cv2.imwrite(output\_path, annotated\_image)

print(f"Annotated image saved to: {output\_path}")

**# Step 6: Print Detected Objects**

print("Detected objects:")

for box in results[0].boxes.data.tolist():

    x1, y1, x2, y2, conf, cls = box

    print(f"Class: {results[0].names[int(cls)]}, Confidence: {conf:.2f}, Coordinates: ({x1:.2f}, {y1:.2f}), ({x2:.2f}, {y2:.2f})")

**Output :**

****

**Practical 6**

**Aim :** Training a generative adversarial network (GAN) for generating realistic images

**Code :**

!pip install torch torchvision matplotlib numpy

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import datasets, transforms

from torch.utils.data import DataLoader

import matplotlib.pyplot as plt

**# Step 1: Define Generator and Discriminator**

class Generator(nn.Module):

    def \_\_init\_\_(self, noise\_dim, img\_dim):

        super(Generator, self).\_\_init\_\_()

        self.model = nn.Sequential(

            nn.Linear(noise\_dim, 128),

            nn.ReLU(),

            nn.Linear(128, 256),

            nn.ReLU(),

            nn.Linear(256, 512),

            nn.ReLU(),

            nn.Linear(512, img\_dim),

            nn.Tanh(),

        )

    def forward(self, x):

        return self.model(x)

class Discriminator(nn.Module):

    def \_\_init\_\_(self, img\_dim):

        super(Discriminator, self).\_\_init\_\_()

        self.model = nn.Sequential(

            nn.Linear(img\_dim, 512),

            nn.LeakyReLU(0.2),

            nn.Linear(512, 256),

            nn.LeakyReLU(0.2),

            nn.Linear(256, 1),

            nn.Sigmoid(),

        )

    def forward(self, x):

        return self.model(x)

**# Step 2: Define Constants and Hyperparameters**

device = "cuda" if torch.cuda.is\_available() else "cpu"

img\_size = 28

img\_dim = img\_size \* img\_size

noise\_dim = 100

batch\_size = 64

epochs = 50

lr = 0.0002

**# Step 3: Prepare the MNIST Dataset**

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

dataset = datasets.MNIST(root="data", train=True, transform=transform, download=True)

dataloader = DataLoader(dataset, batch\_size=batch\_size, shuffle=True)

**# Step 4: Initialize Models, Loss, and Optimizers**

generator = Generator(noise\_dim, img\_dim).to(device)

discriminator = Discriminator(img\_dim).to(device)

criterion = nn.BCELoss()

optimizer\_g = optim.Adam(generator.parameters(), lr=lr)

optimizer\_d = optim.Adam(discriminator.parameters(), lr=lr)

**# Step 5: Training Loop**

for epoch in range(epochs):

    for real\_images, \_ in dataloader:

        real\_images = real\_images.view(-1, img\_dim).to(device)

        batch\_size = real\_images.size(0)

**# Labels for real and fake images**

        real\_labels = torch.ones(batch\_size, 1).to(device)

        fake\_labels = torch.zeros(batch\_size, 1).to(device)

**# Train Discriminator**

        noise = torch.randn(batch\_size, noise\_dim).to(device)

        fake\_images = generator(noise)

        real\_preds = discriminator(real\_images)

        fake\_preds = discriminator(fake\_images.detach())

        loss\_d\_real = criterion(real\_preds, real\_labels)

        loss\_d\_fake = criterion(fake\_preds, fake\_labels)

        loss\_d = (loss\_d\_real + loss\_d\_fake) / 2

        optimizer\_d.zero\_grad()

        loss\_d.backward()

        optimizer\_d.step()

**# Train Generator**

        noise = torch.randn(batch\_size, noise\_dim).to(device)

        fake\_images = generator(noise)

        fake\_preds = discriminator(fake\_images)

        loss\_g = criterion(fake\_preds, real\_labels)

        optimizer\_g.zero\_grad()

        loss\_g.backward()

        optimizer\_g.step()

**# Print progress**

    print(f"Epoch [{epoch+1}/{epochs}] | Loss D: {loss\_d:.4f} | Loss G: {loss\_g:.4f}")

**# Save generated samples every 10 epochs**

    if (epoch + 1) % 10 == 0:

        noise = torch.randn(16, noise\_dim).to(device)

        generated\_images = generator(noise).view(-1, 1, img\_size, img\_size).cpu().detach()

        plt.figure(figsize=(8, 8))

        for i in range(16):

            plt.subplot(4, 4, i + 1)

            plt.imshow(generated\_images[i].squeeze(), cmap="gray")

            plt.axis("off")

        plt.tight\_layout()

        plt.savefig(f"generated\_images\_epoch\_{epoch+1}.png")

        plt.close()

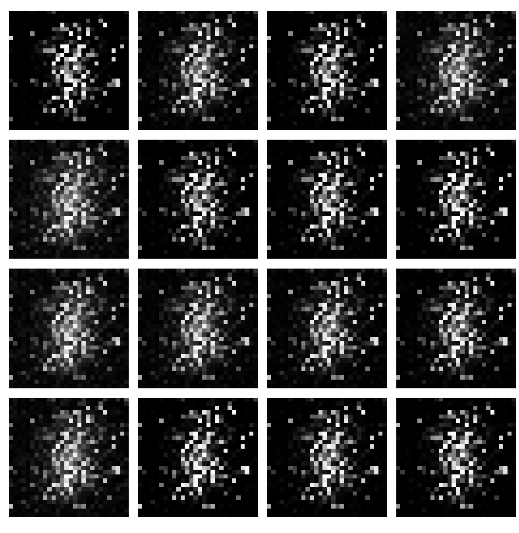
**# Step 6: Save the Generator Model**

torch.save(generator.state\_dict(), "gan\_generator.pth")

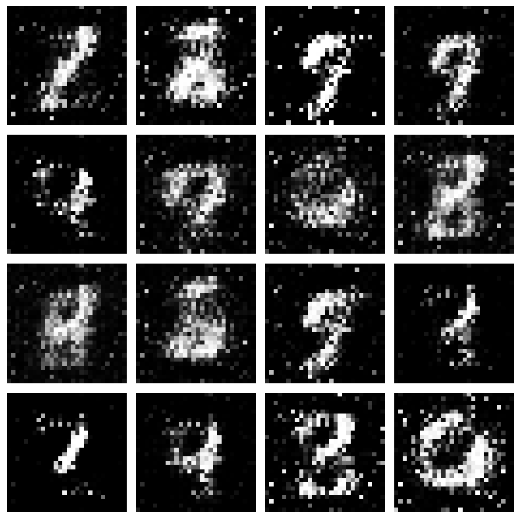
print("Generator model saved as gan\_generator.pth")

**Output :**

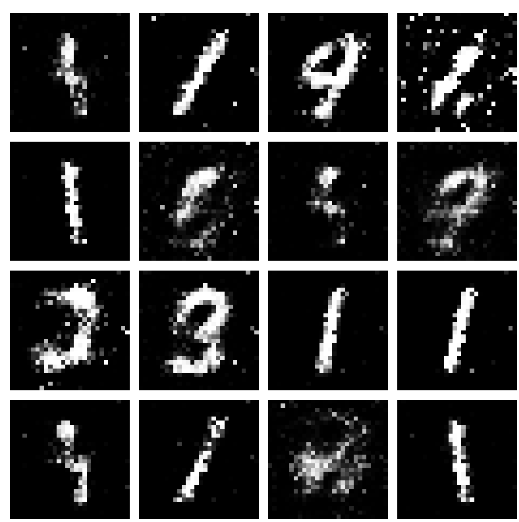
**Generate after 10 Epoch**

****

**Generate after 20 Epoch**

****

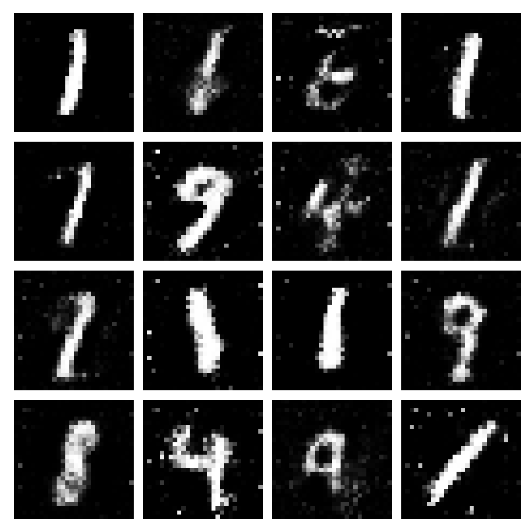
**Generate 30 Epoch**

****

**Generate after 40 Epoch**

****

**Generate after 50 Epoch**

****

**Practical 7**

**Aim :** Applying reinforcement learning algorithms to solve complex decision-making problems.

**Code :**

**Step 1: Install Required Libraries**

Run the following commands in your terminal to install the necessary libraries:

!pip install numpy gym matplotlib

**Step 2: Python Code for Q-Learning**

Save the following code as q\_learning\_cartpole.py.

import gym

import numpy as np

import matplotlib.pyplot as plt

**# Step 1: Initialize Environment and Parameters**

env = gym.make("CartPole-v1")

n\_actions = env.action\_space.n # Number of actions (2: left, right)

n\_states = 20 # Discretize continuous state space

episodes = 500 # Number of episodes

learning\_rate = 0.1 # Learning rate (alpha)

discount\_factor = 0.99 # Discount factor (gamma)

epsilon = 1.0 # Exploration rate

epsilon\_decay = 0.995 # Decay factor for epsilon

min\_epsilon = 0.01 # Minimum epsilon

**# Step 2: Helper Functions for Discretizing States**

def discretize\_state(state, state\_bins):

return tuple(np.digitize(state[i], state\_bins[i]) for i in range(len(state)))

def create\_bins(n\_states, env):

state\_bins = []

for i in range(env.observation\_space.shape[0]):

low, high = env.observation\_space.low[i], env.observation\_space.high[i]

bins = np.linspace(low, high, n\_states - 1)

state\_bins.append(bins)

return state\_bins

**# Step 3: Initialize Q-Table and State Bins**

state\_bins = create\_bins(n\_states, env)

q\_table = np.zeros((n\_states,) \* len(env.observation\_space.shape) + (n\_actions,))

**# Step 4: Training Loop**

rewards = []

for episode in range(episodes):

state = discretize\_state(env.reset()[0], state\_bins)

total\_reward = 0

done = False

while not done:

# Epsilon-greedy action selection

if np.random.rand() < epsilon:

action = np.random.choice(n\_actions) # Explore

else:

action = np.argmax(q\_table[state]) # Exploit

**# Take action and observe results**

next\_state\_raw, reward, done, \_, \_ = env.step(action)

next\_state = discretize\_state(next\_state\_raw, state\_bins)

total\_reward += reward

**# Q-Learning update**

q\_table[state][action] += learning\_rate \* (

reward + discount\_factor \* np.max(q\_table[next\_state]) - q\_table[state][action]

)

state = next\_state

**# Decay epsilon**

epsilon = max(min\_epsilon, epsilon \* epsilon\_decay)

rewards.append(total\_reward)

print(f"Episode {episode + 1}/{episodes}, Reward: {total\_reward}, Epsilon: {epsilon:.3f}")

**# Step 5: Plot Rewards**

plt.plot(rewards)

plt.title("Total Rewards Over Episodes")

plt.xlabel("Episode")

plt.ylabel("Total Reward")

plt.show()

**# Step 6: Save the Q-Table**

np.save("q\_table.npy", q\_table)

print("Q-Table saved as 'q\_table.npy'.")

**# Step 7: Test the Trained Model**

state = discretize\_state(env.reset()[0], state\_bins)

done = False

total\_reward = 0

while not done:

action = np.argmax(q\_table[state]) # Use trained Q-Table

next\_state\_raw, reward, done, \_, \_ = env.step(action)

state = discretize\_state(next\_state\_raw, state\_bins)

total\_reward += reward

env.render()

env.close()

print(f"Total reward during test: {total\_reward}")

**Practical 8**

**Aim :** Utilizing transfer learning to improve model performance on limited datasets.

**Code :**

**Step 1: Install Required Libraries**

Run the following command to install the necessary libraries:

!pip install torch torchvision matplotlib numpy

**Step 2: Python Code for Transfer Learning**

Save the following code as transfer\_learning.py.

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import datasets, models, transforms

from torch.utils.data import DataLoader

import matplotlib.pyplot as plt

**# Step 1: Set Device and Hyperparameters**

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

num\_classes = 10 # CIFAR-10 has 10 classes

batch\_size = 32

epochs = 10

learning\_rate = 0.001

**# Step 2: Define Data Transformations**

transform = transforms.Compose([

transforms.Resize((224, 224)), # Resize to match ResNet input size

transforms.ToTensor(),

transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),

])

**# Step 3: Load CIFAR-10 Dataset**

train\_dataset = datasets.CIFAR10(root="data", train=True, transform=transform, download=True)

test\_dataset = datasets.CIFAR10(root="data", train=False, transform=transform, download=True)

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size, shuffle=False)

**# Step 4: Load Pre-trained ResNet18 Model**

model = models.resnet18(pretrained=True)

for param in model.parameters():

param.requires\_grad = False # Freeze all layers

**# Replace the final fully connected layer for CIFAR-10**

model.fc = nn.Linear(model.fc.in\_features, num\_classes)

model = model.to(device)

**# Step 5: Define Loss and Optimizer**

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.fc.parameters(), lr=learning\_rate)

**# Step 6: Training Loop**

def train(model, loader, criterion, optimizer):

model.train()

running\_loss = 0.0

correct = 0

total = 0

for inputs, labels in loader:

inputs, labels = inputs.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

running\_loss += loss.item()

\_, predicted = outputs.max(1)

total += labels.size(0)

correct += predicted.eq(labels).sum().item()

accuracy = 100. \* correct / total

return running\_loss / len(loader), accuracy

**# Step 7: Testing Loop**

def test(model, loader, criterion):

model.eval()

running\_loss = 0.0

correct = 0

total = 0

with torch.no\_grad():

for inputs, labels in loader:

inputs, labels = inputs.to(device), labels.to(device)

outputs = model(inputs)

loss = criterion(outputs, labels)

running\_loss += loss.item()

\_, predicted = outputs.max(1)

total += labels.size(0)

correct += predicted.eq(labels).sum().item()

accuracy = 100. \* correct / total

return running\_loss / len(loader), accuracy

**# Step 8: Train and Evaluate**

train\_losses, test\_losses = [], []

train\_accuracies, test\_accuracies = [], []

for epoch in range(epochs):

train\_loss, train\_acc = train(model, train\_loader, criterion, optimizer)

test\_loss, test\_acc = test(model, test\_loader, criterion)

train\_losses.append(train\_loss)

test\_losses.append(test\_loss)

train\_accuracies.append(train\_acc)

test\_accuracies.append(test\_acc)

print(f"Epoch {epoch+1}/{epochs}")

print(f"Train Loss: {train\_loss:.4f}, Train Accuracy: {train\_acc:.2f}%")

print(f"Test Loss: {test\_loss:.4f}, Test Accuracy: {test\_acc:.2f}%")

**# Step 9: Plot Training and Testing Metrics**

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(train\_losses, label="Train Loss")

plt.plot(test\_losses, label="Test Loss")

plt.legend()

plt.title("Loss")

plt.subplot(1, 2, 2)

plt.plot(train\_accuracies, label="Train Accuracy")

plt.plot(test\_accuracies, label="Test Accuracy")

plt.legend()

plt.title("Accuracy")

plt.show()

**# Step 10: Save the Model**

torch.save(model.state\_dict(), "resnet18\_cifar10.pth")

print("Model saved as 'resnet18\_cifar10.pth'.")

**Practical 9**

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

**Code :**

!pip install numpy pandas matplotlib scikit-learn tensorflow

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

**# Step 1: Load the Dataset**

url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv"

data = pd.read\_csv(url, usecols=[1], header=0)

data = data.values.astype("float32")  # Ensure the data is float

**# Step 2: Visualize the Data**

plt.plot(data)

plt.title("Airline Passengers Over Time")

plt.xlabel("Time")

plt.ylabel("Passengers")

plt.show()

**# Step 3: Normalize the Data**

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

scaled\_data = scaler.fit\_transform(data)

**# Step 4: Prepare the Data for LSTM**

def create\_dataset(dataset, look\_back=1):

    X, y = [], []

    for i in range(len(dataset) - look\_back):

        X.append(dataset[i:(i + look\_back), 0])

        y.append(dataset[i + look\_back, 0])

    return np.array(X), np.array(y)

look\_back = 12  # Use 12 months (1 year) as input to predict the next value

X, y = create\_dataset(scaled\_data, look\_back)

X = X.reshape((X.shape[0], X.shape[1], 1))  # Reshape for LSTM [samples, time\_steps, features]

**# Step 5: Split Data into Training and Testing Sets**

train\_size = int(len(X) \* 0.8)

X\_train, X\_test = X[:train\_size], X[train\_size:]

y\_train, y\_test = y[:train\_size], y[train\_size:]

**# Step 6: Build the LSTM Model**

model = Sequential([

    LSTM(50, activation="relu", input\_shape=(look\_back, 1)),

    Dense(1)

])

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

**# Step 7: Train the Model**

history = model.fit(X\_train, y\_train, epochs=50, batch\_size=32, validation\_data=(X\_test, y\_test), verbose=1)

**# Step 8: Evaluate the Model**

loss = model.evaluate(X\_test, y\_test, verbose=0)

print(f"Test Loss: {loss:.4f}")

**# Step 9: Predict and Inverse Transform**

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

y\_pred = scaler.inverse\_transform(y\_pred)

y\_test\_actual = scaler.inverse\_transform(y\_test.reshape(-1, 1))

**# Step 10: Plot Actual vs Predicted**

plt.figure(figsize=(10, 5))

plt.plot(y\_test\_actual, label="Actual")

plt.plot(y\_pred, label="Predicted")

plt.title("Actual vs Predicted Airline Passengers")

plt.xlabel("Time")

plt.ylabel("Passengers")

plt.legend()

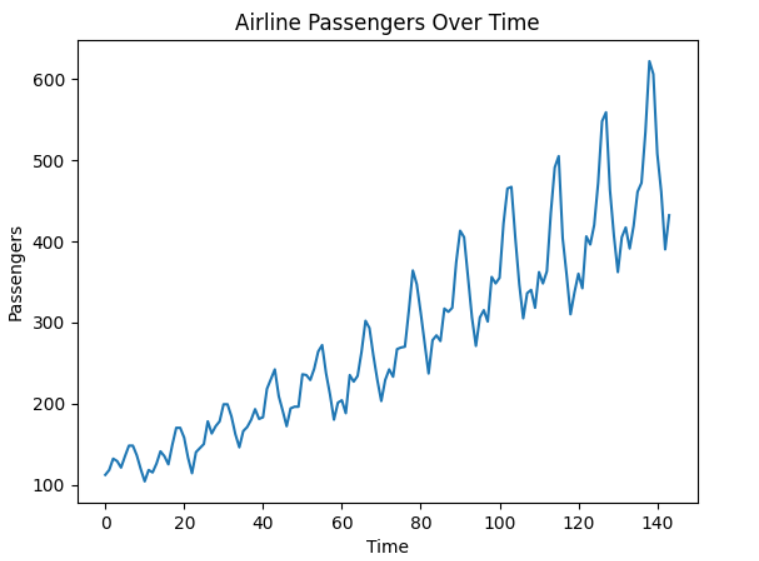
plt.show()

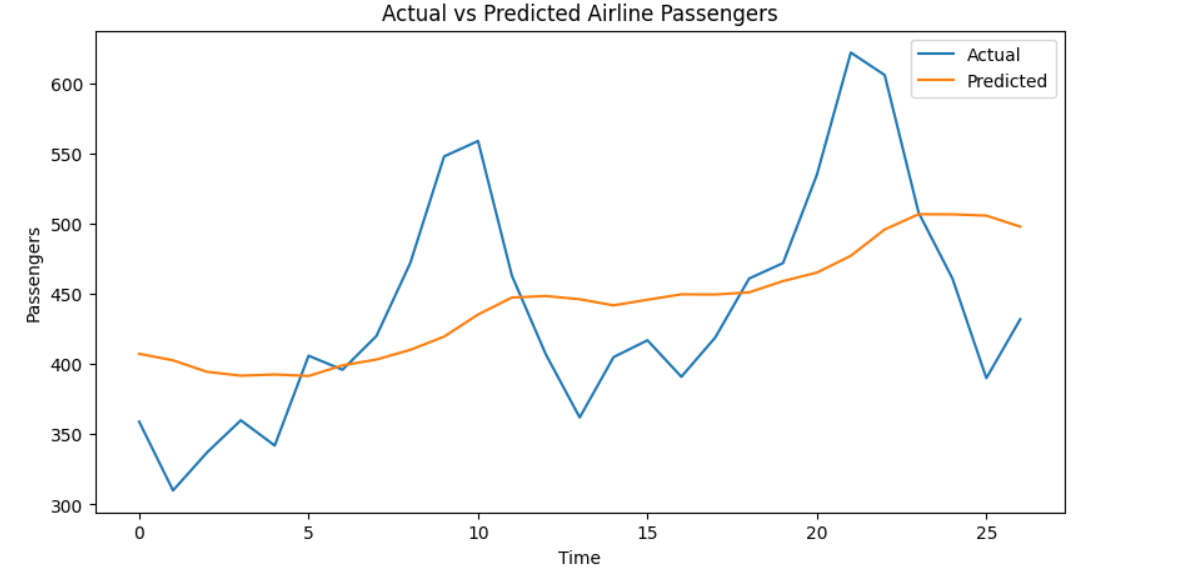
**# Step 11: Save the Model**

model.save("lstm\_time\_series.h5")

print("Model saved as 'lstm\_time\_series.h5'.")

**Output :**

****

****

**Practical 10**

**Aim :** Implementing a machine learning pipeline for automated feature engineering and model selection.

**Code :**

**Step 1: Install Required Libraries**

Run the following command to install the required libraries

pip install pandas numpy scikit-learn

**Step 2: Python Code for Machine Learning Pipeline**

Save the following code as ml\_pipeline.py.

import pandas as pd

import numpy as np

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

from sklearn.pipeline import Pipeline

from sklearn.compose import ColumnTransformer

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.feature\_selection import SelectKBest, f\_classif

from sklearn.ensemble import RandomForestClassifier

from sklearn.svm import SVC

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

**# Step 1: Load Dataset**

# Using the Titanic dataset for demonstration

url = "https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"

data = pd.read\_csv(url)

**# Step 2: Basic Preprocessing**

# Drop unnecessary columns

data = data.drop(["PassengerId", "Name", "Ticket", "Cabin"], axis=1)

**# Handle missing values**

data["Age"].fillna(data["Age"].median(), inplace=True)

data["Embarked"].fillna(data["Embarked"].mode()[0], inplace=True)

**# Separate features and target**

X = data.drop("Survived", axis=1)

y = data["Survived"]

**# Step 3: Split Data into Train and Test Sets**

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

**# Step 4: Define Preprocessing Steps**

# Numerical features: Scale values

numerical\_features = ["Age", "Fare"]

numerical\_transformer = Pipeline(steps=[

("scaler", StandardScaler())

])

# Categorical features: One-hot encode

categorical\_features = ["Sex", "Embarked", "Pclass"]

categorical\_transformer = Pipeline(steps=[

("onehot", OneHotEncoder(handle\_unknown="ignore"))

])

# Combine preprocessors into a column transformer

preprocessor = ColumnTransformer(

transformers=[

("num", numerical\_transformer, numerical\_features),

("cat", categorical\_transformer, categorical\_features)

]

)

**# Step 5: Define Feature Selection and Model Options**

feature\_selection = SelectKBest(score\_func=f\_classif)

# Define candidate models

models = {

"RandomForest": RandomForestClassifier(random\_state=42),

"SVC": SVC(probability=True, random\_state=42)

}

**# Step 6: Create the Pipeline**

pipeline = Pipeline(steps=[

("preprocessor", preprocessor),

("feature\_selection", feature\_selection),

("classifier", RandomForestClassifier())

])

**# Step 7: Define Grid Search for Hyperparameter Tuning**

param\_grid = {

"feature\_selection\_\_k": [5, 6, 7],

"classifier": [models["RandomForest"], models["SVC"]],

"classifier\_\_n\_estimators": [100, 200] if "n\_estimators" in RandomForestClassifier().get\_params() else [None],

"classifier\_\_C": [0.1, 1, 10] if "C" in SVC().get\_params() else [None]

}

grid\_search = GridSearchCV(pipeline, param\_grid, cv=3, scoring="accuracy", verbose=2)

**# Step 8: Train the Model**

grid\_search.fit(X\_train, y\_train)

**# Step 9: Evaluate the Model**

best\_model = grid\_search.best\_estimator\_

y\_pred = best\_model.predict(X\_test)

print("Best Parameters:", grid\_search.best\_params\_)

print("Accuracy on Test Set:", accuracy\_score(y\_test, y\_pred))

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

**Practical 11**

**Aim :** Using advanced optimization techniques like evolutionary algorithms or Bayesian optimization for hyperparameter tuning.

**Code :**

!pip install numpy pandas scikit-learn scikit-optimize

import numpy as np

import pandas as pd

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

from sklearn.ensemble import RandomForestClassifier

from skopt import BayesSearchCV

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

**# Step 1: Load the Dataset**

data = load\_iris()

X, y = data.data, data.target

**# Step 2: Split the Data 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)

**# Step 3: Define the Model**

model = RandomForestClassifier(random\_state=42)

**# Step 4: Define the Search Space for Hyperparameters**

param\_space = {

    "n\_estimators": (10, 200),          # Number of trees in the forest

    "max\_depth": (1, 20),              # Maximum depth of each tree

    "min\_samples\_split": (2, 10),      # Minimum samples to split a node

    "min\_samples\_leaf": (1, 10),       # Minimum samples at each leaf

    "max\_features": ["sqrt", "log2", None]  # Number of features considered for split

}

**# Step 5: Use Bayesian Optimization for Hyperparameter Tuning**

optimizer = BayesSearchCV(

    estimator=model,

    search\_spaces=param\_space,

    n\_iter=30,  # Number of iterations to search

    cv=3,       # 3-fold cross-validation

    random\_state=42,

    n\_jobs=-1

)

**# Step 6: Train the Optimized Model**

print("Starting Bayesian Optimization...")

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

**# Step 7: Evaluate the Best Model**

best\_model = optimizer.best\_estimator\_

y\_pred = best\_model.predict(X\_test)

print("\nBest Parameters:", optimizer.best\_params\_)

print("Accuracy on Test Set:", accuracy\_score(y\_test, y\_pred))

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

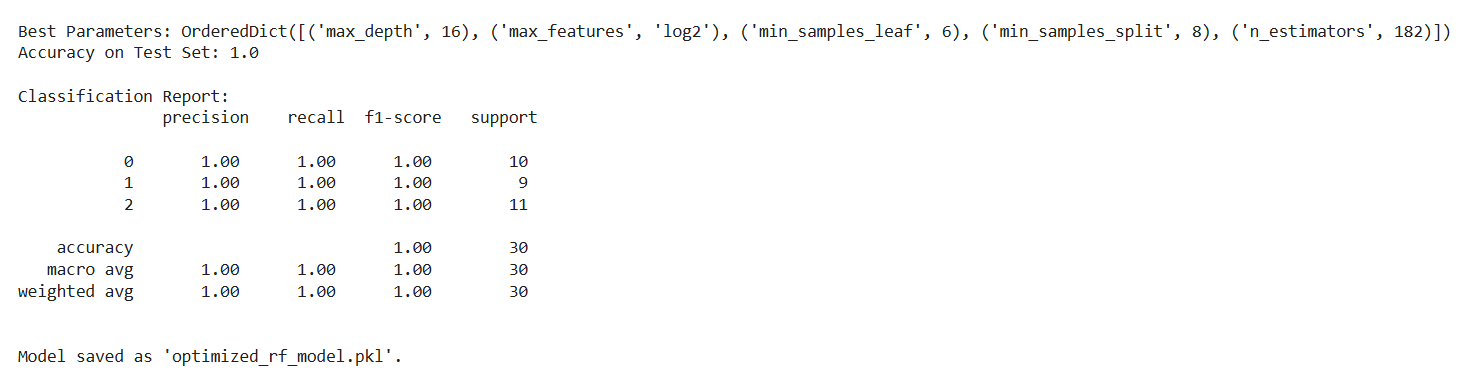
**# Optional: Save the Best Model**

import joblib

joblib.dump(best\_model, "optimized\_rf\_model.pkl")

print("\nModel saved as 'optimized\_rf\_model.pkl'.")

**Output :**

****

**Practical 12**

**Aim** : Deploying a machine learning model in a production environment using containerization and cloud services

**Code :**

**Step 1: Install Required Libraries**

!pip install scikit-learn pandas fastapi uvicorn

**Step 2: Build the Machine Learning Model**

Save the following Python script as train\_model.py. This script trains the model and saves it.

import pandas as pd

from sklearn.datasets import load\_iris

from sklearn.ensemble import RandomForestClassifier

import joblib

**# Step 1: Load Dataset**

data = load\_iris()

X, y = data.data, data.target

**# Step 2: Train Model**

model = RandomForestClassifier(random\_state=42)

model.fit(X, y)

**# Step 3: Save Model**

joblib.dump(model, "iris\_model.pkl")

print("Model saved as iris\_model.pkl")

Run the script to save the trained model:

bash

Copy code

python train\_model.py

**Step 3: Create the API**

Save the following Python script as app.py.

python

Copy code

from fastapi import FastAPI

from pydantic import BaseModel

import joblib

import numpy as np

**# Step 1: Load the trained model**

model = joblib.load("iris\_model.pkl")

**# Step 2: Initialize FastAPI**

app = FastAPI()

**# Step 3: Define input schema**

class IrisRequest(BaseModel):

sepal\_length: float

sepal\_width: float

petal\_length: float

petal\_width: float

**# Step 4: Define prediction endpoint**

@app.post("/predict/")

def predict(iris: IrisRequest):

features = np.array([[iris.sepal\_length, iris.sepal\_width, iris.petal\_length, iris.petal\_width]])

prediction = model.predict(features)

species = ["setosa", "versicolor", "virginica"]

return {"prediction": species[prediction[0]]}

Run the API locally for testing:

bash

Copy code

uvicorn app:app --host 0.0.0.0 --port 8000

Test the API in your browser or with a tool like **Postman**:

* URL: http://127.0.0.1:8000/predict/
* Example Request Body:

json

Copy code

{

"sepal\_length": 5.1,

"sepal\_width": 3.5,

"petal\_length": 1.4,

"petal\_width": 0.2

}

**Step 4: Create a Dockerfile**

Save the following as Dockerfile.

dockerfile

Copy code

# Use an official Python runtime as the base image

FROM python:3.9-slim

# Set the working directory in the container

WORKDIR /app

# Copy the current directory contents into the container

COPY . /app

# Install required Python libraries

RUN pip install --no-cache-dir fastapi uvicorn scikit-learn joblib

# Expose the API port

EXPOSE 8000

# Command to run the application

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]

**Step 5: Build and Run the Docker Container**

1. **Build the Docker image**:

bash

Copy code

docker build -t iris-api .

1. **Run the Docker container**:

bash

Copy code

docker run -d -p 8000:8000 iris-api

1. **Test the API**:
   * URL: http://localhost:8000/predict/
   * Use the same JSON request as earlier.

**Step 6: Deploy to a Cloud Service (Optional)**

1. **Prepare the Docker Image**:
   * Tag the image for a container registry (e.g., Docker Hub, AWS ECR, or GCP Artifact Registry):

bash

Copy code

docker tag iris-api <your\_dockerhub\_username>/iris-api

docker push <your\_dockerhub\_username>/iris-api

1. **Deploy to AWS ECS (Example)**:
   * Create an ECS cluster.
   * Use the Docker image in a task definition.
   * Deploy the task to the cluster.
2. **Other Options**:
   * Use **AWS Lambda** with **API Gateway**.
   * Deploy on **Google Cloud Run** or **Azure App Service** for managed hosting.

**Step 7: Dataset**

* **Iris Dataset**:
  + Included with sklearn.datasets for demonstration purposes.
  + Automatically loaded in the train\_model.py script.

**Practical 13**

**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.

**Code :**

**Step 1: Install Required Libraries**

Run the following commands to install necessary libraries:

!pip install transformers datasets torch

**Step 2: Prepare a Text Dataset**

For demonstration, we'll use the Tiny Shakespeare Corpus available via Hugging Face's datasets library. Alternatively, you can use your own dataset.

**Step 3: Python Code for Training and Generating Text**

Save the following code as train\_gpt2.py.

import os

from datasets import load\_dataset

from transformers import GPT2LMHeadModel, GPT2Tokenizer, Trainer, TrainingArguments

**# Step 1: Load the Dataset**

print("Loading dataset...")

dataset = load\_dataset("tiny\_shakespeare")

**# Split into train and test sets**

train\_data = dataset["train"]

test\_data = dataset["test"]

**# Step 2: Load Pre-trained GPT-2 Tokenizer and Model**

print("Loading GPT-2 tokenizer and model...")

tokenizer = GPT2Tokenizer.from\_pretrained("gpt2")

model = GPT2LMHeadModel.from\_pretrained("gpt2")

**# Step 3: Tokenize the Dataset**

def tokenize\_function(examples):

return tokenizer(examples["text"], truncation=True, padding="max\_length", max\_length=512)

print("Tokenizing dataset...")

tokenized\_train = train\_data.map(tokenize\_function, batched=True)

tokenized\_test = test\_data.map(tokenize\_function, batched=True)

**# Step 4: Define Training Arguments**

training\_args = TrainingArguments(

output\_dir="./results",

evaluation\_strategy="epoch",

learning\_rate=5e-5,

weight\_decay=0.01,

per\_device\_train\_batch\_size=4,

num\_train\_epochs=3,

save\_total\_limit=2,

logging\_dir="./logs",

logging\_steps=10,

)

**# Step 5: Initialize Trainer**

trainer = Trainer(

model=model,

args=training\_args,

train\_dataset=tokenized\_train,

eval\_dataset=tokenized\_test,

)

**# Step 6: Train the Model**

print("Starting training...")

trainer.train()

**# Step 7: Save the Fine-Tuned Model**

model.save\_pretrained("./fine\_tuned\_gpt2")

tokenizer.save\_pretrained("./fine\_tuned\_gpt2")

print("Model saved to './fine\_tuned\_gpt2'.")

**# Step 8: Generate Text Using the Fine-Tuned Model**

print("Generating new text...")

model.eval()

input\_text = "To be or not to be, that is the"

inputs = tokenizer.encode(input\_text, return\_tensors="pt")

outputs = model.generate(inputs, max\_length=100, num\_return\_sequences=1, temperature=0.7)

generated\_text = tokenizer.decode(outputs[0], skip\_special\_tokens=True)

print("\nGenerated Text:\n")

print(generated\_text)

**Practical 14**

**Aim :** Experiment with neural networks like GANs (Generative Adversarial Networks) using Python libraries like TensorFlow or PyTorch to generate new images based on a dataset of images.

**Code :**

!pip install torch torchvision matplotlib

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import datasets, transforms

from torch.utils.data import DataLoader

import matplotlib.pyplot as plt

import os

**# Step 1: Set Device Configuration**

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

print(f"Using device: {device}")

**# Step 2: Define Generator**

class Generator(nn.Module):

    def \_\_init\_\_(self, noise\_dim, img\_dim):

        super(Generator, self).\_\_init\_\_()

        self.gen = nn.Sequential(

            nn.Linear(noise\_dim, 256),

            nn.ReLU(),

            nn.Linear(256, 512),

            nn.ReLU(),

            nn.Linear(512, img\_dim),

            nn.Tanh()

        )

    def forward(self, x):

        return self.gen(x)

**# Step 3: Define Discriminator**

class Discriminator(nn.Module):

    def \_\_init\_\_(self, img\_dim):

        super(Discriminator, self).\_\_init\_\_()

        self.disc = nn.Sequential(

            nn.Linear(img\_dim, 512),

            nn.LeakyReLU(0.2),

            nn.Linear(512, 256),

            nn.LeakyReLU(0.2),

            nn.Linear(256, 1),

            nn.Sigmoid()

        )

    def forward(self, x):

        return self.disc(x)

**# Step 4: Define Hyperparameters**

noise\_dim = 100

img\_dim = 28 \* 28  # 28x28 images flattened

batch\_size = 64

lr = 0.0002

epochs = 50

**# Step 5: Load Dataset**

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

dataset = datasets.MNIST(root="data", train=True, transform=transform, download=True)

dataloader = DataLoader(dataset, batch\_size=batch\_size, shuffle=True)

**# Step 6: Initialize Models, Optimizers, and Loss Function**

gen = Generator(noise\_dim, img\_dim).to(device)

disc = Discriminator(img\_dim).to(device)

criterion = nn.BCELoss()

opt\_gen = optim.Adam(gen.parameters(), lr=lr)

opt\_disc = optim.Adam(disc.parameters(), lr=lr)

**# Step 7: Training Loop**

print("Starting Training...")

for epoch in range(epochs):

    for batch\_idx, (real, \_) in enumerate(dataloader):

        real = real.view(-1, img\_dim).to(device)

        batch\_size = real.size(0)

        # Train Discriminator

        noise = torch.randn(batch\_size, noise\_dim).to(device)

        fake = gen(noise)

        disc\_real = disc(real).view(-1)

        disc\_fake = disc(fake.detach()).view(-1)

        loss\_disc = criterion(disc\_real, torch.ones\_like(disc\_real)) + \

                    criterion(disc\_fake, torch.zeros\_like(disc\_fake))

        opt\_disc.zero\_grad()

        loss\_disc.backward()

        opt\_disc.step()

        # Train Generator

        output = disc(fake).view(-1)

        loss\_gen = criterion(output, torch.ones\_like(output))

        opt\_gen.zero\_grad()

        loss\_gen.backward()

        opt\_gen.step()

    print(f"Epoch [{epoch+1}/{epochs}] | Loss D: {loss\_disc:.4f}, Loss G: {loss\_gen:.4f}")

**# Save and Display Sample Images**

    if (epoch + 1) % 10 == 0:

        with torch.no\_grad():

            fake\_images = gen(torch.randn(16, noise\_dim).to(device)).view(-1, 1, 28, 28)

        plt.figure(figsize=(4, 4))

        for i in range(16):

            plt.subplot(4, 4, i+1)

            plt.imshow(fake\_images[i][0].cpu(), cmap="gray")

            plt.axis("off")

        plt.tight\_layout()

        os.makedirs("generated\_images", exist\_ok=True)

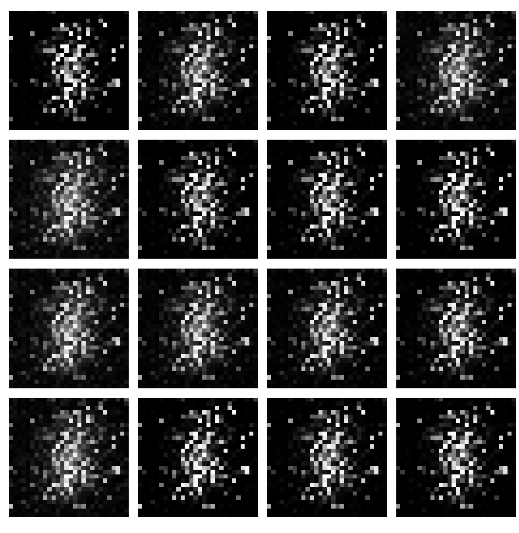
        plt.savefig(f"generated\_images/epoch\_{epoch+1}.png")

        plt.close()

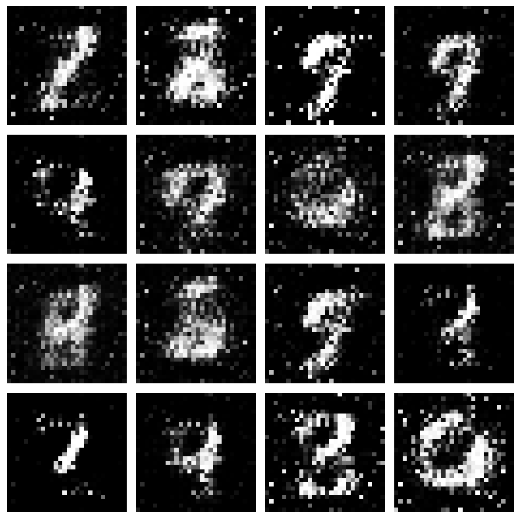
print("Training Complete. Generated images are saved in 'generated\_images' folder.")

**Output :**

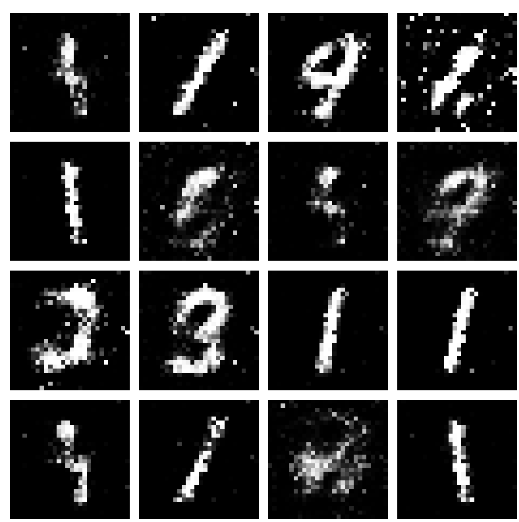
**Generate after 10 Epoch**

****

**Generate after 20 Epoch**

****

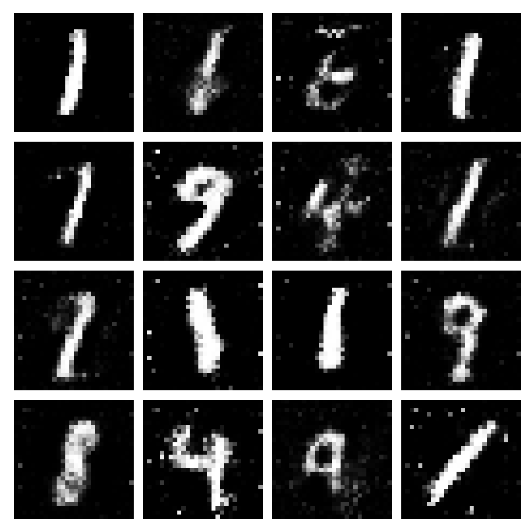
**Generate 30 Epoch**

****

**Generate after 40 Epoch**

****

**Generate after 50 Epoch**

****

**PRACTICAL JOURNAL**

in

**MACHINE LEARNING**

Submitted to

**Laxman Devram Sonawane College, Kalyan (W) 421301**

in partial fulfilment for the award of the degree of

** Master of Science in Information Technology**

(Affiliated to Mumbai University)

*Submitted by*

**TANIA TAZIM JAMDAR**

Under the guidance of

**Dr. PRIYANKA PAWAR**

Department of Information Technology

Kalyan, Maharashtra

Academic Year 2024-25



The Kalyan Wholesale Merchants Education Society’s

**Laxman Devram Sonawane College,**

**Kalyan (W) 421301**

**Department of Information Technology**

**Masters of Science – Part II**

**Certificate**

This is to certify that **Ms. Tania Tazim Jamdar**, Seat number**\_\_\_\_\_\_\_\_\_\_\_\_\_**, studying in Masters of Science in Information Technology Part II Semester II has satisfactorily completed the practical of “**Machine Learning** ”as prescribed by University of Mumbai, during the academic year 2024-25.

Subject In-charge Coordinator In-charge External Examiner

College Seal

**MACHINE LEARNING**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.no**. | **Practical** | **Date** | **Sign** |
| **1**. | **Data Pre-processing and Exploration**   1. Load a CSV dataset. Handle missing values, inconsistent formatting, and outliers. 2. Load a dataset, calculate descriptive summary statistics, create visualizations using different graphs, and identify potential features and target variables Note: Explore Univariate and Bivariate graphs (Matplotlib) and Seaborn for visualization 3. Create or Explore datasets to use all pre-processing routines like label encoding, scaling, and binarization. |  |  |
| **2**. | **Testing Hypothesis**   1. Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from CSV file and generate the final specific hypothesis. (Create your dataset) |  |  |
| **3**. | **Linear Models**   1. Simple Linear Regression Fit a linear regression model on a dataset. Interpret coefficients, make predictions, and evaluate performance using metrics like R-squared and MSE 2. Multiple Linear Regression Extend linear regression to multiple feature. Handle feature selection and potential multicollinearity 3. Regualarized Linear Models Implement Regression variants like LASSO and Ridge on any generated dataset |  |  |
| **4**. | **Discriminative Models**   1. Logistic Regression : Perform binary classification using logistic regression. Calculate accuracy, precision, recall, and understand the ROC curve." 2. Implement and demonstrate k-nearest Neighbor algorithm. Read the training data from a .CSV file and build the model to classify a test sample. Print both correct and wrong predictions. 3. Build a decision tree classifier or regressor. Control hyperparameters like tree depth to avoid overfitting. Visualize the tree. 4. Implement a Support Vector Machine for any relevant dataset. 5. Train a random forest ensemble. Experiment with the number of trees and feature sampling. Compare performance to a single decision tree. 6. Implement a gradient boosting machine (e.g., XGBoost). Tune hyperparameters and explore feature importance. |  |  |
| **5**. | **Generative Models**   1. Implement and demonstrate the working of a Naive Bayesian classifier using a sample data set. Build the model to classify a test sample. 2. Implement Hidden Markov Models using hmmlearn |  |  |
| **6**. | **Probabilistic Models**   1. Implement Bayesian Linear Regression to explore prior and posterior distribution. 2. Implement Gaussian Mixture Models for density estimation and unsupervised clustering. |  |  |
| **7**. | **Model Evaluation and Hyperparameter Tuning**   1. Implement cross-validation techniques (k-fold, stratified, etc.) for robust model evaluation 2. Systematically explore combinations of hyperparameters to optimize model performance.(use grid and randomized search) |  |  |
| **8**. | **Bayesian Learning**   1. Implement Bayesian Learning using inferences |  |  |

**Practical 1: Data Pre-processing and Exploration**

**1a. Load a CSV dataset. Handle missing values, inconsistent formatting, and outliers.**

**Code :**

1. **Import Libraries**

**# Import necessary libraries**

import pandas as pd import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

1. **Load the Dataset**

**# Load the Titanic dataset from a URL**

url="https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv" data = pd.read\_csv(url)

**# Display the first few rows**

print(data.head())

1. **Handle Missing Values**

**# Check for missing values**

print("Missing values in each column:")

print(data.isnull().sum())

**# Fill missing values in 'Age' with the mean**

data['Age'].fillna(data['Age'].mean(), inplace=True)

**# Fill missing values in 'Embarked' with the most common value** data['Embarked'].fillna(data['Embarked'].mode()[0], inplace=True)

**# Drop rows where 'Cabin' is missing (too many NaNs)**

data.drop(columns=['Cabin'], inplace=True)

**# Verify missing values are handled**

print("\nAfter handling missing values:")

print(data.isnull().sum())

1. **Fix Inconsistent Formatting**

**# Fix inconsistent formatting in the 'Sex' column**

data['Sex'] = data['Sex'].str.lower().str.strip()

**# Verify unique values**

print("\nUnique values in 'Sex' column after formatting:")

print(data['Sex'].unique())

1. **Detect and Handle Outliers**  **# Boxplot for the 'Fare' column** sns.boxplot(data['Fare'], color='skyblue') plt.title('Boxplot of Fare') plt.show()

**# Detect outliers using the IQR method**

Q1 = data['Fare'].quantile(0.25) Q3 = data['Fare'].quantile(0.75) IQR = Q3 - Q1 lower\_bound = Q1 - 1.5 \* IQR upper\_bound = Q3 + 1.5 \* IQR

**# Capping outliers**

data['Fare'] = np.where(data['Fare'] > upper\_bound, upper\_bound, np.where(data['Fare'] < lower\_bound, lower\_bound, data['Fare']))

**# Verify with an updated** **boxplot**

sns.boxplot(data['Fare'], color='lightgreen')

plt.title('Boxplot of Fare (After Handling Outliers)')

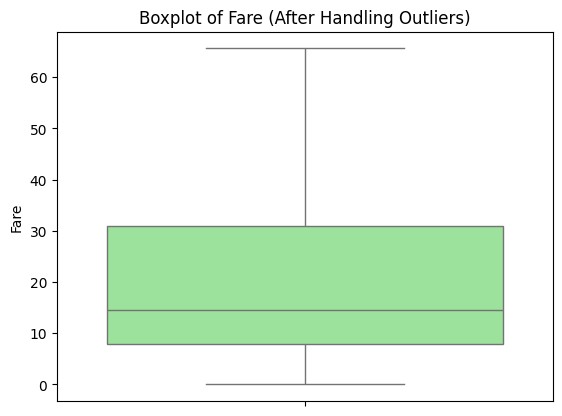
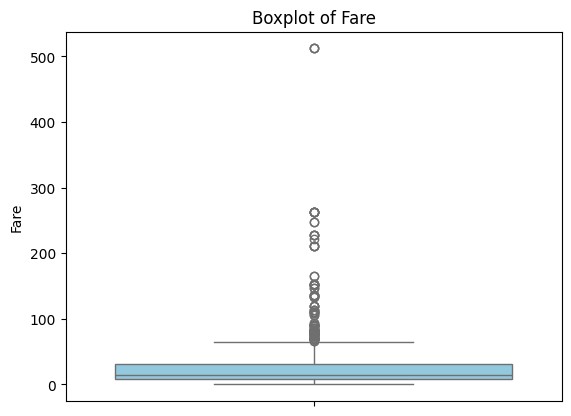
plt.show()

1. **Save the Cleaned Dataset**  **# Save the cleaned dataset**

data.to\_csv('cleaned\_titanic.csv', index=False)

print("\nCleaned dataset saved as 'cleaned\_titanic.csv'") .

**Output :**



**1b. Load a dataset, calculate descriptive summary statistics, create visualizations using different graphs, and identify potential features and target variables Note:**

**Explore Univariate and Bivariate graphs (Matplotlib) and Seaborn for visualization**

**Code :**

1. **Import Necessary Libraries # Import required libraries**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

1. **Load the Dataset**

**# Load the dataset from the URL**

url = "https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv"

data = pd.read\_csv(url)

**# Display the first few rows**

print("First 5 rows of the dataset:") print(data.head())

1. **Calculate Descriptive Summary Statistics # Dataset information**

print("\nDataset Info:")

print(data.info())

**# Summary statistics for numerical columns**

print("\nDescriptive Statistics for Numerical Columns:") print(data.describe())

**# Check unique values for categorical columns**

print("\nUnique values in 'species' column:") print(data['species'].value\_counts())

1. **Univariate Analysis**

**# Histograms for numerical columns**

data.hist(figsize=(10,8), color='skyblue', edgecolor='black') plt.suptitle("Histograms of Numerical Features")

plt.show()

**# Bar plot for 'species' column** sns.countplot(x='species', data=data, palette='pastel') plt.title("Count of Each Species") plt.show()

1. **Bivariate Analysis**

**# Scatter plot for two features**

plt.figure(figsize=(8, 6))

plt.scatter(data['sepal\_length'], data['sepal\_width'], alpha=0.7, c='blue')

plt.title("Sepal Length vs Sepal Width")

plt.xlabel("Sepal Length")

plt.ylabel("Sepal Width")

plt.show()

**# Pairplot to visualize relationships between features** sns.pairplot(data, hue='species', palette='husl', diag\_kind='kde') plt.suptitle("Pairplot of Features by Species", y=1.02)

plt.show()

**# Boxplot for petal\_length across species** sns.boxplot(x='species', y='petal\_length', data=data, palette='Set3')

plt.title("Boxplot of Petal Length by Species")

plt.show()

1. **Identify Potential Features and Target Variables**

**# Separate features and target**

features = data.drop(columns=['species'])

**# Drop the target**

column target = data['species']

**# Target variable**

print("\nFeatures:")

print(features.head())

print("\nTarget:")

print(target.head())

**# Visualize target distribution** sns.countplot(x=target, palette='viridis') plt.title("Target Variable Distribution")

plt.show()

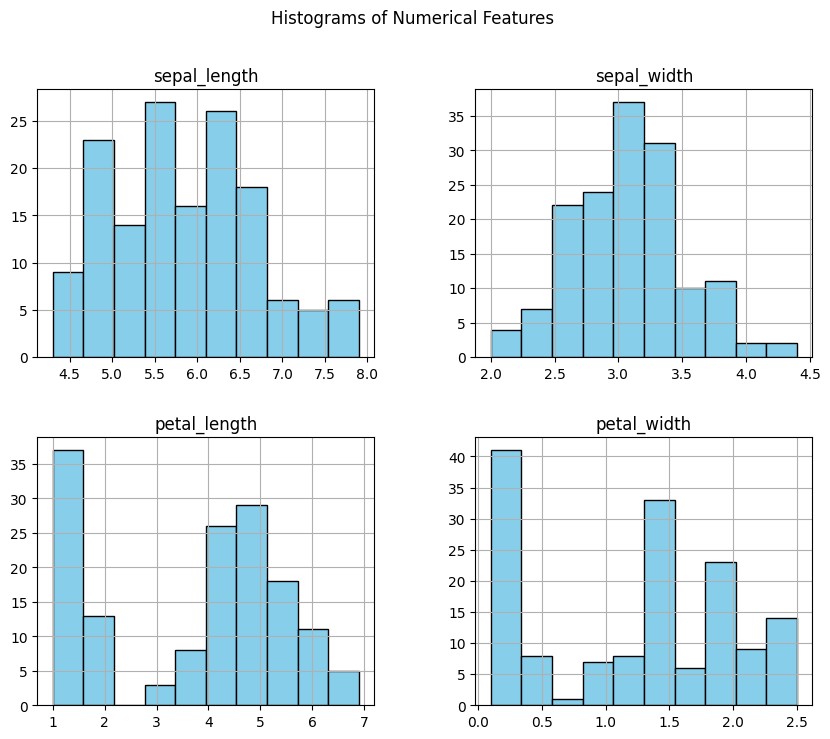
1. **Save the Cleaned and Processed Dataset**

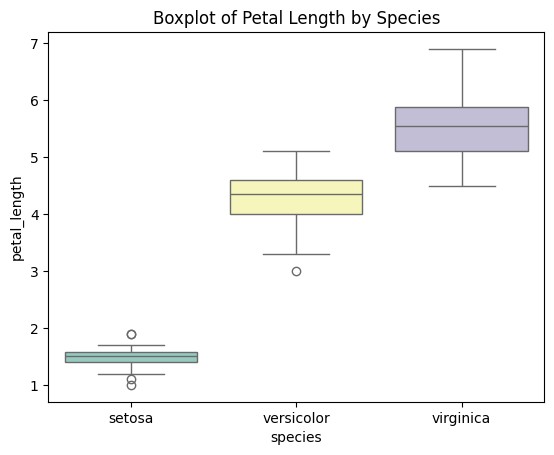
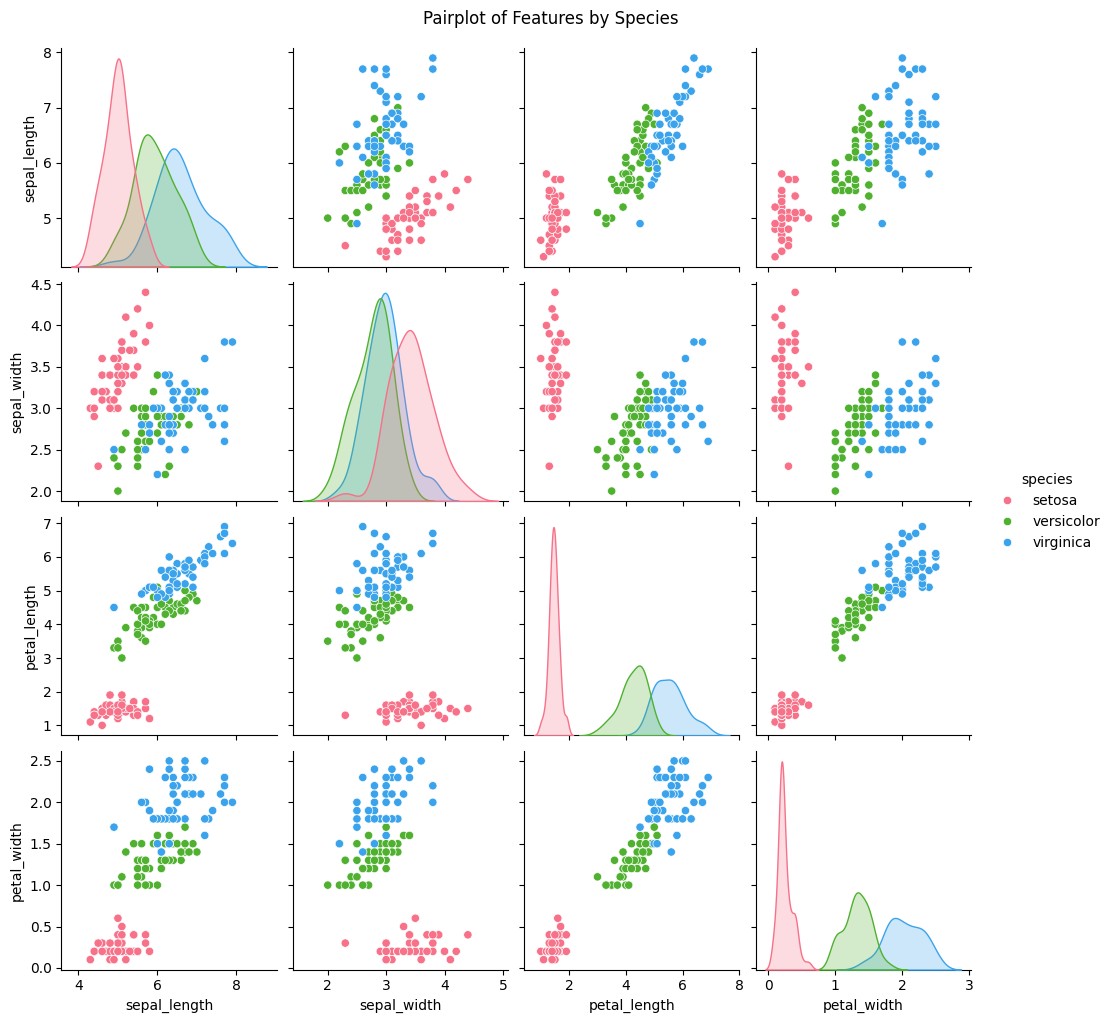
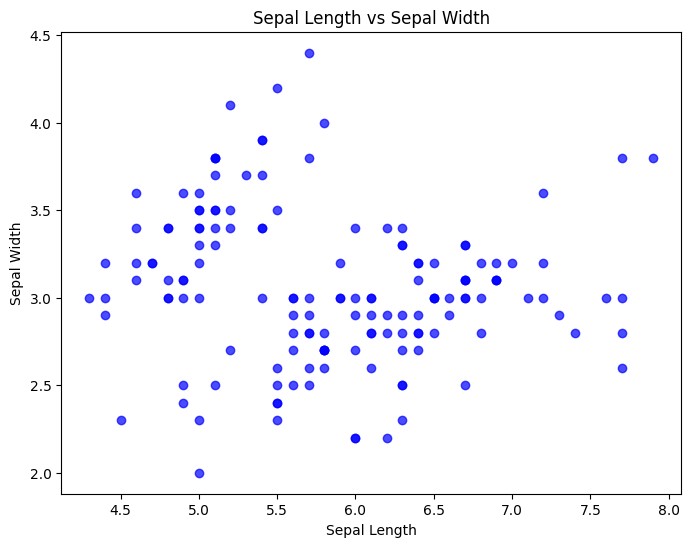
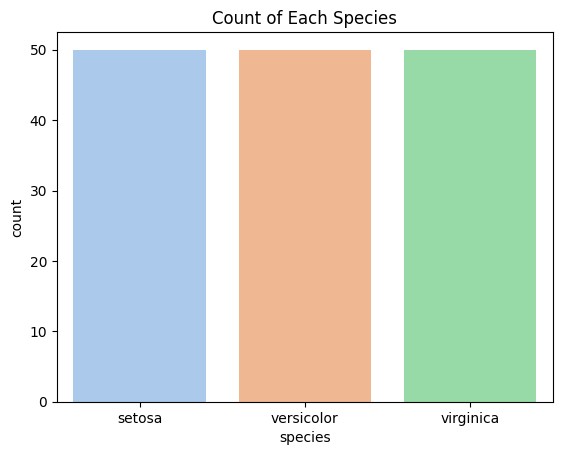
**# Save the dataset**

data.to\_csv('processed\_iris.csv', index=False)

print("\nProcessed dataset saved as 'processed\_iris.csv'")

**Output :**





**1c. Create or Explore datasets to use all pre-processing routines like label encoding, scaling, and binarization.**

**Code :**

1. **Import Necessary Libraries # Import required libraries**

import pandas as pd

import numpy as np from sklearn.preprocessing

import LabelEncoder, MinMaxScaler, StandardScaler, Binarizer

1. **Create or Load a Dataset # Create a sample dataset**

data = pd.DataFrame({

'Category': ['A', 'B', 'C', 'A', 'B', 'C'],

**# Categorical variable**

'Age': [23, 45, 31, 22, 35, 30],

**# Numerical variable**

'Income': [50000, 60000, 70000, 80000, 90000, 100000],

**# Numerical variable 'Has\_Car':**

['Yes', 'No', 'Yes', 'No', 'Yes', 'No']

**# Binary categorical variable** })

**# Display the dataset**

print("Sample Dataset:")

print(data)

1. **Apply Pre-Processing Routines**

**# Label Encoding for 'Category' column**

label\_encoder = LabelEncoder()

data['Category\_Encoded'] = label\_encoder.fit\_transform(data['Category'])

**# Label Encoding for binary column** **'Has\_Car'**

data['Has\_Car\_Encoded'] = label\_encoder.fit\_transform(data['Has\_Car']) print("\nAfter Label Encoding:")

print(data)

**# Min-Max Scaling for 'Income'**

min\_max\_scaler = MinMaxScaler()

data['Income\_MinMax'] = min\_max\_scaler.fit\_transform(data[['Income']])

**# Standard Scaling for 'Age'**

standard\_scaler = StandardScaler()

data['Age\_Standardized'] = standard\_scaler.fit\_transform(data[['Age']]) print("\nAfter Scaling:")

print(data)

**# Binarization for 'Income' with a threshold of 75,000**

binarizer = Binarizer(threshold=75000)

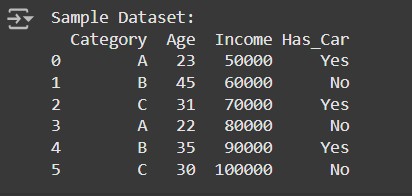
data['Income\_Binary'] = binarizer.fit\_transform(data[['Income']]) print("\nAfter Binarization:")

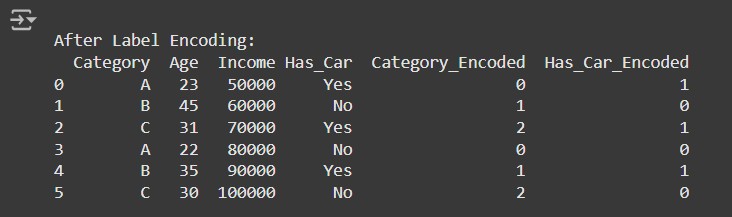
print(data)

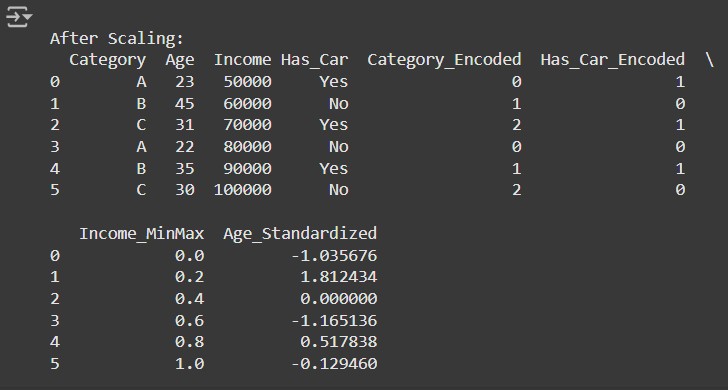
1. **Save the Processed Dataset**

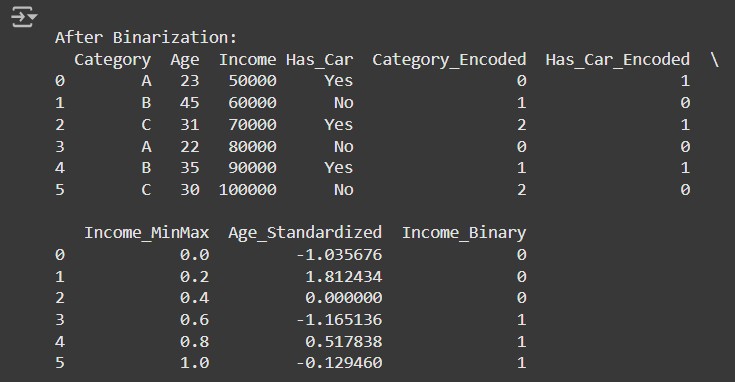
**# Save the processed dataset** data.to\_csv('processed\_data.csv', index=False) print("\nProcessed dataset saved as 'processed\_data.csv'")

**Output :**









**Practical 2 : Testing Hypothesis**

**Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a. CSV file and generate the final specific hypothesis. (Create your dataset)**

**CODE :**

import pandas as pd

**# Step 1: Create the Dataset and Load It**

data = {'Outlook': ['Sunny', 'Sunny', 'Overcast', 'Rainy', 'Rainy', 'Rainy', 'Overcast', 'Sunny', 'Sunny', 'Rainy'],

'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Cool', 'Mild', 'Cool', 'Mild'],

'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'Normal'],

'Wind': ['Weak', 'Strong', 'Weak', 'Weak', 'Weak', 'Strong', 'Strong', 'Weak', 'Weak', 'Weak'],

'PlayTennis': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes']

}

**# Load dataset into a pandas DataFrame**

df = pd.DataFrame(data)

**# Step 2: Implementing the FIND-S Algorithm**

def find\_s\_algorithm(data):

**# Get the positive examples (PlayTennis = 'Yes')**

    positive\_examples = data[data['PlayTennis'] == 'Yes']

**# Initialize hypothesis with the first positive example (most specific)**

    hypothesis = positive\_examples.iloc[0].drop('PlayTennis')

**# Loop through the rest of the positive examples and generalize the hypothesis**

    for index, row in positive\_examples.iterrows():

        for feature in hypothesis.index:

            if hypothesis[feature] != row[feature]:

                hypothesis[feature] = '?'

    return hypothesis

**Step 3: Apply FIND-S to the dataset**

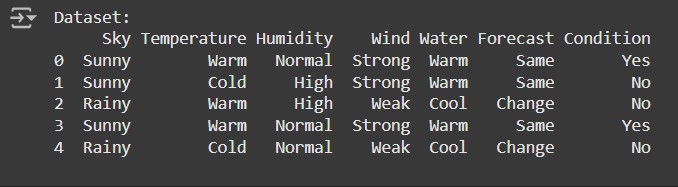
hypothesis = find\_s\_algorithm(df)

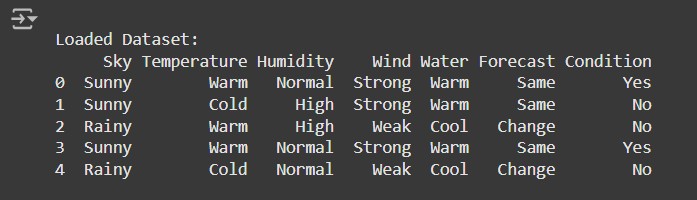
**# Display the final specific hypothesis**

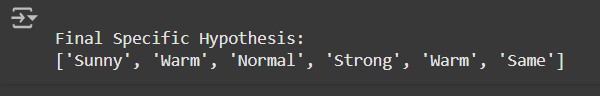
print("The most specific hypothesis is:")

print(hypothesis)

**Output :**







**Practical 3 : Linear Models**

**3a. Simple Linear Regression**

Fit a linear regression model on a dataset. Interpret coefficients, make predictions, and evaluate performance using metrics like R-squared and MSE

**Code :**

**Step 1: Import Libraries # Import required libraries** import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

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

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

**Step 2: Create a Dataset and Save as CSV**

**# Create a sample dataset**

data = {

'House\_Size': [750, 800, 850, 900, 1000, 1100, 1200, 1300, 1400, 1500],

'Price': [150000, 160000, 165000, 170000, 180000, 190000, 200000, 210000, 220000, 230000]

}

**# Convert the dataset into a DataFrame**

df = pd.DataFrame(data)

**# Save to CSV file** df.to\_csv('house\_prices.csv', index=False)

**# Display the dataset** print("Dataset:") print(df)

**Step 3: Load the Dataset**

**# Load the dataset**

dataset = pd.read\_csv('house\_prices.csv')

**# Display the first few rows** print("\nLoaded Dataset:") print(dataset.head())

**Step 4: Split the Dataset into Training and Test Sets**

**# Features and target variable**

X = dataset[['House\_Size']]  **# Feature: House size**

y = dataset['Price'] **# Target: Price**

**# Split data into training and testing sets (80% train, 20% test)**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42) print("\nTraining and Testing Data Sizes:")

print("Training Data Size:", X\_train.shape[0])

print("Testing Data Size:", X\_test.shape[0])

**Step 5: Fit a Linear Regression Model**

**# Initialize and fit the linear regression model**

model = LinearRegression()

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

**# Display the coefficients**

print("\nModel Coefficients:")

print("Slope (m):", model.coef\_[0]) print("Intercept (b):", model.intercept\_)

**Step 6: Make Predictions**

**# Predict on the test set**

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

**# Display predictions**

print("\nPredictions on Test Data:")

print("Actual Prices:", y\_test.values)

print("Predicted Prices:", y\_pred)

**Step 7: Evaluate the Model**

**# Calculate evaluation metrics**

mse = mean\_squared\_error(y\_test, y\_pred) r2 = r2\_score(y\_test, y\_pred)

**# Display metrics**

print("\nModel Performance Metrics:") print("Mean Squared Error (MSE):", mse) print("R-squared (R²):", r2)

**Step 8: Visualize the Results # Scatter plot of the training data** plt.scatter(X\_train, y\_train, color='blue', label='Training Data')

**# Plot the regression line**

plt.plot(X\_train, model.predict(X\_train), color='red', label='Regression Line')

**# Scatter plot of the test data**

plt.scatter(X\_test, y\_test, color='green', label='Test Data')

plt.title("Simple Linear Regression")

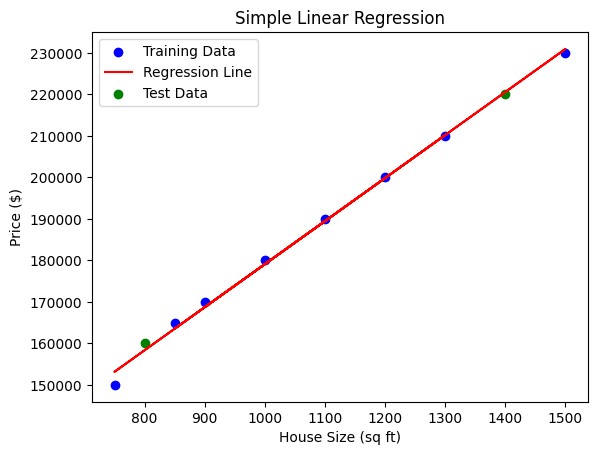
plt.xlabel("House Size (sq ft)")

plt.ylabel("Price ($)")

plt.legend()

plt.show()

**Output :**



**3b. Multiple Linear Regression :**

Extend linear regression to multiple feature. Handle feature selection and potential multicollinearity

**Code :**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

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

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

from statsmodels.stats.outliers\_influence import variance\_inflation\_factor from sklearn.preprocessing import LabelEncoder

**# Import LabelEncoder**

from sklearn.impute import SimpleImputer

**# Load dataset**

from google.colab import files uploaded = files.upload() **# Upload your CSV file**

**# Read the CSV file**

data =pd.read\_csv(list(uploaded.keys())[0])

**# Display the first few rows**

print(data.head())

**# Check for null values and basic statistics**

print(data.info())

print(data.describe())

**# Define a function to calculate VIF**

def calculate\_vif(df):

**# Select only numeric features for VIF calculation**

numeric\_df = df.select\_dtypes(include=np.number)

**# Drop rows with infinite or missing values**

numeric\_df = numeric\_df.replace([np.inf, -np.inf], np.nan).dropna()

vif\_data = pd.DataFrame()

vif\_data["feature"] = numeric\_df.columns

vif\_data["VIF"] = [variance\_inflation\_factor(numeric\_df.values, i)

for i in range(numeric\_df.shape[1])]

return vif\_data

**# Selecting features and target variable**

X = data.drop("Survived", axis=1)

# Changed 'y' to 'Survived' y = data["Survived"]

**# Handle categorical features (e.g., using Label Encoding)**

for col in X.select\_dtypes(include=['object']).columns:

le = LabelEncoder()

X[col] = le.fit\_transform(X[col])

**# Impute missing values using the mean (you can choose other strategies)**

imputer = SimpleImputer(strategy='mean')

**# Create an imputer instance**

X = pd.DataFrame(imputer.fit\_transform(X), columns=X.columns)

**# Impute and update X**

**# Calculate VIF for initial features**

print("VIF before handling multicollinearity:")

print(calculate\_vif(X)) **# Call the modified function**

**# Drop features based on VIF analysis (example: drop 'X1' if VIF is high)**

**# Check if the column exists before dropping**

if 'X1' in X.columns:

X = X.drop("X1", axis=1) # Replace 'X1' with the actual high VIF feature name

else:

print("Column 'X1' not found in the DataFrame.")

**# Recalculate VIF**

print("VIF after handling multicollinearity:")

print(calculate\_vif(X))

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

**# Initialize and fit the model**

model = LinearRegression()

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

**# Get coefficients and intercept**

print("Coefficients:", model.coef\_)

print("Intercept:", model.intercept\_)

**# Predictions**

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

**# Evaluation metrics**

rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred)) r2 = r2\_score(y\_test, y\_pred) print(f"RMSE: {rmse}")

print(f"R^2: {r2}")

from sklearn.feature\_selection import RFE

**# Recursive Feature Elimination**

rfe = RFE(estimator=LinearRegression(), n\_features\_to\_select=5)

# Adjust features

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

**# Selected features**

print("Selected Features:", X.columns[rfe.support\_])

**# Scatter plot of actual vs predicted values**

plt.scatter(y\_test, y\_pred) plt.xlabel("Actual")

plt.ylabel("Predicted")

plt.title("Actual vs Predicted")

plt.show()

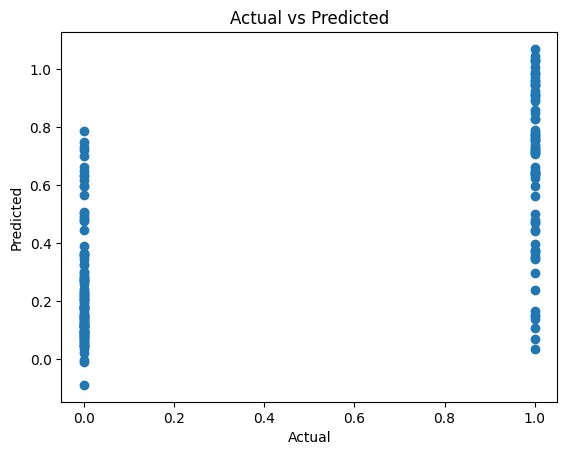
**# Residuals**

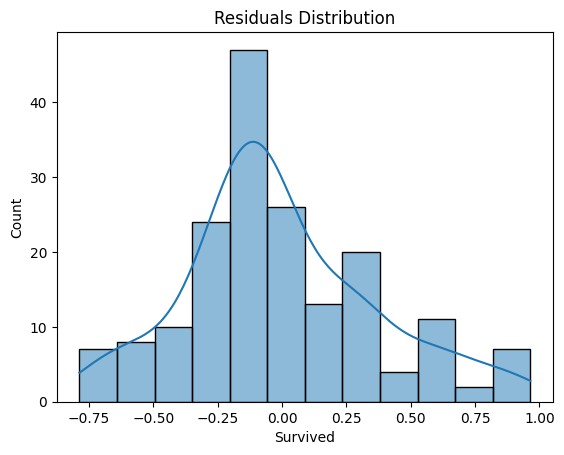
residuals = y\_test - y\_pred sns.histplot(residuals, kde=True)

plt.title("Residuals Distribution")

plt.show()

**Output :**





**3c. Regualarized Linear Models :**

Implement Regression variants like LASSO and Ridge on any generated dataset

**Code :**

1. **Set Up the Environment**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

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

from sklearn.linear\_model import Ridge, Lasso, ElasticNet

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.datasets import make\_regression

**# Set random seed for reproducibility**

np.random.seed(42)

2. **Generate a Synthetic Dataset**

**# Generate synthetic data**

X, y = make\_regression(n\_samples=1000,

**# Number of samples**

n\_features=10,

**# Number of features**

noise=15,

**# Add some noise**

random\_state=42

)

**# Convert to DataFrame for exploration**

data = pd.DataFrame(X, columns=[f"X{i}"

for i in range(1, 11)]) data["y"] = y

**# Display the first few rows**

print(data.head())

3. **Split the Dataset**

**# Split data into training and testing sets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data.drop("y", axis=1),

**# Features**

data["y"],

**# Target variable**

test\_size=0.2,

**# 20% for testing**

random\_state=42

)

4. **Train and Evaluate Ridge Regression**

**# Initialize Ridge Regression with a regularization parameter (alpha)**

ridge = Ridge(alpha=1.0)

**# Train the model**

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

**# Predictions**

ridge\_pred = ridge.predict(X\_test)

**# Evaluate Ridge Regression**

ridge\_rmse = np.sqrt(mean\_squared\_error(y\_test, ridge\_pred))

ridge\_r2 = r2\_score(y\_test, ridge\_pred)

print(f"Ridge RMSE: {ridge\_rmse}")

print(f"Ridge R^2: {ridge\_r2}")

1. **Train and Evaluate Lasso Regression**

**# Initialize Lasso Regression**

lasso = Lasso(alpha=0.1)

**# Train the model**

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

**# Predictions**

lasso\_pred = lasso.predict(X\_test)

**# Evaluate Lasso Regression**

lasso\_rmse = np.sqrt(mean\_squared\_error(y\_test, lasso\_pred))

lasso\_r2 = r2\_score(y\_test, lasso\_pred)

print(f"Lasso RMSE: {lasso\_rmse}")

print(f"Lasso R^2: {lasso\_r2}")

**# Features shrunk to**

zero print("Lasso Coefficients:", lasso.coef\_)

1. **Train and Evaluate ElasticNet Regression**

**# Initialize ElasticNet**

elastic\_net = ElasticNet(alpha=0.1, l1\_ratio=0.5) # l1\_ratio balances L1 and L2 penalties

**# Train the model**

elastic\_net.fit(X\_train, y\_train)

**# Predictions**

elastic\_net\_pred = elastic\_net.predict(X\_test)

**# Evaluate**

ElasticNet Regression elastic\_net\_rmse = np.sqrt(mean\_squared\_error(y\_test, elastic\_net\_pred)) elastic\_net\_r2 = r2\_score(y\_test, elastic\_net\_pred)

print(f"ElasticNet RMSE: {elastic\_net\_rmse}")

print(f"ElasticNet R^2: {elastic\_net\_r2}")

1. **Compare Results**

**# Collect metrics**

metrics = pd.DataFrame({

"Model": ["Ridge", "Lasso", "ElasticNet"],

"RMSE": [ridge\_rmse, lasso\_rmse, elastic\_net\_rmse],

"R^2": [ridge\_r2, lasso\_r2, elastic\_net\_r2]

})

print(metrics)

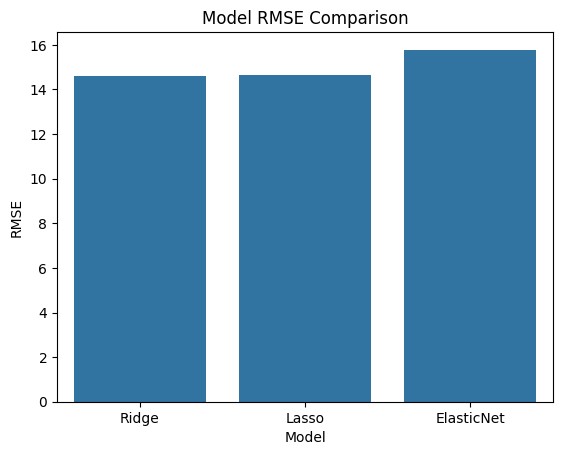
**# Plot RMSE comparison**

sns.barplot(data=metrics, x="Model", y="RMSE")

plt.title("Model RMSE Comparison")

plt.show()

**Output :**



**Practical 4 : Discriminative Models**

**4a. Logistic Regression :**

Perform binary classification using logistic regression. Calculate accuracy, precision, recall, and understand the ROC curve."

**Code :**

**Step 1: Import Required Libraries**

**# Import necessary libraries**

import numpy as np

import pandas as pd

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

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, roc\_curve, auc

import matplotlib.pyplot as plt

**Step 2: Prepare the Dataset**

from sklearn.datasets import make\_classification

**# Create a synthetic dataset**

X, y = make\_classification(n\_samples=1000, n\_features=10, n\_classes=2, random\_state=42)

**# Split data into training and testing sets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

**Step 3: Train the Logistic Regression Model**

**# Initialize the logistic regression model** logreg = LogisticRegression()

**# Train the model on the training data** logreg.fit(X\_train, y\_train)

**Step 4: Make Predictions**

**# Predict labels for the test set** y\_pred = logreg.predict(X\_test)

**# Predict probabilities for the ROC curve** y\_prob = logreg.predict\_proba(X\_test)[:, 1]

**Step 5: Evaluate the Model**

**# Calculate metrics**

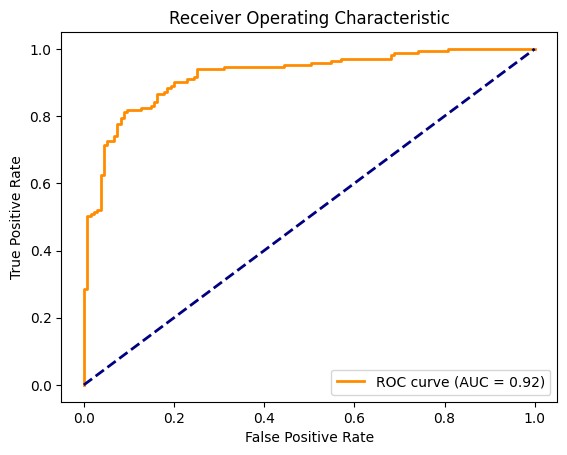
accuracy = accuracy\_score(y\_test, y\_pred)

precision = precision\_score(y\_test, y\_pred)

recall = recall\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

**Output :**



**4b .Implement and demonstrate k-nearest Neighbor algorithm. Read the training data from a .CSV file and build the model to classify a test sample. Print both correct and wrong predictions.**

**Code :**

**Step 1:** **Import Required Libraries # Import necessary libraries**

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split from sklearn.neighbors import KNeighborsClassifier from sklearn.metrics import accuracy\_score

from google.colab import files

**Step 2: Create or Upload the CSV File**

**# Check if the user wants to create a dataset or upload one**

print("Do you have a CSV file to upload? (yes/no)") response = input().lower()

if response == "yes":

uploaded = files.upload() filename = list(uploaded.keys())[0] else:

**# Create a synthetic dataset**

from sklearn.datasets import make\_classification

**# Generate synthetic data**

X,y=make\_classification(n\_samples=200,n\_features=5, n\_classes=2, random\_state=42**)**

**# Combine features and target into a single DataFrame**

data = pd.DataFrame(X, columns=[f"Feature\_{i}"

for i in range(X.shape[1])])

data['Target'] = y

**# Save the dataset to a CSV file**

filename = "synthetic\_data.csv"

data.to\_csv(filename, index=False)

print(f"Synthetic dataset saved as {filename}.")

**Step 3: Load the CSV File into a DataFrame**

# Load the dataset into a DataFrame

data = pd.read\_csv(filename)

**# Display the first few rows of the dataset**

print("Loaded Dataset:")

print(data.head())

**Step 4: Preprocess the Data**

**# Separate features (X) and labels (y)**

X = data.iloc[:, :-1].values # All columns except the last one y = data.iloc[:, -1].values # Last column as the target

**# Split the dataset into training and testing sets (80% train, 20% test)**

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

**Step 5: Train the k-NN Model # Initialize the k-NN model with k=3** knn = KNeighborsClassifier(n\_neighbors=3) **# Train the model on the training data** knn.fit(X\_train, y\_train)

**Step 6: Predict Test Samples # Predict the labels for the test set** y\_pred = knn.predict(X\_test)

**Step 7: Evaluate and Print Predictions # Calculate and display the accuracy** accuracy = accuracy\_score(y\_test, y\_pred) print(f"\nModel Accuracy: {accuracy:.2f}\n")

**# Display correct and incorrect predictions**

print("Correct Predictions:")

for i in range(len(y\_test)):

if y\_pred[i] == y\_test[i]:

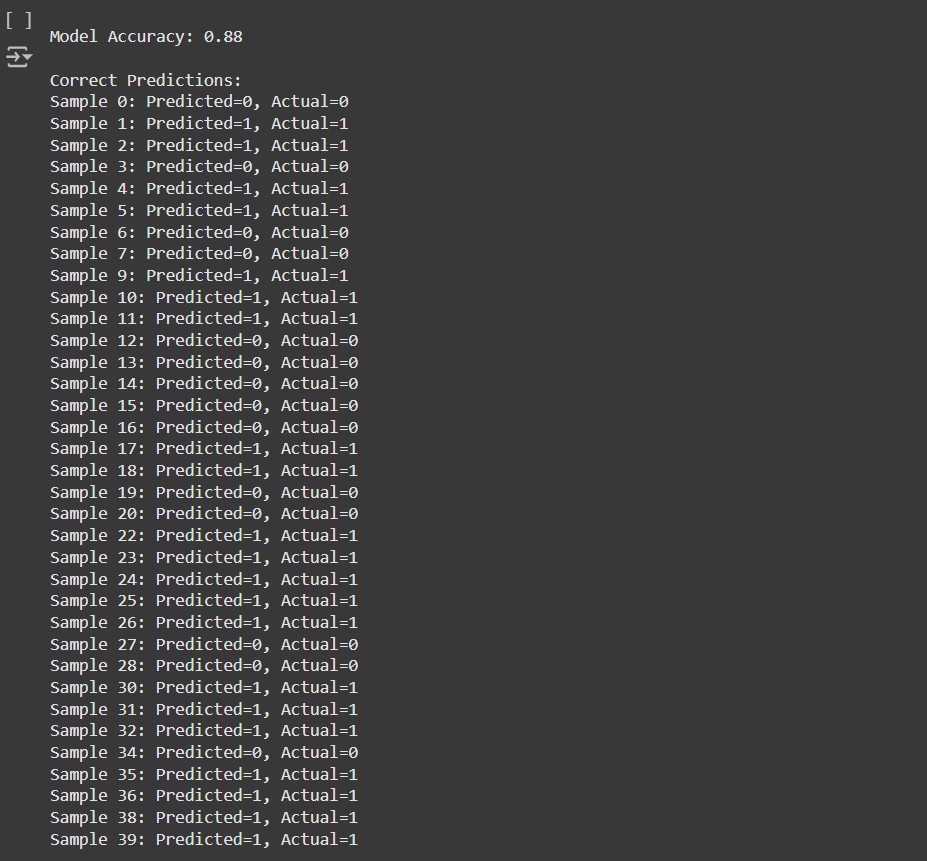
print(f"Sample {i}: Predicted={y\_pred[i]}, Actual={y\_test[i]}") print("\nIncorrect Predictions:")

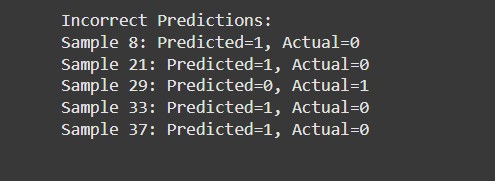
for i in range(len(y\_test)):

if y\_pred[i] != y\_test[i]:

print(f"Sample {i}: Predicted={y\_pred[i]}, Actual={y\_test[i]}")

**Output :**





**4c. Build a decision tree classifier or regressor. Control hyperparameters like tree depth to avoid overfitting. Visualize the tree.**

**Code :**

**Step 1: Import Required Libraries**

**# Import necessary libraries**

import pandas as pd

import numpy as np

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

from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor, plot\_tree

from sklearn.metrics import accuracy\_score, mean\_squared\_error

import matplotlib.pyplot as plt

from google.colab import files

**Step 2: Create or Upload the CSV File**

**# Check if the user wants to upload a file or generate one**

print("Do you have a CSV file to upload? (yes/no)") response = input().lower()

if response == "yes":

**# Upload the CSV file**

uploaded = files.upload()

filename = list(uploaded.keys())[0]

else:

**# Generate synthetic data (classification or regression)**

from sklearn.datasets import make\_classification, make\_regression

print("Choose a task: (1) Classification (2) Regression")

task = int(input())

if task == 1:

**# Generate synthetic classification data**

X, y = make\_classification(n\_samples=200, n\_features=5, random\_state=42)

task\_type = "classification"

else:

**# Generate synthetic regression data**

X, y = make\_regression(n\_samples=200, n\_features=5, random\_state=42)

task\_type = "regression"

**# Combine features and target into a single DataFrame**

data = pd.DataFrame(X, columns=[f"Feature\_{i}"

for i in range(X.shape[1])]) data['Target'] = y

**# Save the dataset to a CSV file**

filename = "synthetic\_data.csv"

data.to\_csv(filename, index=False)

print(f"Synthetic {task\_type} dataset saved as {filename}.")

**Step 3: Load the Dataset**

**# Load the dataset**

data =pd.read\_csv(filename)

**# Display the first few rows of the dataset** print("Dataset Preview:")

print(data.head())

**Step 4: Preprocess the Data**

**# Separate features and target**

X = data.iloc[:, :-1].values # All columns except the last one y = data.iloc[:, -1].values # Last column as the target

**# Split data 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)

**Step 5: Build the Decision Tree**

**# Define the tree depth to avoid overfitting** max\_depth = 3

**# Initialize the model**

if task\_type =="classification":

model = DecisionTreeClassifier(max\_depth=max\_depth, random\_state=42)

else:

model = DecisionTreeRegressor(max\_depth=max\_depth, random\_state=42)

**# Train the model** model.fit(X\_train, y\_train)

**Step 6: Make Predictions**

**# Predict on the test set**

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

**# Evaluate the model**

if task\_type == "classification":

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

else:

mse = mean\_squared\_error(y\_test, y\_pred) print(f"Mean Squared Error: {mse:.2f}")

**Step 7: Visualize the Tree**

**# Visualize the decision tree**

plt.figure(figsize=(12, 8))

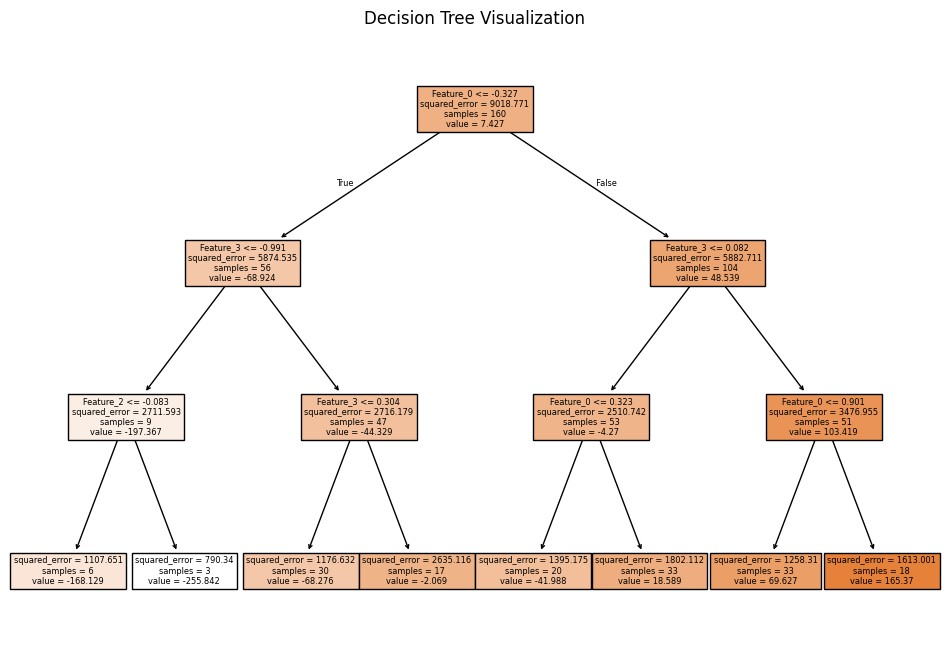
plot\_tree(model, feature\_names=data.columns[:-1], class\_names=str(np.unique(y))

if task\_type == "classification" else None, filled=True)

plt.title("Decision Tree Visualization")

plt.show()

**Output :**



**4d. Implement a Support Vector Machine for any relevant dataset.**

**Code:**

**Step 1: Import Required Libraries**

**# Import necessary libraries**

import pandas as pd

import numpy as np

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

from sklearn.svm import SVC

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

from google.colab import files

**Step 2: Create or Upload a Dataset**

**# Check if the user wants to upload a file or generate one**

print("Do you have a CSV file to upload? (yes/no)") response = input().lower()

if response == "yes":

**# Upload the CSV file**

uploaded = files.upload()

filename = list(uploaded.keys())[0]

else:

**# Generate synthetic classification data**

from sklearn.datasets import make\_classification

X, y = make\_classification(n\_samples=200, n\_features=5, n\_classes=2, random\_state=42)

**# Combine features and target into a DataFrame**

data = pd.DataFrame(X, columns=[f"Feature\_{i}"

for i in range(X.shape[1])])

data['Target'] = y

**#Save the synthetic dataset to a CSV file**

filename="synthetic\_data.csv" data.to\_csv(filename,index=False) print(f"Synthetic dataset saved as {filename}.")

**Step 3: Load the Dataset**

**# Load the dataset into a DataFrame** data = pd.read\_csv(filename)

**# Display the first few rows of the dataset**

print("Dataset Preview:")

print(data.head())

**Step 4: Preprocess the Data**

**# Separate features (X) and target (y)**

X = data.iloc[:, :-1].values # All columns except the last one y = data.iloc[:, -1].values # Last column as the target

**# Split the dataset into training (80%) and testing (20%) sets**

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

**Step 5: Train the Support Vector Machine**

**# Initialize the SVM model (use RBF kernel as default)**

svm\_model = SVC(kernel='rbf', C=1.0, gamma='scale', random\_state=42)

**# Train the SVM model on the training data** svm\_model.fit(X\_train, y\_train)

**Step 6: Make Predictions**

**# Predict the labels for the test set** y\_pred = svm\_model.predict(X\_test)

**Step 7: Evaluate the Model**

**# Calculate and print the accuracy**

accuracy = accuracy\_score(y\_test, y\_pred) print(f"Model Accuracy: {accuracy:.2f}")

**# Print a detailed classification report** print("\nClassification Report:") print(classification\_report(y\_test, y\_pred))

**Step 8: Visualize the Decision Boundary (Optional for 2D Data)**

import matplotlib.pyplot as plt

**# Generate 2D synthetic data**

from sklearn.datasets import make\_blobs

X, y = make\_blobs(n\_samples=100, centers=2, random\_state=42, cluster\_std=1.5)

**# Fit the SVM on this data** svm\_model.fit(X, y)

**#Plot the decision boundary**

plt.figure(figsize=(8, 6))

plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolor='k')

**# Create a grid to evaluate the model**

xx, yy = np.meshgrid (np.linspace(X[:, 0].min(), X[:, 0].max(), 100), np.linspace(X[:, 1].min(), X[:, 1].max(), 100))

Z = svm\_model.decision\_function(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

**# Plot the decision boundary and margins**

plt.contour(xx, yy, Z, levels=[-1, 0, 1], linestyles=['--', '-', '--'], colors='k')

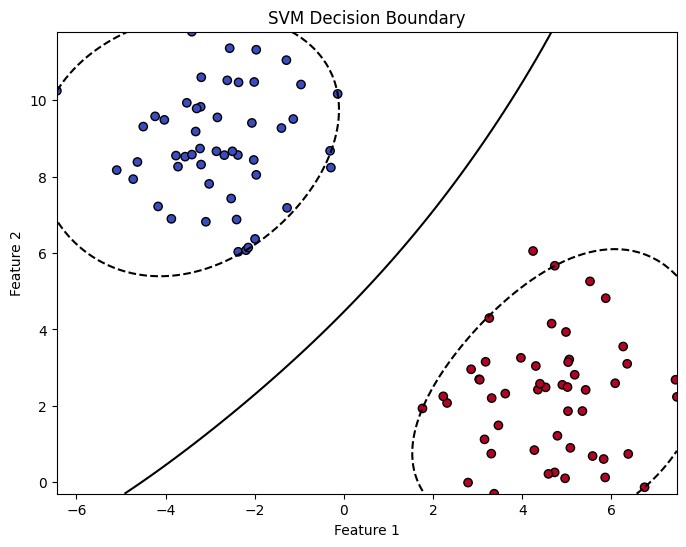
plt.title("SVM Decision Boundary")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

**Output :**



**4e. Train a random forest ensemble. Experiment with the number of trees and feature sampling. Compare performance to a single decision tree.**

**Code :**

**Step 1: Import Required Libraries # Import necessarylibraries**

import pandas as pd

import numpy as np

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

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

from google.colab import files

**Step 2: Create or Upload a Dataset**

**# Check if the user wants to upload a file or generate one**

print("Do you have a CSV file to upload? (yes/no)")

response = input().lower()

if response == "yes":

**# Upload the CSV file**

uploaded = files.upload()

filename = list(uploaded.keys())[0]

else:

**# Generate synthetic classification data**

from sklearn.datasets import make\_classification

X, y = make\_classification(n\_samples=300, n\_features=10, n\_classes=2, random\_state=42)

**# Combine features and target into a DataFrame**

data = pd.DataFrame(X, columns=[f"Feature\_{i}"

for i in range(X.shape[1])])

data['Target'] = y

**# Save the synthetic dataset to a CSV file**

filename = "synthetic\_data.csv"

data.to\_csv(filename, index=False)

print(f"Synthetic dataset saved as {filename}.")

**Step 3: Load the Dataset**

**# Load the dataset**

data =pd.read\_csv(filename)

**# Display the first few rows of the dataset**

print("Dataset Preview:") print(data.head())

**Step 4: Preprocess the Data**

**# Separate features (X) and target (y)**

X = data.iloc[:, :-1].values # All columns except the last one y = data.iloc[:, -1].values # Last column as the target

**# Split the dataset into training (80%) and testing (20%) sets**

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

**Step 5: Train a Single Decision Tree Classifier**

**# Initialize and train the Decision Tree model**

decision\_tree =DecisionTreeClassifier(random\_state=42)

decision\_tree.fit(X\_train, y\_train)

**# Predict and evaluate**

y\_pred\_tree = decision\_tree.predict(X\_test)

accuracy\_tree = accuracy\_score(y\_test, y\_pred\_tree)

print(f"Decision Tree Accuracy: {accuracy\_tree:.2f}")

**Step 6: Train a Random Forest Classifier**

**# Initialize the Random Forest model with hyperparameter tuning**

random\_forest = RandomForestClassifier(n\_estimators=100, max\_features='sqrt', random\_state=42)

**# Train the model** random\_forest.fit(X\_train, y\_train)

**# Predict and evaluate**

y\_pred\_rf = random\_forest.predict(X\_test)

accuracy\_rf = accuracy\_score(y\_test, y\_pred\_rf)

print(f"Random Forest Accuracy (100 trees, sqrt features): {accuracy\_rf:.2f}")

**Step 7: Experiment with Random Forest Hyperparameters**

**# Experiment with fewer trees and different feature sampling**

rf\_experiment = RandomForestClassifier(n\_estimators=50, max\_features=3, random\_state=42)

rf\_experiment.fit(X\_train, y\_train)

**# Predict and evaluate**

y\_pred\_rf\_exp = rf\_experiment.predict(X\_test)

accuracy\_rf\_exp = accuracy\_score(y\_test, y\_pred\_rf\_exp)

print(f"Random Forest Accuracy (50 trees, max\_features=3): {accuracy\_rf\_exp:.2f}")

**Step 8: Compare the Models**

print("\nModel Comparison:")

print(f"Decision Tree Accuracy: {accuracy\_tree:.2f}")

print(f"Random Forest Accuracy (100 trees): {accuracy\_rf:.2f}")

print(f"Random Forest Accuracy (50 trees, max\_features=3): {accuracy\_rf\_exp:.2f}")

**Step 9: Visualize Feature Importance (Optional)** import matplotlib.pyplot as plt

**# Extract feature importance from the Random Forest model** feature\_importances = random\_forest.feature\_importances\_

**# Plot the feature importance**

plt.figure(figsize=(10, 6))

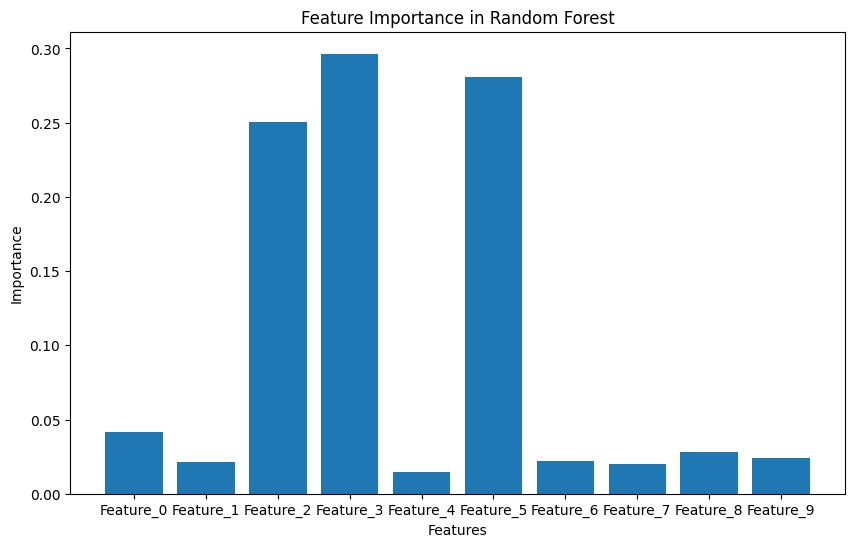
plt.bar(range(len(feature\_importances)), feature\_importances, tick\_label=data.columns[:-1]) plt.title("Feature Importance in Random Forest")

plt.xlabel("Features")

plt.ylabel("Importance")

plt.show()

**Output :**



**4f. Implement a gradient boosting machine (e.g., XGBoost). Tune hyperparameters and explore feature importance.**

**Code :**

**Step 1: Import Required Libraries**

**# Import necessary libraries**

import pandas as pd

import numpy as np

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

from sklearn.metrics import accuracy\_score, classification\_report from xgboost import XGBClassifier, plot\_importance

import matplotlib.pyplot as plt

from google.colab import files

**Step 2: Create or Upload a Dataset**

**# Check if the user wants to upload a file or generate one** print("Do you have a CSV file to upload? (yes/no)") response = input().lower()

if response == "yes":

# Upload the CSV file

uploaded=files.upload()

filename = list(uploaded.keys())[0]

else:

**# Generate synthetic classification data**

from sklearn.datasets import make\_classification

X, y = make\_classification(n\_samples=300, n\_features=10, n\_classes=2, random\_state=42)

**# Combine features and target into a DataFrame**

data = pd.DataFrame(X, columns=[f"Feature\_{i}"

for i in range(X.shape[1])])data['Target'] = y

**# Save the synthetic dataset to a CSV file**

filename="synthetic\_data.csv" data.to\_csv(filename,index=False) print(f"Synthetic dataset saved as {filename}.")

**Step 3: Load the Dataset**

**# Load the dataset**

data = pd.read\_csv(filename)

**# Display the first few rows of the dataset** print("Dataset Preview:") print(data.head())

**Step 4: Preprocess the Data**

**# Separate features (X) and target (y)**

X = data.iloc[:, :-1].values # All columns except the last one y = data.iloc[:, -1].values # Last column as the target

**# Split the dataset into training (80%) and testing (20%) sets**

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

**Step 5: Train a Basic XGBoost Model**

**# Initialize and train the XGBoost model with default parameters** xgb = XGBClassifier(random\_state=42)

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

**# Predict and evaluate the model**

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

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"XGBoost Accuracy (Default Parameters): {accuracy:.2f}")

**Step 6: Tune Hyperparameters with GridSearchCV**

**# Define a grid of hyperparameters**

param\_grid = { 'n\_estimators': [50, 100, 150],'learning\_rate': [0.01, 0.1, 0.2], 'max\_depth': [3, 5, 7] }

**# Initialize GridSearchCV**

grid\_search = GridSearchCV(estimator=XGBClassifier(random\_state=42),param\_grid=param\_grid, scoring='accuracy', cv=3,verbose=1)

**# Fit the model with grid search** grid\_search.fit(X\_train, y\_train)

**# Best parameters from GridSearch**

print(f"Best Parameters: {grid\_search.best\_params\_}")

**# Train the final model with the best parameters**

best\_xgb = grid\_search.best\_estimator\_

**# Predict and evaluate**

y\_pred\_best = best\_xgb.predict(X\_test)

accuracy\_best = accuracy\_score(y\_test, y\_pred\_best)

print(f"XGBoost Accuracy (Tuned Parameters): {accuracy\_best:.2f}")

**Step 7: Explore Feature Importance**

**# Plot feature importance for the tuned model**

plt.figure(figsize=(10, 6))

plot\_importance(best\_xgb,importance\_type='weight', xlabel="Importance", ylabel="Features")

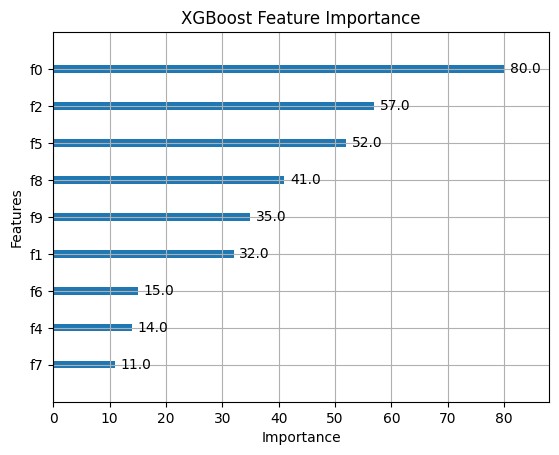
plt.title("XGBoost Feature Importance")

plt.show()

**Step 8: Evaluate the Model**

**# Print a detailed classification report** print("Classification Report:") print(classification\_report(y\_test, y\_pred\_best))

**Output :**



**Practical 5**

**5a. Implement and demonstrate the working of a Naive Bayesian classifier using a sample data set. Build the model to classify a test sample.**

**Step 1: Import Required Libraries**

**# Import necessary libraries**

import pandas as pd

import numpy as np

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

from sklearn.metrics import accuracy\_score,classification\_report from sklearn.naive\_bayes import GaussianNB

from google.colab import files

**Step 2: Create or Upload a Dataset**

**# Ask if the user wants to upload a file or generate one**

print("Do you have a CSV file to upload? (yes/no)")

response = input().lower()

if response == "yes":

**# Upload the CSV file**

uploaded =files.upload()

filename = list(uploaded.keys())[0]

else:

**# Generate synthetic classification data**

from sklearn.datasets import make\_classification

X, y = make\_classification(n\_samples=300, n\_features=8, n\_classes=2, random\_state=42)

**# Combine features and target into a DataFrame**

data = pd.DataFrame(X, columns=[f"Feature\_{i}" for i in range(X.shape[1])])

data['Target'] = y

**# Save the synthetic dataset to a CSV file**  filename="synthetic\_naive\_bayes\_data.csv" data.to\_csv(filename,index=False) print(f"Synthetic dataset saved as {filename}.")

**Step 3: Load the Dataset**

**# Load the dataset**

data =pd.read\_csv(filename)

**# Display the first few rows of the dataset**

print("Dataset Preview:")

print(data.head())

**Step 4: Preprocess the Data**

**# Separate features (X) and target (y)**

X = data.iloc[:, :-1].values # All columns except the last one y = data.iloc[:, -1].values # Last column as the target

**# Split the dataset into training (80%) and testing (20%) sets**

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

**Step 5: Train a Naive Bayes Classifier**

**# Initialize the Gaussian Naive Bayes classifier**

naive\_bayes = GaussianNB()

**# Train the model**

naive\_bayes.fit(X\_train, y\_train)

**Step 6: Make Predictions and Evaluate**

**# Predict on the test set**

y\_pred =naive\_bayes.predict(X\_test)

**# Evaluate the model**

accuracy = accuracy\_score(y\_test, y\_pred) print(f"Naive Bayes Accuracy: {accuracy:.2f}")

**# Detailed classification report**

print("Classification Report:")

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

**Step 7: Test the Model with a Custom Sample**

**# Define a sample test input (replace with meaningful values based on your dataset)**

test\_sample = [X\_test[0]]

**# Taking the first test sample for demonstration**

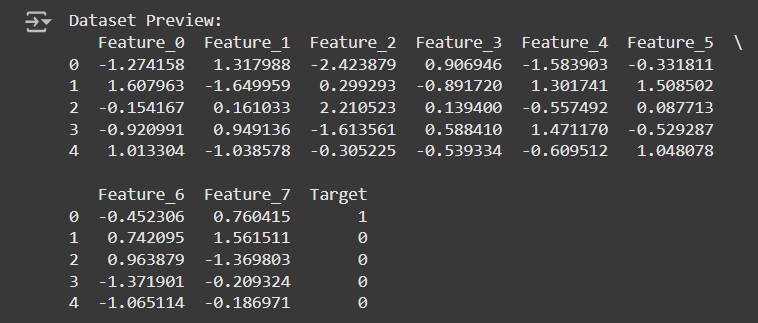
**# Predict the class for the test sample**

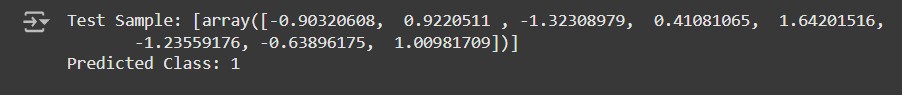
predicted\_class = naive\_bayes.predict(test\_sample)

print(f"Test Sample: {test\_sample}")

print(f"Predicted Class: {predicted\_class[0]}")

**Output :**





**5b. Implement Hidden Markov Models using hmmlearn**

**Code :**

**Step 1: Install Required Libraries**

**# Install hmmlearn**

!pip install hmmlearn

**Step 2: Import Required Libraries**

**# Import necessary libraries** import numpy as np

import pandas as pd

from hmmlearn import hmm

import matplotlib.pyplot as plt

**Step 3: Create or Load a Dataset**

**# Generate synthetic observable data**

np.random.seed(42)

**# Create a sequence of observations and hidden states**

observations = np.random.choice(['A', 'B', 'C'], size=100, p=[0.5, 0.3, 0.2])

hidden\_states = np.random.choice(['X', 'Y'], size=100, p=[0.6, 0.**4])**

**# Save the data in a DataFrame for analysis**

data = pd.DataFrame({'Observations': observations, 'Hidden States': hidden\_states}) print("Generated Data:")

print(data.head())

**Step 4: Encode Observations**

**# Encode the observations into integers**

observation\_mapping = {obs: idx for idx, obs in enumerate(np.unique(observations))} encoded\_observations = np.array([observation\_mapping[obs] for obs in observations]) **# Print the mapping**

print("Observation Encoding:")

print(observation\_mapping)

**Step 5: Initialize and Configure the HMM**

**# Initialize the HMM model**

n\_states = 2 **# Number of hidden states**

n\_observations = len(observation\_mapping)

**# Number of unique observations**

model = hmm.MultinomialHMM(n\_components=n\_states, random\_state=42, n\_iter=100, tol=0.01)

**# Define start probabilities (initial distribution of states)** start\_probs = np.array([0.6, 0.4]) # Assumed probabilities model.startprob\_ = start\_probs

**# Define transition probabilities between states** trans\_probs = np.array([

[0.7, 0.3], # From state X

[0.4, 0.6], # From state Y])

model.transmat\_ = trans\_probs

**# Define emission probabilities (probability of observations given states)** emission\_probs = np.array([

[0.5, 0.4, 0.1], # State X emits A, B, C

[0.2, 0.3, 0.5], # State Y emits A, B, C

])

model.emissionprob\_ = emission\_probs

**# Print the configured model parameters** print("Start Probabilities:", model.startprob\_) print("Transition Matrix:", model.transmat\_) print("Emission Probabilities:", model.emissionprob\_)

**Step 6: Train the Model**

**# Reshape the data for HMM (requires 2D array)** encoded\_observations = encoded\_observations.reshape(-1, 1)

**# Fit the model** model.fit(encoded\_observations)

**# Predict hidden states for the observations** predicted\_states = model.predict(encoded\_observations)

**# Print the predicted states** print("Predicted States:") print(predicted\_states)

**Step 7: Visualize the Results**

**# Map predicted states back to their original labels**

state\_mapping = {0: 'X', 1: 'Y'}

predicted\_state\_labels = [state\_mapping[state] for state in predicted\_states]

**# Add predicted states to the DataFrame** data['Predicted States'] = predicted\_state\_labels

**# Display the first few rows with predicted states**

print("Data with Predicted States:")

print(data.head())

**# Plot the observations and predicted states**

plt.figure(figsize=(12, 6))

plt.plot(data['Observations'], label='Observations', marker='o', linestyle='-', alpha=0.7) plt.plot(data['Predicted States'], label='Predicted States', marker='x', linestyle='--', alpha=0.7) plt.legend()

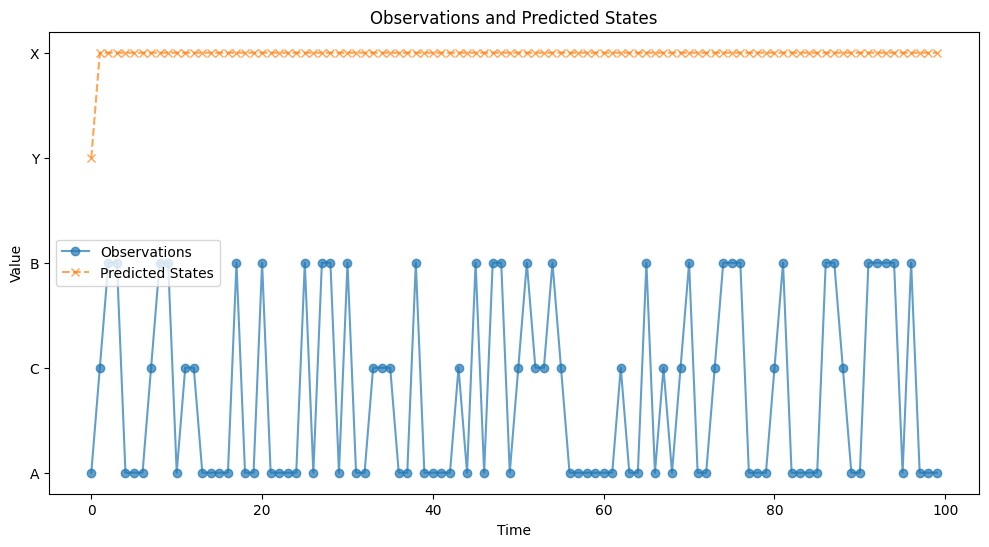
plt.title("Observations and Predicted States")

plt.xlabel("Time")

plt.ylabel("Value")

plt.show()

**Output :**



**Practical 6 : Probabilistic Model**

**6a. Implement Bayesian Linear Regression to explore prior and posterior distribution.**

Bayesian Linear Regression is a probabilistic approach to linear regression that incorporates uncertainty in the model parameters. Instead of estimating point values for parameters (as in traditional linear regression), we estimate distributions over the parameters.

**Code :**

**Step 1: Install Required Libraries**

**# Install necessary libraries**

!pip install matplotlib seaborn scikit-learn

**Step 2: Import Required Libraries**

**# Import necessary libraries**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.linear\_model import BayesianRidge from sklearn.model\_selection import train\_test\_split from sklearn.metrics import mean\_squared\_error from google.colab import files

**Step 3: Create or Upload a Dataset**

**# Upload a CSV file if you have one**

print("Do you have a CSV file to upload? (yes/no)")

response = input().lower()

if response == "yes":

# Upload the CSV file

uploaded = files.upload()

filename = list(uploaded.keys())[0] else:

**# Generate synthetic data for demonstration** np.random.seed(42)

X = np.random.rand(100, 1) \* 10

**# Random data between 0 and 10**

y = 2 \* X + 1 + np.random.randn(100, 1) \* 2

**# y = 2x + 1 with some noise**

**# Convert to a DataFrame**

data = pd.DataFrame(np.hstack((X, y)), columns=["X", "y"])

**# Save to CSV for convenience**

filename="synthetic\_data.csv" data.to\_csv(filename,index=False) print(f"Synthetic dataset saved as {filename}.")

**Step 4: Load and Explore the Data**

**# Load the dataset (for CSV file)**

data = pd.read\_csv(filename)

**# Display first few rows**

print("Dataset Preview:")

print(data.head())

**Step 5: Preprocess the Data**

**# Separate features (X) and target (y)**

X = data["X"].values.reshape(-1, 1) # Feature matrix y = data["y"].values # Target vector

**# Split the dataset into training (80%) and testing (20%) sets**

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

**Step 6: Implement Bayesian Linear Regression Model**

**# Initialize the BayesianRidge model (which implements Bayesian Linear Regression)** bayesian\_regressor = BayesianRidge()

**# Fit the model on the training data**

bayesian\_regressor.fit(X\_train, y\_train)

**# Predict on the test data**

y\_pred = bayesian\_regressor.predict(X\_test)

**Step 7: Visualize the Prior and Posterior Distributions**

**# Plot the prior and posterior distributions of the parameters**

fig, ax = plt.subplots(1, 2, figsize=(12, 6))

**# Plot prior distribution (assuming the model starts with a standard prior)** ax[0].set\_title("Prior Distribution (Assumed)") ax[0].hist(np.random.normal(0, 1, 1000), bins=50, alpha=0.7, color='blue', label="Prior") ax[0].legend()

**# Plot posterior distribution (after model fitting)**

ax[1].set\_title("Posterior Distribution (After Fitting)") ax[1].hist(bayesian\_regressor.coef\_, bins=50, alpha=0.7, color='green', label="Posterior") ax[1].legend()

plt.show()

**Step 8: Evaluate the Model Performance # Calculate the Mean Squared Error (MSE)** mse = mean\_squared\_error(y\_test, y\_pred) print(f"Mean Squared Error (MSE): {mse:.2f}")

**Step 9: Visualize the Fit of the Model**

**# Plot the true values and the predicted values**

plt.figure(figsize=(8, 6))

plt.scatter(X\_test, y\_test, color="blue", label="True values") plt.plot(X\_test, y\_pred, color="red", label="Predicted values", linewidth=2)

plt.title("Bayesian Linear Regression: True vs Predicted Values") plt.xlabel("X")

plt.ylabel("y")

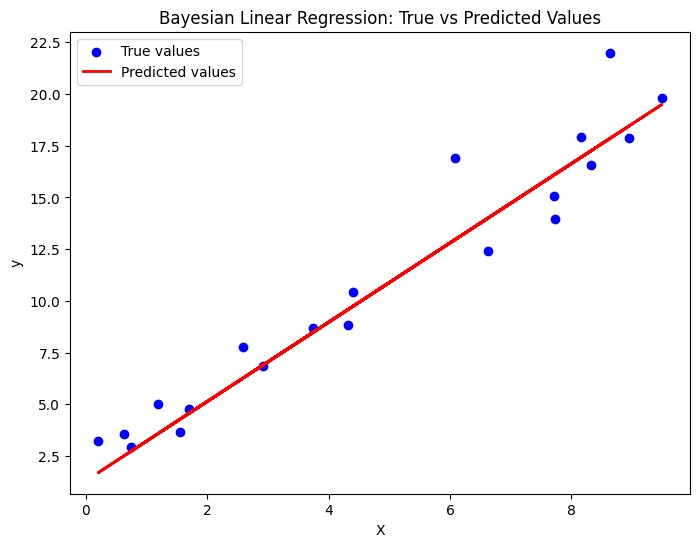
plt.legend()

plt.show()

**Mean Squared Error (MSE): 3.9**

**Output :**





**6b. Implement Gaussian Mixture Models for density estimation and unsupervised clustering.**

**Code :**

**Step 1: Install Required Libraries**

**# Install required libraries**

!pip install matplotlib seaborn scikit-learn

**Step 2: Import Required Libraries**

**# Import necessary libraries**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.mixture import GaussianMixture

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

from google.colab import files

**Step 3: Create or Upload a Dataset**

**#Ask if the user has a CSV file to upload**

print("Do you have a CSV file to upload? (yes/no)")

response = input().lower()

if response == "yes":

**# Upload the CSV file**

uploaded = files.upload()

filename = list(uploaded.keys())[0]

else:

**# Generate synthetic 2D data with two clusters for demonstration**

np.random.seed(42)

**# Generate data for two Gaussian distributions**

X1 = np.random.normal(loc=0, scale=1, size=(300, 2)) # Cluster 1: mean = 0, std = 1

X2 = np.random.normal(loc=5, scale=1, size=(300, 2)) # Cluster 2: mean = 5, std = 1  **# Stack the data to create a dataset**

X = np.vstack([X1, X2])

**# Create DataFrame to simulate the CSV file for consistency**

data = pd.DataFrame(X, columns=["Feature\_1", "Feature\_2"])

filename = "synthetic\_data.csv"

data.to\_csv(filename, index=False)

print(f"Synthetic dataset saved as {filename}.")

**Step 4: Load and Explore the Dataset**

**# Load the dataset (if CSV file is uploaded)**

data = pd.read\_csv(filename)

**# Display the first few rows**

print("Dataset Preview:")

print(data.head())

**# Plot the data to visualize its structure** sns.scatterplot(data=data, x="Feature\_1", y="Feature\_2") plt.title("Synthetic Data")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

**Step 5: Fit a Gaussian Mixture Model (GMM)**

**# Define the GMM model**

n\_components = 2 **# Number of Gaussian distributions (clusters)**

gmm = GaussianMixture(n\_components=n\_components, covariance\_type='full', random\_state=42)

**# Fit the GMM model to the data** gmm.fit(data)

**# Predict the cluster labels for each data point**

labels = gmm.predict(data)

**# Add the cluster labels to the dataset for visualization**

data['Cluster'] = labels

**# Plot the clustered data**

sns.scatterplot(data=data, x="Feature\_1", y="Feature\_2", hue="Cluster", palette="viridis", marker="o")

plt.title("Gaussian Mixture Model Clustering")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.legend()

plt.show()

**Step 6: Visualize the Gaussian Mixture Model (GMM) Components**

**# Extract the means and covariances of the Gaussian components**

means = gmm.means\_

covariances = gmm.covariances\_

**# Plot the GMM components on top of the data**

plt.figure(figsize=(8, 6)) **# Plot data points**

sns.scatterplot(data=data, x="Feature\_1", y="Feature\_2", hue="Cluster", palette="viridis", marker="o", s=60, alpha=0.7)

**# Plot the GMM ellipses** for mean, covar in zip(means, covariances):

**# Plot the Gaussian components as ellipses**

v, w = np.linalg.eigh(covar)

v = 2.0 \* np.sqrt(2.0) \* np.sqrt(v)

**# Scaling factor for the ellipse**

u = w[0] / np.linalg.norm(w[0])

**# Normalize the eigenvector**

angle = np.arctan(u[1] / u[0])

**# Create the ellipse**

angle = angle \* 180.0 / np.pi **# Convert to degrees**   
ellipse = plt.matplotlib.patches.Ellipse(mean, v[0], v[1], angle=angle, color='red', alpha=0.3) plt.gca().add\_patch(ellipse)

plt.title("GMM Clustering with Gaussian Components")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.legend()

plt.show()

**Step 7: Model Evaluation (Optional)**

**# Compute the log-likelihood of the data under the fitted GMM model**

log\_likelihood = gmm.score(data)

print(f"Log-Likelihood of the data: {log\_likelihood:.2f}")

**Step 8: Predict New Data Points**

**# Example of predicting the cluster for new data points** new\_data = np.array([[1.5, 2.5], [4.5, 5.5], [7.0, 8.0]])

new\_labels = gmm.predict(new\_data)

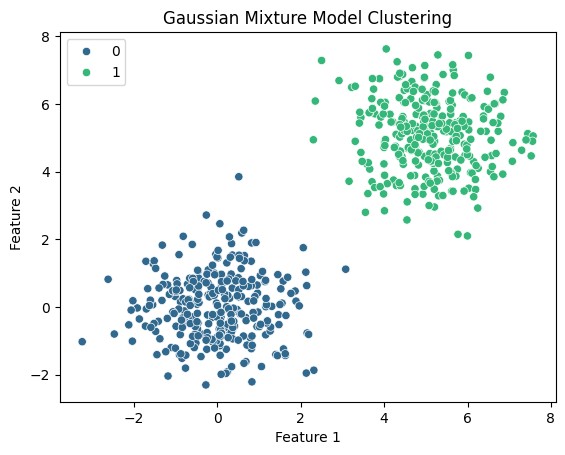
**# Print the predicted clusters for the new data points**

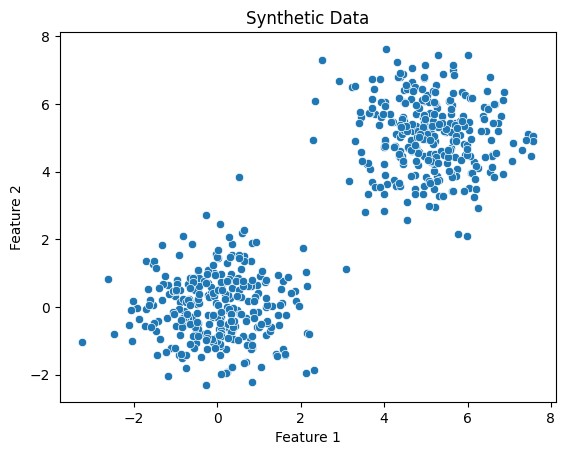
print("Predicted Clusters for New Data Points:")

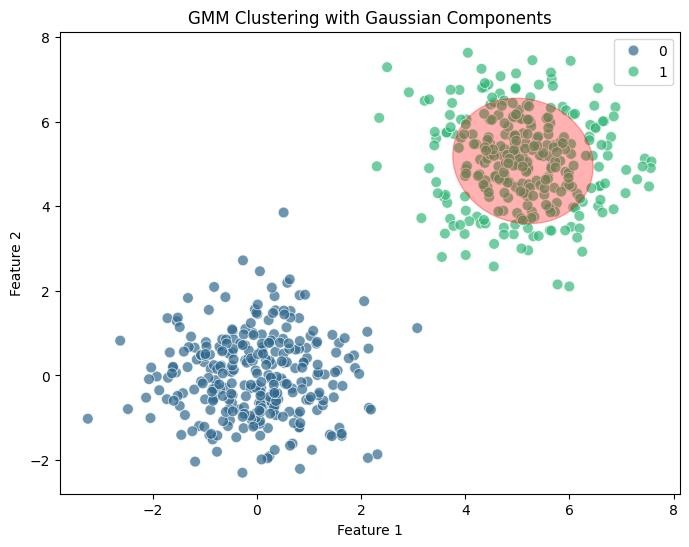
for i, label in enumerate(new\_labels):

print(f"Data point {new\_data[i]} is in Cluster {label}")

**Output :**







**Practical 7 : Model Evaluation and Hyperparameter Tuning**

**7a. Implement cross-validation techniques (k-fold, stratified, etc.) for robust model evaluation**

**Code :**

**1. Import Necessary Libraries**

import numpy as np

import pandas as pd

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split, KFold, StratifiedKFold, GridSearchCV

from sklearn.ensemble import RandomForestClassifier

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

import matplotlib.pyplot as plt

import seaborn as sns

**2. Generate a Synthetic Dataset**

**# Create a synthetic dataset with 2 classes**

X, y = make\_classification(

n\_samples=1000, n\_features=10, n\_informative=8, n\_redundant=2,

n\_clusters\_per\_class=1, random\_state=42

)

**# Convert to a DataFrame for visualization**

df = pd.DataFrame(X, columns=[f'Feature\_{i}' for i in range(1, 11)])

df['Target'] = y

**# Display the first few rows**

print(df.head())

**3. Split Data into Train and Test Sets**

**# Split data into 80% training and 20% testing**

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

**4. Define k-Fold Cross-Validation**

kf = KFold(n\_splits=5, shuffle=True, random\_state=42)

print("k-Fold Cross-Validation:")

for train\_index, val\_index in kf.split(X\_train):

print("TRAIN:", train\_index, "VALIDATION:", val\_index)

**5. Define Stratified k-Fold Cross-Validation**

skf = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)

print("\nStratified k-Fold Cross-Validation:")

for train\_index, val\_index in skf.split(X\_train, y\_train):

print("TRAIN:", train\_index, "VALIDATION:", val\_index)

**6. Train and Evaluate Using k-Fold Cross-Validation**

**# Initialize model**

model = RandomForestClassifier(random\_state=42)

**# Perform k-Fold Cross-Validation**

accuracies = []

for train\_index, val\_index in kf.split(X\_train):

X\_kf\_train, X\_kf\_val = X\_train[train\_index], X\_train[val\_index]

y\_kf\_train, y\_kf\_val = y\_train[train\_index], y\_train[val\_index]

**# Train model**

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

**# Validate model**

y\_pred = model.predict(X\_kf\_val)

accuracy = accuracy\_score(y\_kf\_val, y\_pred)

accuracies.append(accuracy)

print(f"Average Accuracy from k-Fold: {np.mean(accuracies):.2f}")

**7. Hyperparameter Tuning Using GridSearchCV**

**# Define parameter grid**

param\_grid = {

'n\_estimators': [50, 100, 200],

'max\_depth': [None, 10, 20, 30],

'min\_samples\_split': [2, 5, 10],

}

**# Perform GridSearchCV with Stratified k-Fold**

grid\_search = GridSearchCV(

estimator=RandomForestClassifier(random\_state=42),

param\_grid=param\_grid,

cv=StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42),

scoring='accuracy',

n\_jobs=-1,

verbose=1

)

**# Fit to training data**

grid\_search.fit(X\_train, y\_train)

print("Best Parameters:", grid\_search.best\_params\_)

print("Best Cross-Validation Accuracy:", grid\_search.best\_score\_)

**8. Evaluate the Final Model**

**# Use the best model for evaluation**

best\_model = grid\_search.best\_estimator\_

**# Predict on test dat**a

y\_test\_pred = best\_model.predict(X\_test)

**# Evaluate performance**

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

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

**# Confusion matrix**

conf\_matrix = confusion\_matrix(y\_test, y\_test\_pred)

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Class 0', 'Class 1'], yticklabels=['Class 0', 'Class 1'])

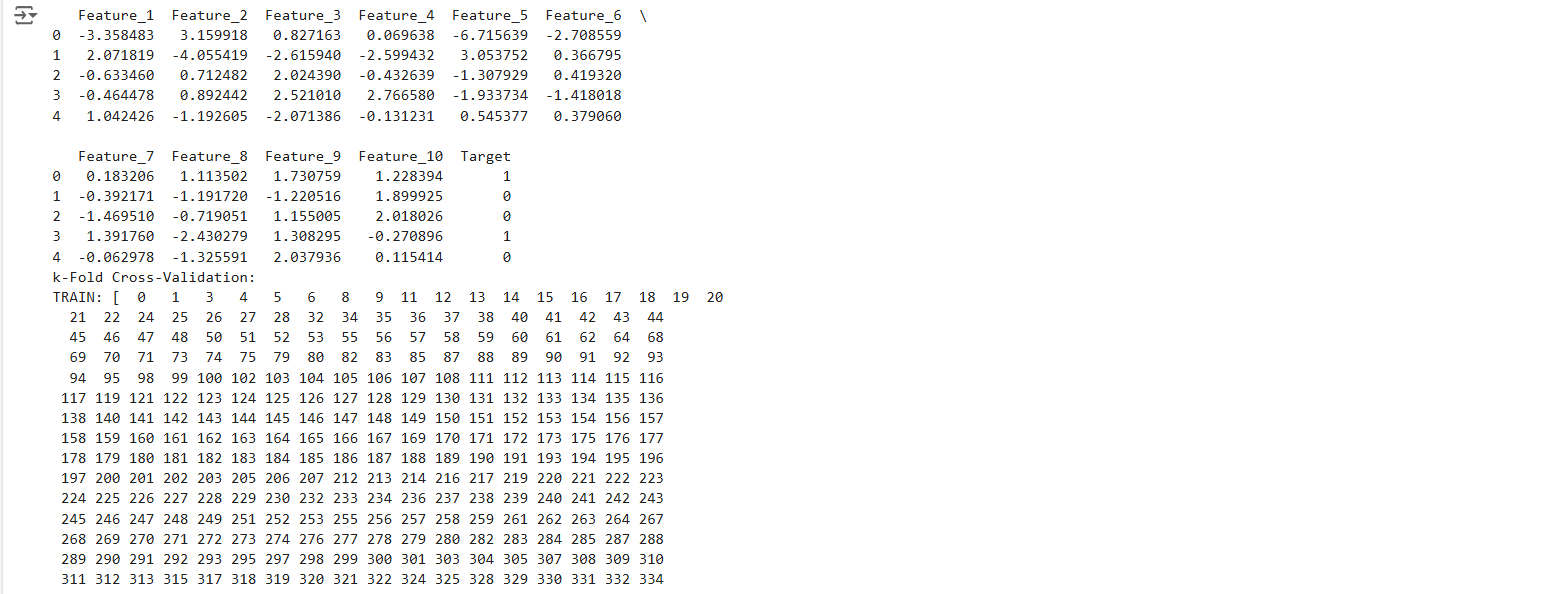
plt.xlabel('Predicted')

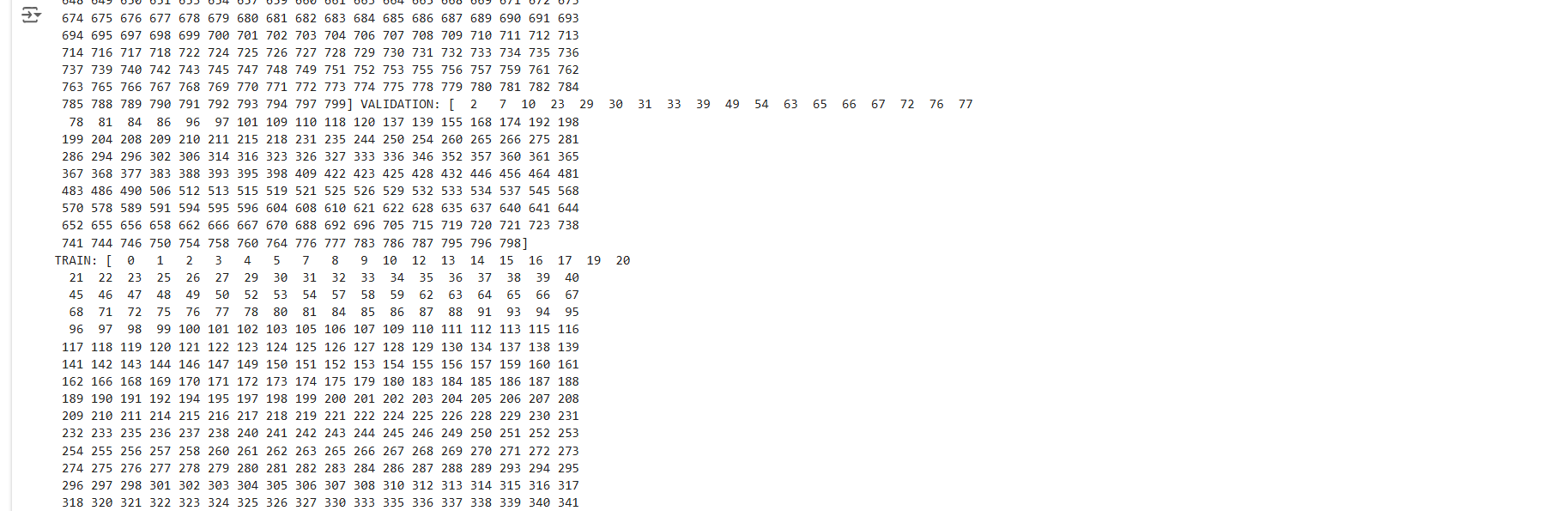
plt.ylabel('Actual')

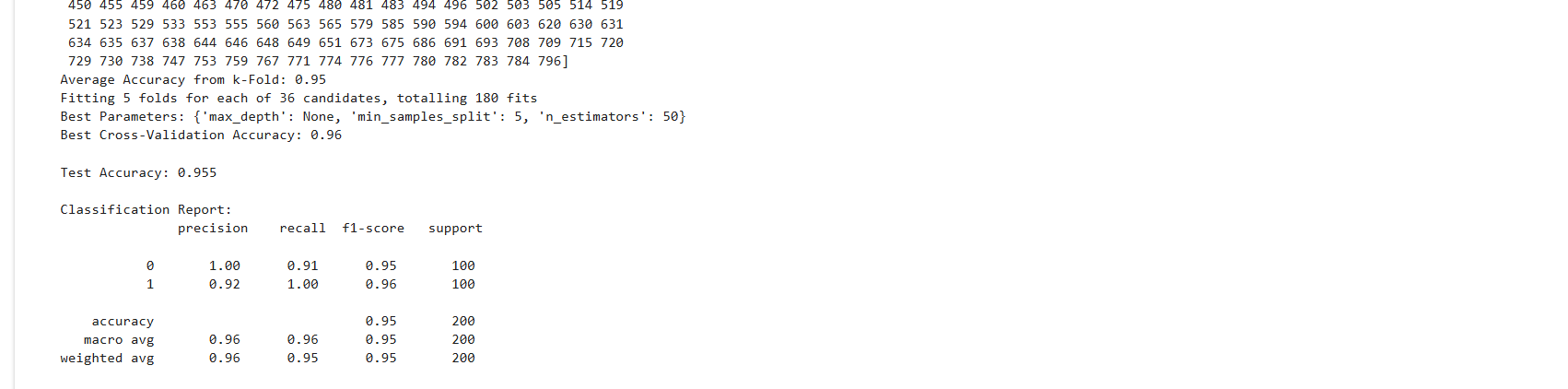
plt.title('Confusion Matrix')

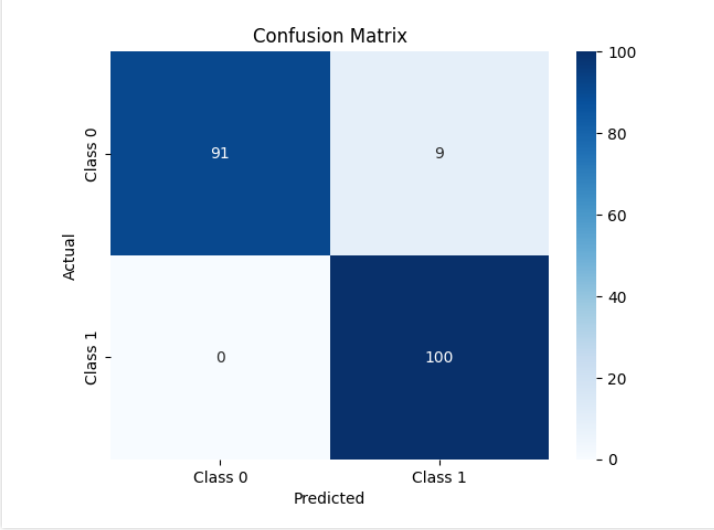
plt.show()

**Output :**

****







**7b. Systematically explore combinations of hyperparameters to optimize model performance.(use grid and randomized search)**

**Code :**

**1. Import Necessary Libraries**

import numpy as np

import pandas as pd

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split, GridSearchCV, RandomizedSearchCV, StratifiedKFold

from sklearn.ensemble import RandomForestClassifier

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

import matplotlib.pyplot as plt

import seaborn as sns

**2. Generate a Synthetic Dataset**

**# Generate a binary classification dataset**

X, y = make\_classification(

n\_samples=1000, n\_features=12, n\_informative=8, n\_redundant=2,

n\_clusters\_per\_class=1, flip\_y=0.03, random\_state=42

)

**# Convert to a DataFrame for visualization**

df = pd.DataFrame(X, columns=[f'Feature\_{i}' for i in range(1, 13)])

df['Target'] = y

**# Display the first few rows**

print(df.head())

**3. Split Data into Train and Test Sets**

**# Split data 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, stratify=y)

**4. Define the Model**

**# Initialize a Random Forest classifier**

model = RandomForestClassifier(random\_state=42)

**5. Hyperparameter Tuning Using Grid Search**

# Define a parameter grid for Grid Search

param\_grid = {

'n\_estimators': [50, 100, 200],

'max\_depth': [None, 10, 20],

'min\_samples\_split': [2, 5, 10],

'min\_samples\_leaf': [1, 2, 4]

}

**# GridSearchCV with 5-fold cross-validation**

grid\_search = GridSearchCV(

estimator=model,

param\_grid=param\_grid,

scoring='accuracy',

cv=StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42),

verbose=1,

n\_jobs=-1

)

**# Fit the model**

grid\_search.fit(X\_train, y\_train)

**# Best parameters and score from Grid Search**

print("Best Parameters from Grid Search:", grid\_search.best\_params\_)

print("Best Cross-Validation Accuracy from Grid Search:", grid\_search.best\_score\_)

**6. Hyperparameter Tuning Using Randomized Search**

from scipy.stats import randint

**# Define a parameter distribution for Randomized Search**

param\_dist = {

'n\_estimators': randint(50, 300),

'max\_depth': [None, 10, 20, 30],

'min\_samples\_split': randint(2, 15),

'min\_samples\_leaf': randint(1, 10)

}

**# RandomizedSearchCV with 5-fold cross-validation**

random\_search = RandomizedSearchCV(

estimator=model,

param\_distributions=param\_dist,

n\_iter=50, # Number of random combinations to try

scoring='accuracy',

cv=StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42),

verbose=1,

n\_jobs=-1,

random\_state=42

)

**# Fit the model**

random\_search.fit(X\_train, y\_train)

**# Best parameters and score from Randomized Search**

print("Best Parameters from Randomized Search:", random\_search.best\_params\_)

print("Best Cross-Validation Accuracy from Randomized Search:", random\_search.best\_score\_)

**7. Evaluate the Best Model**

# Select the best model from Grid Search and Randomized Search

best\_model = random\_search.best\_estimator\_ # Or use grid\_search.best\_estimator\_

**# Predict on test data**

y\_test\_pred = best\_model.predict(X\_test)

**# Evaluate the performance**

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

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

**# Confusion Matrix**

conf\_matrix = confusion\_matrix(y\_test, y\_test\_pred)

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Class 0', 'Class 1'], yticklabels=['Class 0', 'Class 1'])

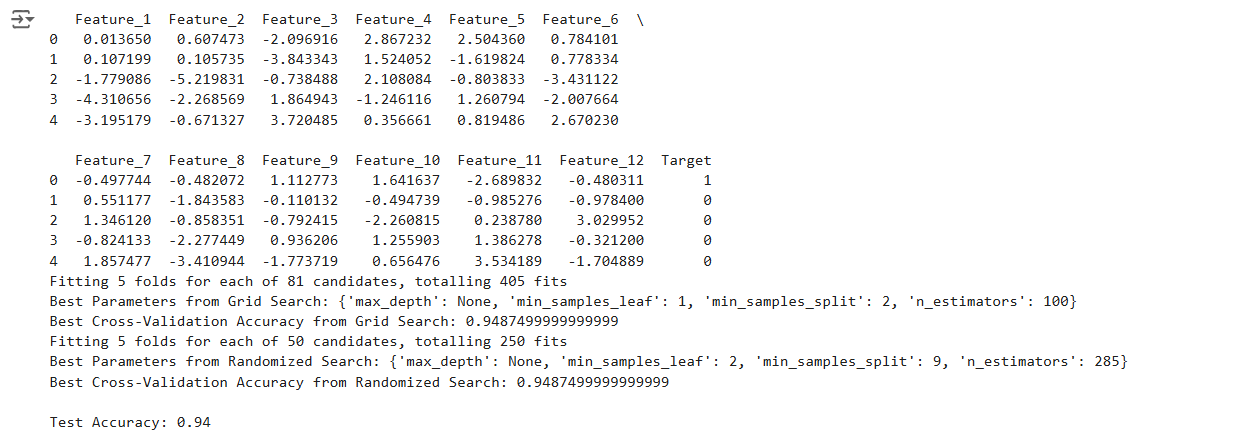
plt.xlabel('Predicted')

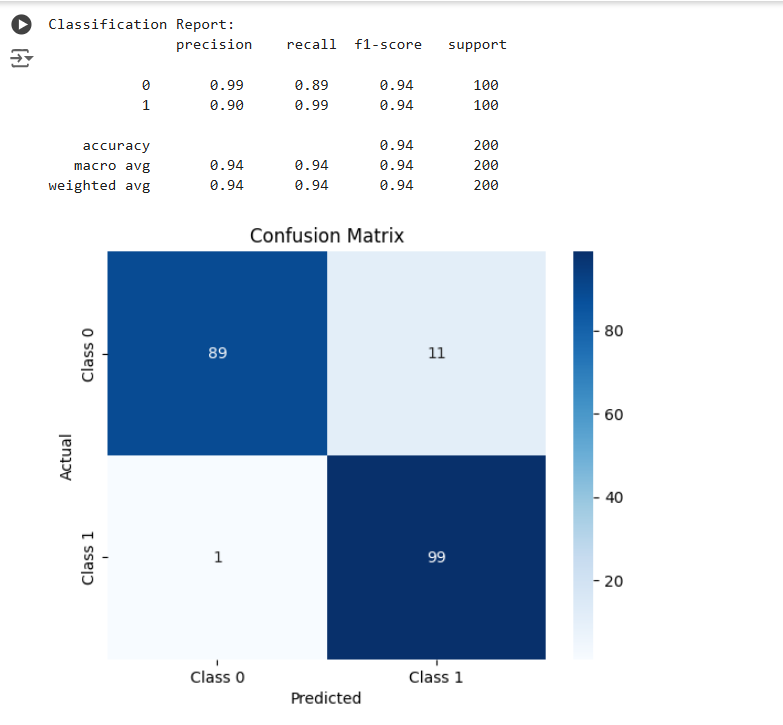
plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()

**Output :**

****

****

**Practical 8 : Bayesian Learning**

**Implement Bayesian Learning using inferences**

**Code :**

**1. Import Necessary Libraries**

import numpy as np

import pandas as pd

from sklearn.datasets import make\_classification

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

from sklearn.naive\_bayes import GaussianNB

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

import seaborn as sns

import matplotlib.pyplot as plt

**2. Generate a Synthetic Dataset**

We create a dataset suitable for classification problems.

**# Generate a dataset with 2 classes**

X, y = make\_classification(

n\_samples=1000, n\_features=8, n\_informative=6, n\_redundant=2,

n\_classes=2, random\_state=42)

**# Convert to DataFrame for visualization**

df = pd.DataFrame(X, columns=[f'Feature\_{i}' for i in range(1, 9)])

df['Target'] = y

**# Display the first few rows**

print(df.head())

**3. Split the Dataset**

Divide the data into training and testing sets.

**# Split data into 80% training and 20% testing**

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

**4. Bayesian Learning with Naive Bayes**

Here, we implement Bayesian Learning using the Gaussian Naive Bayes classifier.

**# Initialize the Gaussian Naive Bayes model**

model = GaussianNB()

**# Fit the model to the training data**

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

**# Predict on the test data**

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

**5. Evaluate the Model**

We evaluate the model's performance using accuracy, classification report, and confusion matrix.

**# Calculate accuracy**

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Test Accuracy: {accuracy:.2f}")

**# Print classification report**

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

**# Generate and plot confusion matrix**

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Class 0', 'Class 1'], yticklabels=['Class 0', 'Class 1'])

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()

**6. Understanding Bayesian Inference**

In Bayesian Learning, the model predicts based on the probabilities:

* **Prior Probability (P(C)P(C)P(C)):** The likelihood of each class based on historical data.
* **Likelihood (P(X∣C)P(X|C)P(X∣C)):** The probability of the data given a class.
* **Posterior Probability (P(C∣X)P(C|X)P(C∣X)):** Calculated using Bayes' theorem: P(C∣X)=P(X∣C)⋅P(C)P(X)P(C|X) = \frac{P(X|C) \cdot P(C)}{P(X)}P(C∣X)=P(X)P(X∣C)⋅P(C)​

**# Example: Compute posterior probabilities for the first test sample**

sample = X\_test[0].reshape(1, -1)

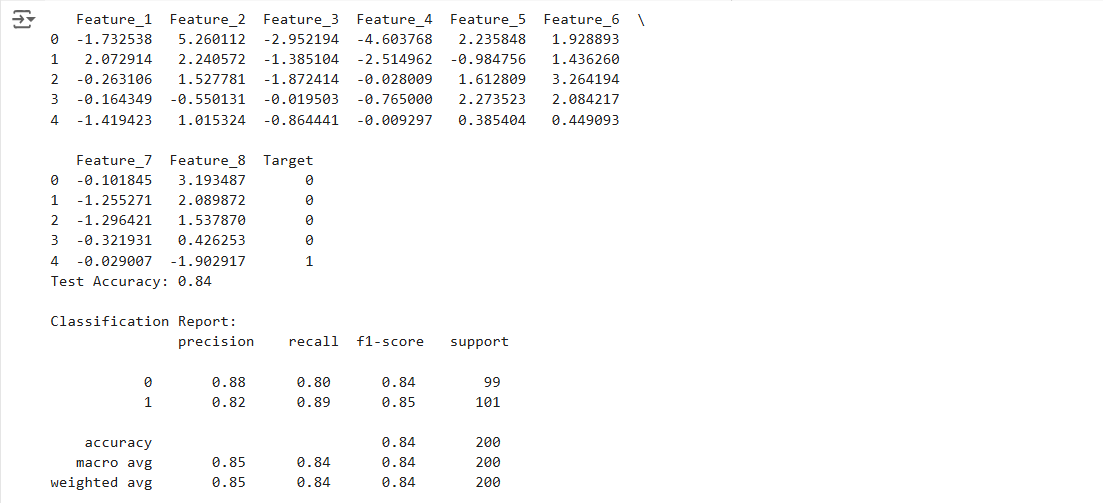
posterior\_probs = model.predict\_proba(sample)

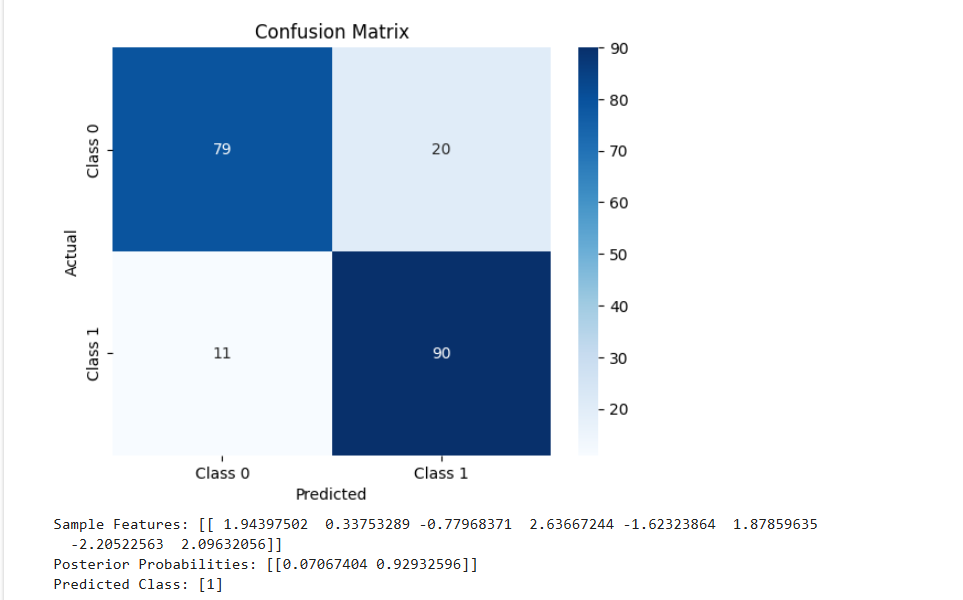
print(f"Sample Features: {sample}")

print(f"Posterior Probabilities: {posterior\_probs}")

print(f"Predicted Class: {model.predict(sample)}")

**Output :**

****

****