NURTURING POTENTIAL

SAKET GYANPEETH'S

SAKET COLLEGE OF ARTS, SCIENCE AND COMMERCE

KALYAN (EAST)

ACADEMIC YEAR 2025-26

M.Sc. Information Technology

Part II NEP 2020 Semester III

SUBMITTED BY

Mr. YASH PRAMOD GAIKWAD

AS PRESCRIBED BY

UNIVERSITY OF MUMBAI

MUMBAI UNIVERSITY

NURTURING POTENTIAL

SAKET GYANPEETH'S

SAKET COLLEGE OF ARTS, SCIENCE AND COMMERCE

(Permanently Affiliated to University of Mumbai)

NAAC Accredited

Saket Vidyanagri Marg, Chinchpada Road, Katemanivali,
Kalyan (East) -421306(Mah)

Department of Information Technology

This is to certify that

Mr./Ms. YASH PRAMOD GAIKWAD Seat No. 254337

of ADVANCED ARTIFICIAL INTELLIGENCE

M.Sc. Information Technology

Part II NEP 2020 Semester III

has satisfactory carried out the required practical in the subject
of
For the Academic year 2025 – 2026

Practical In-Charge

Head of the Department

External Examiner

College Seal

INDEX

Sr. No.	Practical Aim	Signature
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.	

Practical No. 1

Aim: Implementing advanced deep learning algorithms such as convolutional neural networks (CNNs) using TensorFlow.

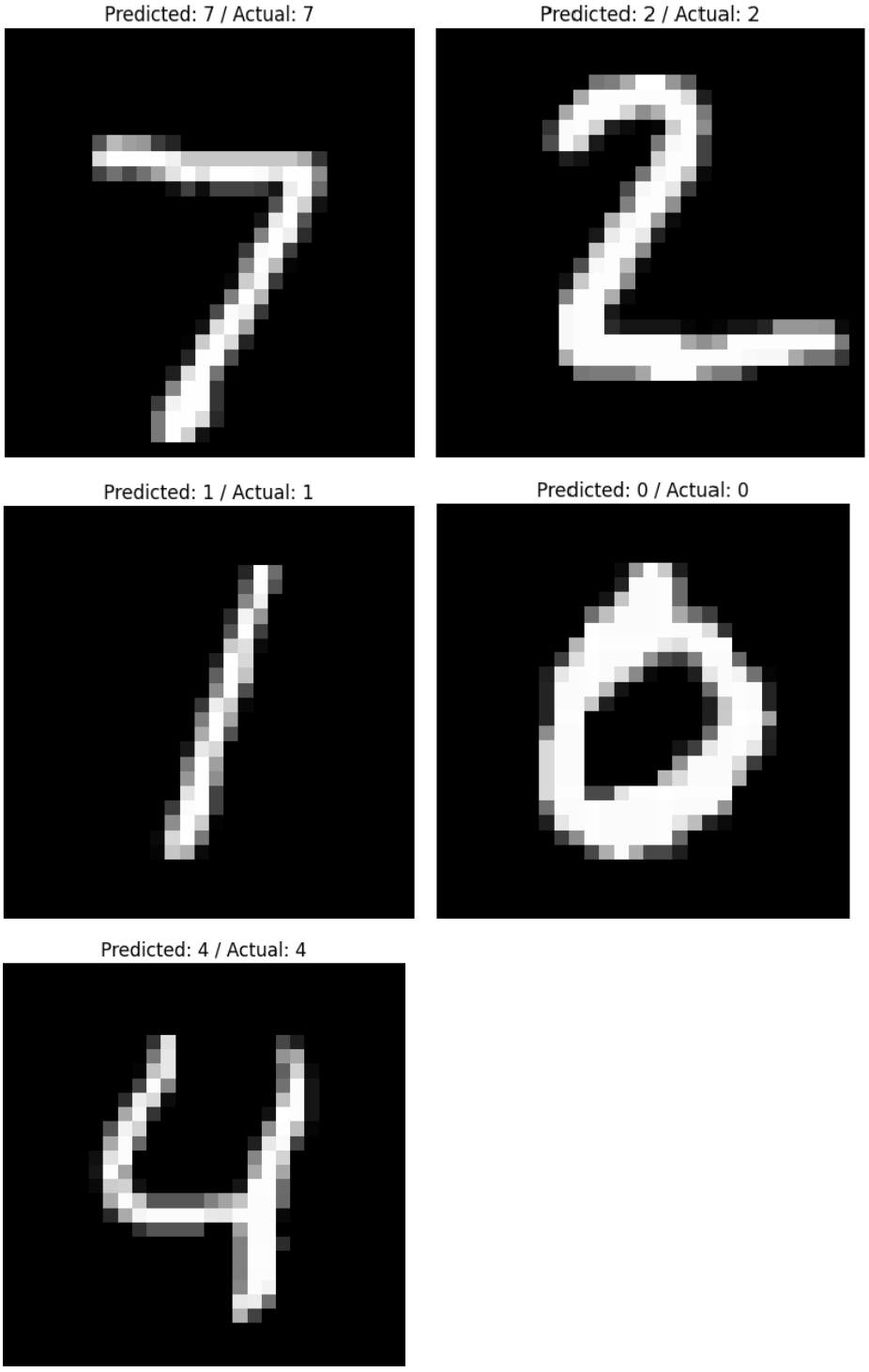
Writeup:

- **Import Libraries:** It starts by importing tensorflow and specific layers and models from tensorflow.keras. numpy and matplotlib.pyplot are also imported for data handling and visualization.
- **Load and Preprocess Data:** The MNIST dataset is loaded, which consists of grayscale images of digits (0-9). The pixel values are normalized by dividing by 255.0 to scale them between 0 and 1. A channel dimension is added to the images (x_train[..., tf.newaxis]) because CNNs typically expect input with a channel dimension (e.g., 1 for grayscale, 3 for RGB).
- **Build the CNN Model:** A models.Sequential model is created, which is a linear stack of layers.
    - ***layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)):*** This is the first convolutional layer. It uses 32 filters (which learn features), each of size 3x3. The activation='relu' applies the ReLU activation function, which introduces non-linearity. input_shape=(28, 28, 1) specifies the shape of the input images (28x28 pixels with 1 channel).
    - ***layers.MaxPooling2D((2, 2)):*** This is a max pooling layer with a pool size of 2x2. It reduces the spatial dimensions of the feature maps, helping to reduce computation and extract the most important features.
    - ***layers.Conv2D(64, (3, 3), activation='relu'):*** Another convolutional layer, this time with 64 filters.
    - ***layers.MaxPooling2D((2, 2)):*** Another max pooling layer.
    - ***layers.Flatten():*** This layer flattens the output of the convolutional and pooling layers into a 1D vector. This is necessary before passing the data to the dense layers.
    - ***layers.Dense(64, activation='relu'):*** A fully connected (dense) layer with 64 neurons and ReLU activation.
    - ***layers.Dense(10, activation='softmax'):*** The output layer. It's a dense layer with 10 neurons (one for each digit class). The softmax activation function outputs a probability distribution over the 10 classes, indicating the model's confidence in each digit.
- **Compile the Model:** The model is compiled with an optimizer ('adam'), a loss function ('sparse_categorical_crossentropy' which is suitable for multi-class classification with integer labels), and a metric to monitor during training ('accuracy').
- **Train the Model:** The model is trained using the model.fit() method on the training data (x_train, y_train) for a specified number of epochs. validation_split=0.1 sets aside 10% of the training data for validation during training.
- **Evaluate the Model:** The trained model is evaluated on the test data (x_test, y_test) using model.evaluate() to determine its performance on unseen data.
- **Make and Visualize Predictions:** The model makes predictions on the test data using model.predict(). A loop then visualizes the first few test images along with the model's predicted digit and the actual digit.

Code:

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np
import matplotlib.pyplot as plt
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# Normalize the images (scale from 0-255 → 0-1)
x_train = x_train / 255.0
x_test = x_test / 255.0
# Add a channel dimension: (28,28) → (28,28,1)
x_train = x_train[..., tf.newaxis]
x_test = x_test[..., tf.newaxis]
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')  # 10 digits: 0–9
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, validation_split=0.1)
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.4f}")
predictions = model.predict(x_test)
# Plot a few predictions
for i in range(5):
    plt.imshow(x_test[i].squeeze(), cmap='gray')
    plt.title(f"Predicted: {np.argmax(predictions[i])} / Actual: {y_test[i]}")
    plt.axis('off')
    plt.show()
```

Output:


Predicted: 7 / Actual: 7
Predicted: 2 / Actual: 2
Predicted: 1 / Actual: 1
Predicted: 0 / Actual: 0
Predicted: 4 / Actual: 4

Practical No. 2

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

Writeup:

- **Import Libraries:** It imports necessary libraries like tensorflow , Tokenizer and pad_sequences for text preprocessing, Sequential for building the model, and layers like Embedding, Flatten, and Dense. numpy is used for handling labels.
- **Prepare Data:**
  - ***texts:*** This is your input text data, which are short sentences expressing different sentiments.
  - ***labels:*** These are the corresponding sentiment labels for each sentence (1 for positive, 0 for negative). This is the target variable your model will learn to predict.
- **Text Preprocessing:**
  - ***Tokenizer:*** This is a crucial step in NLP. The Tokenizer converts text into numerical sequences. It builds a vocabulary from your text data, assigning a unique integer to each word. The oov_token="<OOV>" handles words that are not in the vocabulary during later processing.
  - ***fit_on_texts(texts):*** This method trains the tokenizer on your texts to build the vocabulary.
  - ***texts_to_sequences(texts):*** This converts your text sentences into sequences of integers based on the vocabulary created by the tokenizer.
  - ***pad_sequences:*** Neural networks typically require input sequences of the same length. pad_sequences adds padding (usually zeros) to the sequences to make them all the same length (maxlen=5 in this case). padding='post' means padding is added at the end of the sequence.
- **Build the NLP Model:** A Sequential model is created:
  - ***Embedding(input_dim=vocab_size, output_dim=8, input_length=5):*** This is the embedding layer, which is fundamental for many NLP tasks. It converts the integer sequences into dense vectors of fixed size (8 in this case). Each word in your vocabulary will be represented by a unique vector. This layer learns to represent words in a way that captures their semantic meaning. The input_length=5 specifies the length of the input sequences.
  - ***Flatten():*** This layer flattens the output of the embedding layer into a 1D vector, preparing it for the dense layers.
  - ***Dense(6, activation='relu'):*** A dense layer with 6 neurons and ReLU activation. These layers learn to combine the features from the embedding layer.
  - ***Dense(1, activation='sigmoid'):*** The output layer. It has 1 neuron with a sigmoid activation function. The sigmoid function outputs a value between 0 and 1, which can be interpreted as the probability of the input text having positive sentiment (closer to 1) or negative sentiment (closer to 0).
- **Compile the Model:** The model is compiled with the adam optimizer and binary_crossentropy loss function, which is suitable for binary classification tasks like sentiment analysis (positive or negative). metrics=['accuracy'] tracks the model's accuracy during training.

- **Train the Model:** The model is trained on the padded sequences and corresponding labels for a specified number of epochs.
- **Test and Predict:**
  - ***test_texts:*** New sentences are defined to test the trained model.
  - These test sentences are preprocessed using the same tokenizer and padding as the training data.
  - ***model.predict(test_pad):*** The trained model makes predictions on the preprocessed test data. The output is a probability score for each test sentence.

The code then iterates through the test sentences and their predictions, classifying the sentiment as "Positive" if the probability is above 0.51 and "Negative" otherwise. The sentiment and the prediction probability are printed.

Code:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense
import numpy as np
texts = [
    "I love this product",
    "This is the worst movie",
    "I am so happy",
    "I hate this book",
    "What a great day",
    "I am very disappointed"
]
labels = [1, 0, 1, 0, 1, 0]  # 1 = Positive, 0 = Negative
# Initialize tokenizer
tokenizer = Tokenizer(oov_token="<OOV>")
tokenizer.fit_on_texts(texts)
# Convert to sequences
sequences = tokenizer.texts_to_sequences(texts)
padded = pad_sequences(sequences, padding='post', maxlen=5)
# Vocabulary size
vocab_size = len(tokenizer.word_index) + 1
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=8, input_length=5),
    Flatten(),
    Dense(6, activation='relu'),
    Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
#model.summary()
# Convert labels to NumPy array
labels = np.array(labels)
```

```
# Train
model.fit(padded, labels, epochs=50, verbose=1)  # Set verbose=1 if you want output
# Test new sentences
#test_texts = ["so happy", "I hate this taste", "What a worst day", "Great movie"]
test_texts = [
    "I love this product",
    "worst movie",
    "I am so happy",
    "I hate this book",
    "great day",
    "I am very disappointed"
]
# Tokenize and pad
test_seq = tokenizer.texts_to_sequences(test_texts)
test_pad = pad_sequences(test_seq, maxlen=5, padding='post')
predictions = model.predict(test_pad)

for i, text in enumerate(test_texts):
    sentiment = "Positive" if predictions[i] > 0.51 else "Negative"
    print(f"{text} → {sentiment} ({predictions[i][0]:.2f})")

for i, text in enumerate(test_texts):
    if predictions[i] > 0.51:
      sentiment = "Positive"
    else:
      sentiment = "Negative"
    print(f"{text} → {sentiment} ({predictions[i][0]:.2f})")
```

Output:

```
I love this product → Positive (0.53)
worst movie → Negative (0.51)
I am so happy → Positive (0.56)
I hate this book → Negative (0.50)
great day → Positive (0.51)
I am very disappointed → Negative (0.45)
```

Practical No. 3

Aim: Creating a chatbot using advanced techniques like transformer models

Writeup:

- **Import Libraries:** It imports tensorflow and several components from the transformers library, including BertTokenizer, TFBertForQuestionAnswering, and pipeline. These are essential for loading and using pre-trained transformer models.
- **Define Context and Questions:**
  - ***context:*** This is the block of text from which the model will find answers to the questions. It contains information about various Indian national symbols.
  - ***question:*** This is a list of questions about the information provided in the context.
- **Load Model and Tokenizer:**
  - ***AutoTokenizer.from_pretrained("deepset/bert-base-cased-squad2"):*** This line loads the tokenizer associated with the "deepset/bert-base-cased-squad2" pre-trained model. The tokenizer is responsible for converting text (both the context and the questions) into numerical IDs that the BERT model can understand.
  - ***TFBertForQuestionAnswering.from_pretrained("deepset/bert-base-cased-squad2",from_pt=True):*** This line loads the pre-trained BERT model specifically fine-tuned for the Question Answering task on the SQuAD2.0 dataset. from_pt=True indicates that the model weights are being loaded from a PyTorch checkpoint.
- **Build the Question Answering Pipeline:**
  - ***pipeline('question-answering',model = model,tokenizer = tokenizer):*** The pipeline utility from transformers is used to create a streamlined workflow for the question answering task. It combines the loaded model and tokenizer into a single object that can easily take a question and context as input and return the predicted answer.
- **Define Chatbot Function:**
  - ***def chatbot(question,context)::*** This function encapsulates the question answering process.
  - Inside the function, ***nlp({"question": question, "context": context})*** uses the question answering pipeline to find the answer to the given question within the provided context.
  - ***return output['answer']:*** The function extracts and returns the predicted answer from the pipeline's output.
- **Ask Questions and Print Answers:** The code then iterates through a few of the questions defined earlier, calls the chatbot function with each question and the context, and prints the question along with the answer returned by the chatbot.

Code:

```
#optional → !pip install transformers torch tensorflow
import tensorflow as tf
import transformers
from transformers import BertTokenizer
from transformers import TFBertForQuestionAnswering
```

```
from transformers import AutoTokenizer
from transformers import pipeline
print("Transformers version:", transformers.__version__)
print("TensorFlow version:", tf.__version__)
print("Model path or name:", "deepset/bert-base-cased-squad2")
# This is the given context
context = """
The Bengal tiger was chosen as the national animal in a meeting of the Indian wildlife board in
1972 and was adopted officially in April 1973. It was chosen over the Asiatic lion due to the
wider presence of the tiger across India.
Indian peacock was designated as the national bird of India in February 1963.
Indian elephant is the largest terrestrial mammal in India and a cultural symbol throughout its
range, appearing in various religious traditions and mythologies.
Indian banyan is a large tree native to the Indian subcontinent and produces aerial roots from the
branches which grow downwards, eventually becoming trunks
Mango is a large fruit tree with many varieties, believed to have originated in northeast India.
Lotus is an aquatic plant adapted to grow in the flood plains. Lotus seeds can remain dormant
and viable for many years, therefore the plant is regarded as a symbol of longevity.
"""
# These are some questions which we are going to ask to BERT model
question = [
    "What is the National animal of India?",
    "What is the National bird of India?",
    "What is the largest terrestrial mammal in India?",
    "Which tree produces aerial roots?",
    "Which fruit originated in northeast India?",
    "What is the symbol of longevity?",
]
# Importing the tokenizer of the BERT model
tokenizer = AutoTokenizer.from_pretrained("deepset/bert-base-cased-squad2")
# Importing the BERT model
model = TFBertForQuestionAnswering.from_pretrained("deepset/bert-base-cased-
squad2",from_pt=True)
# Tokenizing the first question and print the encoded output
tokenizer.encode(question[0],truncation = True, padding = True)
# Building the BERT "Question - Answering" pipeline
nlp = pipeline('question-answering',model = model,tokenizer = tokenizer)
# Making a chatbot function for question answering
def chatbot(question,context):
  output = nlp({
      "question": question,
      "context": context
  })
  return output['answer']
# Asking the first question to the chatbot
print("Question: ",question[1])
```

```
print("Answer: ",chatbot(question[1],context))
print("Question: ",question[2])
print("Answer: ",chatbot(question[2],context))
print("Question: ",question[3])
print("Answer: ",chatbot(question[3],context))
print("Question: ",question[4])
print("Answer: ",chatbot(question[4],context))
print("Question: ",question[5])
print("Answer: ",chatbot(question[5],context))
```

Output:

```
Transformers version: 4.55.4
TensorFlow version: 2.19.0
Model path or name: deepset/bert-base-cased-squad2

[101, 1327, 1110, 1103, 1305, 3724, 1104, 1726, 136, 102]
Question:  What is the National bird of India?
Answer:  Indian peacock
Question:  What is the largest terrestrial mammal in India?
Answer:  Indian elephant
Question:  Which tree produces aerial roots?
Answer:  Indian banyan
Question:  Which fruit originated in northeast India?
Answer:  Mango
Question:  What is the symbol of longevity?
Answer:  Lotus
```

Practical No. 4

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

Writeup:

- **Import Libraries:** It imports numpy for numerical operations and modules from tensorflow.keras for building the neural network model (Input, Embedding, Dot, Flatten, Model).
- **Sample Data:** ratings is a small NumPy array representing sample user-item ratings. Each row is a [user_id, item_id, rating] triplet.
- **Determine Dimensions:** It calculates the number of unique users (num_users) and items (num_items) from the ratings data to set the input dimensions for the embedding layers.
- **Define Embedding Size:** embedding_size is set to 2. This determines the size of the learned vector representation (embedding) for each user and item.
- **Define Model Inputs:** user_input and item_input are defined as Keras Input layers, specifying that the model will take two separate inputs, one for user IDs and one for item IDs.
- **Create Embedding Layers:**
  - ***user_embedding = Embedding(...):*** An Embedding layer is created for users. It takes the num_users as input dimension and outputs a vector of size embedding_size for each user ID.
  - ***item_embedding = Embedding(...):*** An Embedding layer is created similarly for items. These layers learn to represent users and items as dense vectors in a low-dimensional space.
- **Calculate Dot Product:**
  - ***dot_product = Dot(axes=2)([user_embedding, item_embedding]):*** A Dot layer is used to calculate the dot product between the user embedding and the item embedding. The dot product of these learned vectors is used to predict the rating. axes=2 specifies that the dot product is calculated across the last dimension of the input tensors (the embedding dimension).
- **Flatten Output:** ***dot_product = Flatten()(dot_product):*** The output of the dot product is flattened into a 1D tensor.
- **Define the Model:** ***model = Model(inputs=[user_input, item_input], outputs=dot_product):*** A Keras Model is created, specifying the input layers (user and item inputs) and the output layer (the flattened dot product). This model takes a user ID and an item ID and outputs a predicted rating.
- **Compile the Model:** ***model.compile(...):*** The model is compiled with the adam optimizer and mse (Mean Squared Error) loss function. The goal is to minimize the difference between the predicted rating and the actual rating.
- **Prepare Training Data:** The ratings data is separated into user IDs (X_users), item IDs (X_items), and actual ratings (y_ratings). The data types are cast to integers and floats as needed.
- **Train the Model:** ***model.fit([X_users, X_items], y_ratings, epochs=100, verbose=1):*** The model is trained on the prepared data. The model learns the user and item embeddings such

that their dot product approximates the known ratings. It trains for 100 epochs, and verbose=1 shows the training progress.

- **Make Predictions:** The code then defines arrays of user_ids and item_ids for which to predict ratings. ***model.predict()*** is used to get the predicted ratings for these user-item pairs.
- **Print Predictions:** The code iterates through the predicted ratings and prints the predicted rating for each user-item pair.

Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Embedding, Dot, Flatten
from tensorflow.keras.models import Model
# Sample user-item-rating data
ratings = np.array([
    [0, 0, 5.0],  # user 0 rated item 0 with 5
    [0, 1, 3.0],  # user 0 rated item 1 with 3
    [1, 0, 4.0],  # user 1 rated item 0 with 4
    [1, 2, 2.0],  # user 1 rated item 2 with 2
    [2, 1, 4.0],  # user 2 rated item 1 with 5
    [2, 2, 5.0],  # user 2 rated item 2 with 5
])
# Number of users and items
num_users = int(np.max(ratings[:, 0])) + 1  # 3 users
num_items = int(np.max(ratings[:, 1])) + 1  # 3 items
embedding_size = 2
print(np.max(ratings[:, 0]),'\n',int(np.max(ratings[:, 0])))
# Inputs for user and item
user_input = Input(shape=(1,))
item_input = Input(shape=(1,))
embedding_size = 2
user_embedding = Embedding(input_dim=num_users,output_dim=embedding_size)(user_input)
item_embedding =Embedding(input_dim=num_items,output_dim=embedding_size)(item_input)
# Dot product of user and item embeddings
dot_product = Dot(axes=2)([user_embedding, item_embedding])
# Flatten the result
dot_product = Flatten()(dot_product)
# Define the model
model = Model(inputs=[user_input, item_input], outputs=dot_product)
model.compile(optimizer='adam', loss='mse')
# Training data
X_users = ratings[:, 0].astype('int32')
X_items = ratings[:, 1].astype('int32')
y_ratings = ratings[:, 2].astype('float32')
X_users
X_items
y_ratings
```

```
# Train the model
model.fit([X_users, X_items], y_ratings, epochs=100, verbose=1)
# Predict for multiple user-item pairs
user_ids = np.array([0, 1, 2])
item_ids = np.array([2, 0, 1])
predicted_batch = model.predict([user_ids, item_ids])
for u, i, p in zip(user_ids, item_ids, predicted_batch):
    print(f"Predicted rating for user {u} on item {i}: {p.item():.2f}")
ratings
# Predict for multiple user-item pairs
user_ids = np.array([0, 1, 2])
item_ids = np.array([0, 2, 1])
predicted_batch = model.predict([user_ids, item_ids])
for u, i, p in zip(user_ids, item_ids, predicted_batch):
    print(f" User {u} - item {i} - Predicted rating : {p.item():.2f}")
```

Output:

```
2.0
 2

array([0, 0, 1, 1, 2, 2], dtype=int32)
array([0, 1, 0, 2, 1, 2], dtype=int32)
array([5., 3., 4., 2., 4., 5.], dtype=float32)

Predicted rating for user 0 on item 2: 0.01
Predicted rating for user 1 on item 0: 0.04
Predicted rating for user 2 on item 1: 0.04

array([[0., 0., 5.],
       [0., 1., 3.],
       [1., 0., 4.],
       [1., 2., 2.],
       [2., 1., 4.],
       [2., 2., 5.]])

User 0 - item 0 - Predicted rating : 0.04
 User 1 - item 2 - Predicted rating : 0.01
 User 2 - item 1 - Predicted rating : 0.04
```

Practical No. 5

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

Writeup:

- **Import Libraries:** It imports cv2 (OpenCV) for image processing, numpy for numerical operations (especially for defining color ranges), and cv2_imshow from google.colab.patches to display images in Google Colab.
- **Load the Image:** ***image = cv2.imread('/content/image.png')*** loads an image file into a NumPy array, which is how OpenCV represents images.
- **Convert to HSV:** ***hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)*** converts the image from the BGR color space (OpenCV's default) to the HSV (Hue, Saturation, Value) color space. This conversion is crucial for color-based segmentation because colors are represented in a way that makes it easier to define color ranges.
    - **Hue (H):** Represents the color type (e.g., red, blue, green).
    - **Saturation (S):** Represents the purity or intensity of the color.
    - **Value (V):** Represents the brightness of the color.
- **Define Color Range** (Examples: blue, green, red):
    - ***lower_blue = np.array([100, 150, 0])*** and ***upper_blue = np.array([140, 255, 255]):*** These lines define the lower and upper bounds for the blue color range in HSV. Pixels with HSV values within this range will be considered blue.
    - Similar lower and upper bounds are defined for green and red. Choosing appropriate HSV ranges for the colors you want to segment is a key part of this technique.
- **Create Mask:** ***mask = cv2.inRange(hsv, lower_blue, upper_blue):*** This is the core segmentation step. cv2.inRange() takes the HSV image and the lower and upper color bounds. It creates a binary mask where pixels within the specified color range are set to white (255), and all other pixels are set to black (0). This mask essentially isolates the pixels of the target color.
- **Segment Image:** ***segmented = cv2.bitwise_and(image, image, mask=mask):*** This line applies the generated mask to the original image. cv2.bitwise_and() performs a bitwise AND operation between the original image and itself, but only where the mask is non-zero (white). This effectively keeps the original pixel values only for the areas where the mask is white (the segmented color) and sets the rest to black, resulting in the segmented image showing only the target color.
- **Show Images:** ***cv2_imshow(image)*** and ***cv2_imshow(segmented):*** These lines display the original image and the segmented image using cv2_imshow, which is necessary for displaying OpenCV images in a Colab environment.

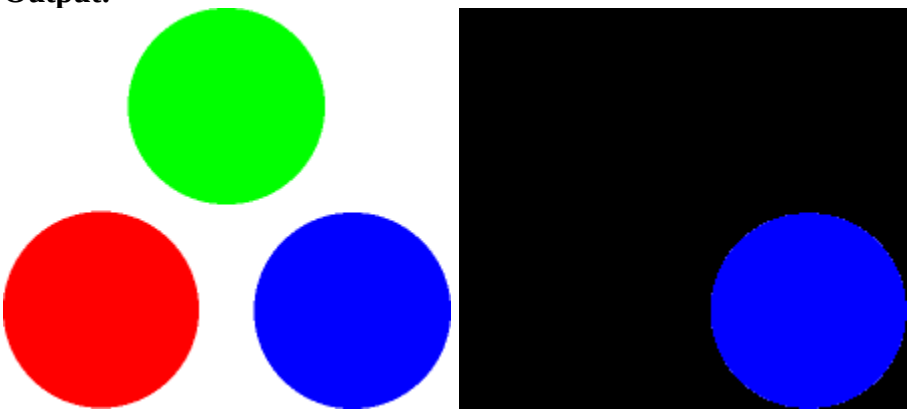
The code repeats the process for blue, green, and red color ranges to demonstrate segmenting different colors.

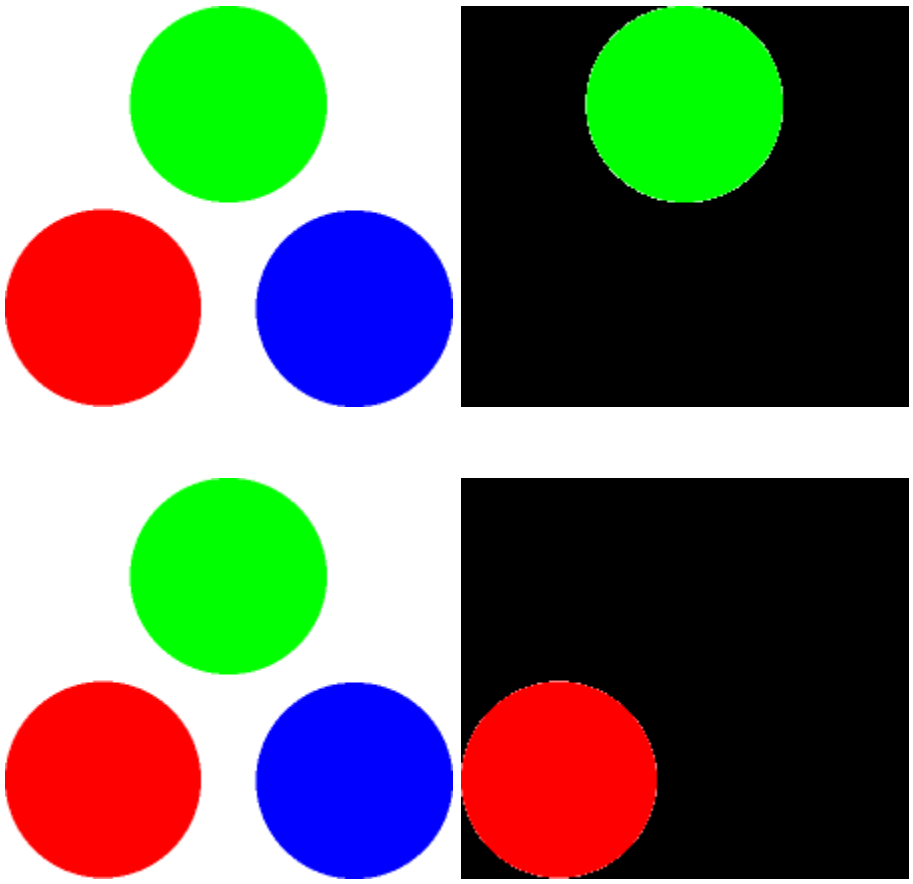
Code:

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow
# Load the image
```

```
image = cv2.imread('/content/image.png')
# Convert to HSV
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
# Define color range (example: blue)
lower_blue = np.array([100, 150, 0])
upper_blue = np.array([140, 255, 255])
# Create mask
mask = cv2.inRange(hsv, lower_blue, upper_blue)
# Segment image
segmented = cv2.bitwise_and(image, image, mask=mask)
# Show images using cv2_imshow in Colab
cv2_imshow(image)
cv2_imshow(segmented)
#Define color range (example: green)
lower_green = np.array([40, 40, 40])
upper_green = np.array([80, 255, 255])
# Create mask
mask = cv2.inRange(hsv, lower_green, upper_green)
# Segment image
segmented = cv2.bitwise_and(image, image, mask=mask)
# Show images using cv2_imshow in Colab
cv2_imshow(image)
cv2_imshow(segmented)
#Define color range (example: red)
lower_red = np.array([0, 100, 100])
upper_red = np.array([10, 255, 255])
# Create mask
mask = cv2.inRange(hsv, lower_red, upper_red)
# Segment image
segmented = cv2.bitwise_and(image, image, mask=mask)
# Show images using cv2_imshow in Colab
cv2_imshow(image)
cv2_imshow(segmented)
```

Output:





Practical No. 6

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

Writeup:

- **Import Libraries:** It imports tensorflow for building and training the neural networks, numpy for numerical operations, and matplotlib.pyplot for visualizing the generated images.
- **Load and Preprocess Data:** The MNIST dataset of handwritten digits is loaded. The images are reshaped to have a channel dimension and normalized to the range [-1, 1]. Normalizing to this range is common in GANs, especially when using tanh activation in the generator's output layer. A tf.data.Dataset is created for efficient batching and shuffling of the training data.
- **Define Hyperparameters:** batch_size, noise_dim (the size of the random noise vector fed to the generator), and epochs (the number of training cycles) are defined.
- **Define the Generator Model:** ***generator = tf.keras.Sequential([...])*** defines the generator network. Its purpose is to take a random noise vector (noise_dim) and transform it into an image that looks like the training data.
    - It starts with a Dense layer to project the noise into a higher dimension.
    - Reshape changes the output shape to a 3D tensor (like a small image with many channels).
    - BatchNormalization helps stabilize training.
    - LeakyReLU is an activation function commonly used in GANs.
    - Conv2DTranspose (also known as deconvolution or upsampling) layers are used to increase the spatial dimensions of the data, gradually building up an image from the smaller representation. The strides greater than 1 (strides=2) are key to upsampling.
    - The final Conv2DTranspose layer outputs a single-channel image (grayscale) with a tanh activation function, which outputs values in the range [-1, 1], matching the normalized input data.
- **Define the Discriminator Model:** ***discriminator = tf.keras.Sequential([...])*** defines the discriminator network. Its purpose is to take an image (either a real image from the dataset or a fake image from the generator) and output a single value indicating whether it thinks the image is real (closer to 1) or fake (closer to 0).
    - Conv2D layers with strides greater than 1 (strides=2) are used to downsample the image and extract features.
    - LeakyReLU and Dropout (to prevent overfitting) are used.
    - Flatten converts the 2D feature maps into a 1D vector.
    - The final Dense layer outputs a single value.
- **Define Loss Function and Optimizers:**
    - ***loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True):*** Binary crossentropy is used as the loss function. from_logits=True is important because the discriminator's final Dense layer does not have an activation function applied (it outputs logits).
    - ***gen_optimizer*** and ***disc_optimizer:*** Separate Adam optimizers are defined for the generator and discriminator, as they are trained adversarially.

- **Training Loop:** The code enters a loop for the specified number of epochs.
  - Inside the epoch loop, it iterates through batches of real images from the dataset.
  - For each batch, random noise is generated.
    - ***Training the Discriminator:***
      - A tf.GradientTape is used to record operations for automatic differentiation.
      - Fake images are generated by the generator.
      - The discriminator is called on both real and fake images.
      - Losses are calculated: real_loss (discriminator trying to classify real images as real) and fake_loss (discriminator trying to classify fake images as fake). The total disc_loss is the sum of these.
      - Gradients of the disc_loss with respect to the discriminator's trainable variables are computed and applied using the disc_optimizer.
    - ***Training the Generator:***
      - Another tf.GradientTape is used.
      - Fake images are generated.
      - The discriminator is called on the fake images.
      - gen_loss is calculated: the generator tries to make the discriminator classify the fake images as real (tf.ones_like(fake_output)).
      - Gradients of the gen_loss with respect to the generator's trainable variables are computed and applied using the gen_optimizer.
- **Visualization:** After each epoch, the generator is used to generate a sample of images from a fixed seed noise vector (to see how the generation improves over epochs). These images are rescaled back to [0, 1] and displayed using matplotlib.pyplot.

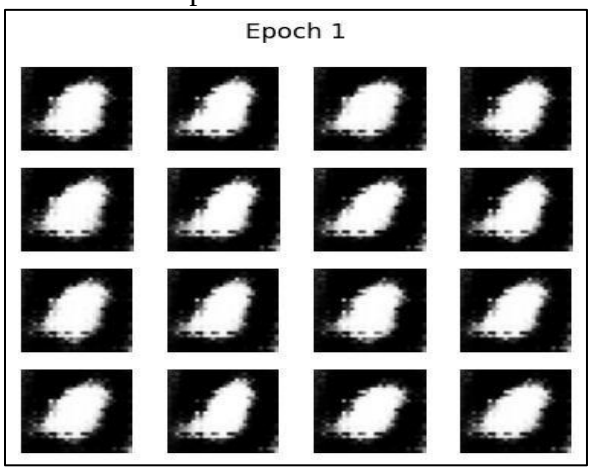
Code:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
(x_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape(-1, 28, 28, 1).astype("float32")
x_train = (x_train - 127.5) / 127.5
batch_size = 128
dataset = tf.data.Dataset.from_tensor_slices(x_train).shuffle(60000).batch(batch_size)
noise_dim = 100
generator = tf.keras.Sequential([
    tf.keras.layers.Dense(7*7*256, input_shape=(noise_dim,)),
    tf.keras.layers.Reshape((7, 7, 256)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2DTranspose(128, 5, strides=1, padding='same'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2DTranspose(64, 5, strides=2, padding='same'),
    tf.keras.layers.BatchNormalization(),
```

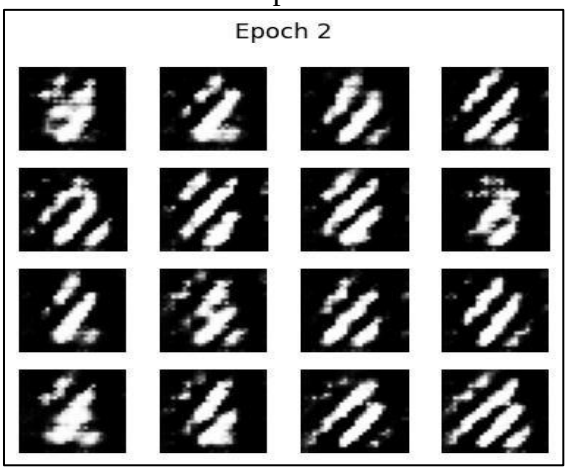
```
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2DTranspose(1, 5, strides=2, padding='same', activation='tanh')
])
discriminator = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, 5, strides=2, padding='same', input_shape=(28, 28, 1)),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Conv2D(128, 5, strides=2, padding='same'),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1)
])
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)
gen_optimizer = tf.keras.optimizers.Adam(1e-4)
disc_optimizer = tf.keras.optimizers.Adam(1e-4)
epochs = 30
seed = tf.random.normal([16, noise_dim])
for epoch in range(epochs):
    print(f"Epoch {epoch+1}/{epochs}")
    for real_images in dataset:
        noise = tf.random.normal([batch_size, noise_dim])
        # Generate fake images
        with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
            fake_images = generator(noise, training=True)
            real_output = discriminator(real_images, training=True)
            fake_output = discriminator(fake_images, training=True)
            gen_loss = loss_fn(tf.ones_like(fake_output), fake_output)
            real_loss = loss_fn(tf.ones_like(real_output), real_output)
            fake_loss = loss_fn(tf.zeros_like(fake_output), fake_output)
            disc_loss = real_loss + fake_loss
        gradients_gen = gen_tape.gradient(gen_loss, generator.trainable_variables)
        gradients_disc = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
        gen_optimizer.apply_gradients(zip(gradients_gen, generator.trainable_variables))
        disc_optimizer.apply_gradients(zip(gradients_disc, discriminator.trainable_variables))
    generated_images = generator(seed, training=False)
    generated_images = (generated_images + 1) / 2.0
    fig = plt.figure(figsize=(4, 4))
    for i in range(16):
        plt.subplot(4, 4, i+1)
        plt.imshow(generated_images[i, :, :, 0], cmap='gray')
        plt.axis('off')
    plt.suptitle(f'Epoch {epoch+1}')
    plt.tight_layout()
    plt.show()
```

Output:

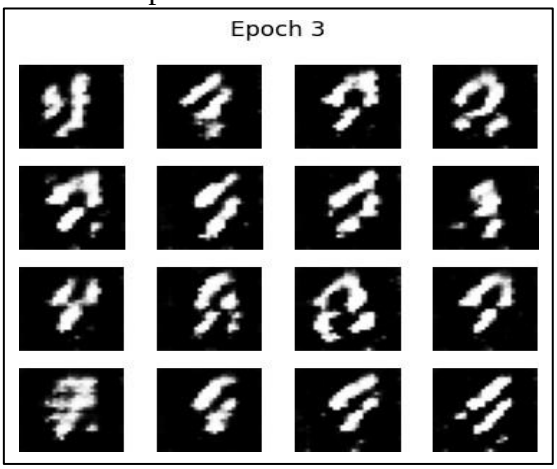
Epoch 1/30



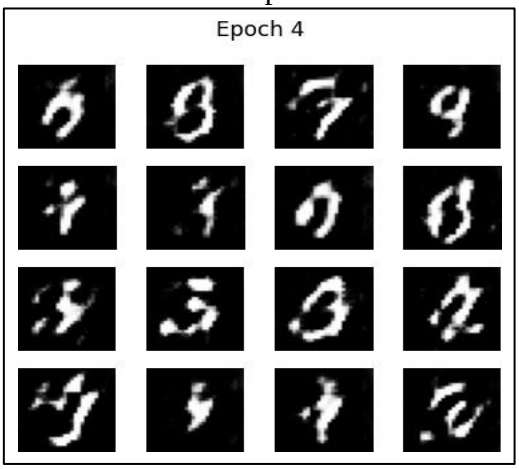
Epoch 2/30



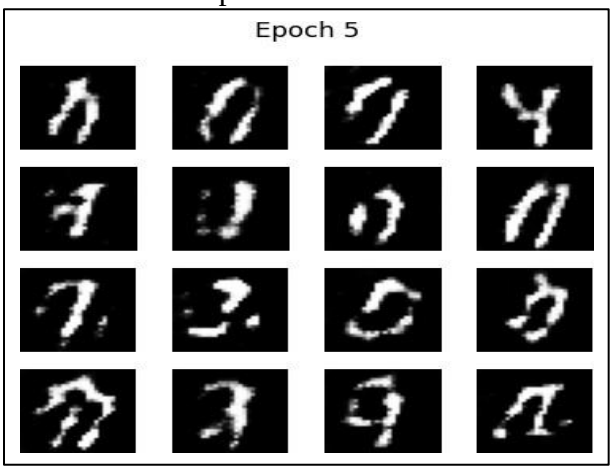
Epoch 3/30



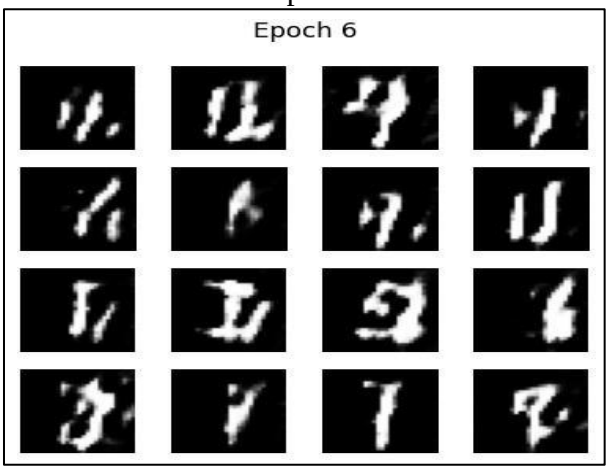
Epoch 4/30



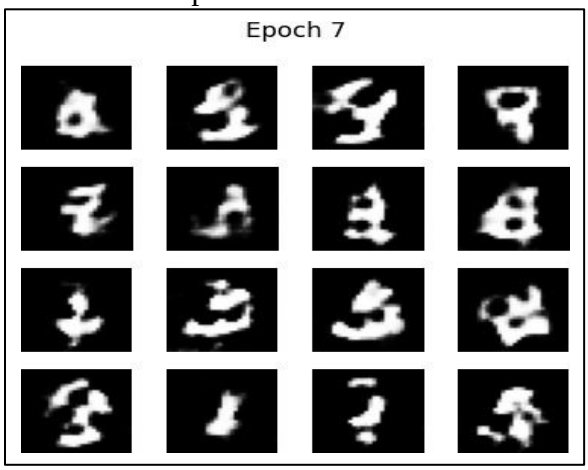
Epoch 5/30



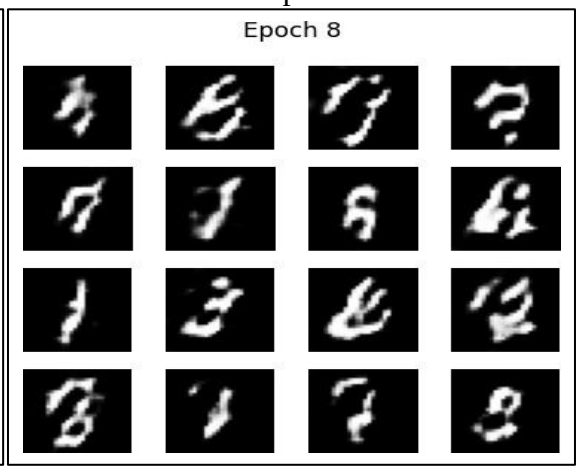
Epoch 6/30



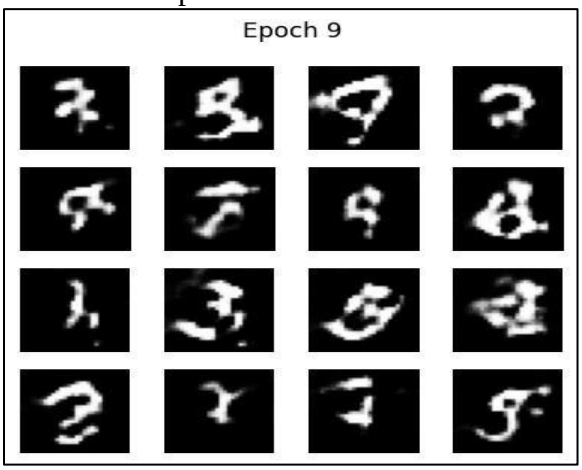
Epoch 7/30



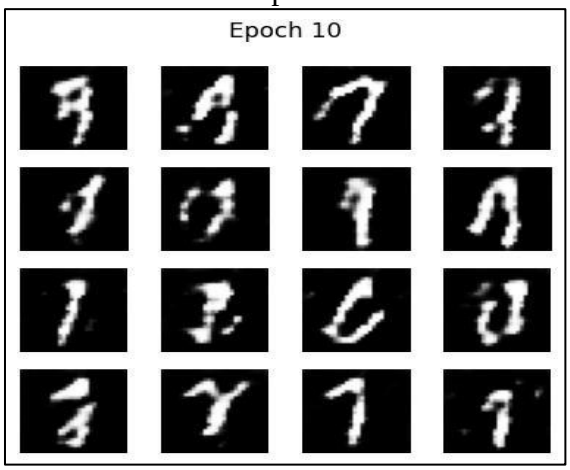
Epoch 8/30



Epoch 9/30



Epoch 10/30



Practical No. 7

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

Writeup:

- **Import Libraries:** It imports numpy for numerical operations and random for random action selection.
- **Environment Setup:**
  - ***grid_size = 4:*** Defines the size of the grid (4x4).
  - ***actions = ['up', 'down', 'left', 'right']:*** Defines the possible actions the agent can take.
  - ***action_dict:*** A dictionary mapping numerical action indices (0-3) to their corresponding changes in row and column coordinates.
- **Q-table Initialization:**
  - ***q_table = np.zeros((grid_size, grid_size, len(actions))):*** This is the core of the Q-learning algorithm. It's a NumPy array (the Q-table) that stores the learned Q-values. The dimensions are (number of rows, number of columns, number of actions). q_table[state_row, state_col, action_index] will store the estimated future reward of taking action_index from the state (state_row, state_col). It's initialized with zeros.
- **Hyperparameters:**
  - ***alpha = 0.1:*** The learning rate. It determines how much the newly acquired information overrides the old information. A higher value means the agent learns faster but might be less stable.
  - ***gamma = 0.9:*** The discount factor. It determines the importance of future rewards. A value closer to 1 means the agent considers future rewards more heavily.
  - ***epsilon = 0.2:*** The exploration rate for the epsilon-greedy policy. It determines the probability of the agent taking a random action instead of the action with the highest Q-value. This encourages exploration of the environment.
  - ***episodes = 500:*** The number of training episodes. An episode is a full run from the start state to the goal state.
- **Reward Function:**
  - ***def get_reward(state)::*** This function defines the reward received for being in a given state.
  - ***if state == (grid_size - 1, grid_size - 1):*** return 10: If the agent reaches the goal state (3,3), it receives a reward of +10, as specified in your problem.
  - ***else:*** return -1: For any other state, the agent receives a reward of -1 for each move, as specified.
- **Next State Function:**
  - ***def next_state(state, action)::*** This function calculates the agent's next state given the current state and the chosen action.
  - It uses the action_dict to determine the change in row and column based on the action.
  - It calculates the new_row and new_col, ensuring the agent stays within the grid boundaries using max(0, min(grid_size - 1, ...)).
  - It returns the (new_row, new_col) as the new_state.

- **Training (Q-learning Algorithm):**
  - ***for episode in range(episodes):***: The main training loop runs for the specified number of episodes.
  - ***state = (0, 0):*** Each episode starts at the initial state (0,0).
  - ***while state != (grid_size - 1, grid_size - 1)::*** The episode continues until the agent reaches the goal state.
  - **Epsilon-Greedy Action Selection:**
    - ***if random.random() < epsilon::*** With probability epsilon, the agent chooses a random action (random.randint(0, 3)).
    - ***else::*** With probability 1 - epsilon, the agent chooses the action with the highest Q-value for the current state (np.argmax(q_table[state[0], state[1]])). This is the greedy action.
    - ***new_state = next_state(state, action):*** Calculate the state the agent moves to based on the chosen action.
    - ***reward = get_reward(new_state):*** Get the reward for being in the new_state.
  - **Q-learning Update Rule:** This is the core learning step:
    - ***old_value = q_table[state[0], state[1], action]:*** Get the current Q-value for the (state, action) pair.
    - ***next_max = np.max(q_table[new_state[0], new_state[1]]):*** Find the maximum Q-value for the new_state across all possible actions. This represents the estimated optimal future reward from the next state.
    - ***q_table[state[0], state[1], action] = old_value + alpha * (reward + gamma * next_max - old_value):*** This is the Q-learning update formula. It updates the old_value based on the reward received, the discounted maximum future reward (gamma * next_max), and the difference between this and the old_value (the temporal difference error), scaled by the alpha learning rate.
    - ***state = new_state:*** Update the current state to the new_state for the next iteration of the while loop.
- **Display Learned Policy:** After training, the code iterates through each state in the grid and prints the action that has the highest learned Q-value (np.argmax(q_table[i, j])). This shows the learned optimal policy for each state.

Code:

```
import numpy as np
import random
# Environment setup
grid_size = 4
actions = ['up', 'down', 'left', 'right']
action_dict = {
   0: (-1, 0),  # up
   1: (1, 0),   # down
   2: (0, -1),  # left
   3: (0, 1)    # right
}
```

```
# Q-table: states are (row, col), actions are 0-3
q_table = np.zeros((grid_size, grid_size, len(actions)))
# Hyperparameters
alpha = 0.1        # Learning rate
gamma = 0.9        # Discount factor
epsilon = 0.2      # Exploration rate
episodes = 500     # Number of episodes
# Reward function
def get_reward(state):
    if state == (grid_size - 1, grid_size - 1):
        return 10
    else:
        return -1
# Next state
def next_state(state, action):
    row, col = state
    dr, dc = action_dict[action]
    new_row = max(0, min(grid_size - 1, row + dr))
    new_col = max(0, min(grid_size - 1, col + dc))
    return (new_row, new_col)
# Training
for episode in range(episodes):
    state = (0, 0)
    while state != (grid_size - 1, grid_size - 1):
        # Epsilon-greedy action selection
        if random.random() < epsilon:
            action = random.randint(0, 3)
        else:
            action = np.argmax(q_table[state[0], state[1]])
        new_state = next_state(state, action)
        reward = get_reward(new_state)
        # Q-learning update
        old_value = q_table[state[0], state[1], action]
        next_max = np.max(q_table[new_state[0], new_state[1]])
        q_table[state[0], state[1], action] = old_value + alpha * (reward + gamma * next_max -
old_value)
        state = new_state
# Display the learned Q-values
for i in range(grid_size):
    for j in range(grid_size):
        best_action = np.argmax(q_table[i, j])
        print(f"({i},{j}): {actions[best_action]}", end=" | ")
    print()
```

Output:

(0,0): down | (0,1): down | (0,2): down | (0,3): down |
(1,0): right | (1,1): down | (1,2): down | (1,3): down |
(2,0): right | (2,1): down | (2,2): down | (2,3): down |
(3,0): right | (3,1): right | (3,2): right | (3,3): up |

Practical No. 8

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

Writeup:

- **Import Libraries:** It imports tensorflow and several components from the transformers library, including BertTokenizer, TFBertForQuestionAnswering, and pipeline. These are essential for loading and using pre-trained transformer models.
- **Define Context and Questions:**
  - ***context:*** This is the block of text from which the model will find answers to the questions. It contains information about various Indian national symbols.
  - ***question:*** This is a list of questions about the information provided in the context.
- **Load Model and Tokenizer:**
  - ***AutoTokenizer.from_pretrained("deepset/bert-base-cased-squad2"):*** This line loads the tokenizer associated with the "deepset/bert-base-cased-squad2" pre-trained model. The tokenizer is responsible for converting text (both the context and the questions) into numerical IDs that the BERT model can understand.
  - ***TFBertForQuestionAnswering.from_pretrained("deepset/bert-base-cased-squad2",from_pt=True):*** This line loads the pre-trained BERT model specifically fine-tuned for the Question Answering task on the SQuAD2.0 dataset. from_pt=True indicates that the model weights are being loaded from a PyTorch checkpoint.
- **Build the Question Answering Pipeline:**
  - ***pipeline('question-answering',model = model,tokenizer = tokenizer):*** The pipeline utility from transformers is used to create a streamlined workflow for the question answering task. It combines the loaded model and tokenizer into a single object that can easily take a question and context as input and return the predicted answer.
- **Define Chatbot Function:**
  - ***def chatbot(question,context)::*** This function encapsulates the question answering process.
  - Inside the function, ***nlp({"question": question, "context": context})*** uses the question answering pipeline to find the answer to the given question within the provided context.
  - ***return output['answer']:*** The function extracts and returns the predicted answer from the pipeline's output.
- **Ask Questions and Print Answers:** The code then iterates through a few of the questions defined earlier, calls the chatbot function with each question and the context, and prints the question along with the answer returned by the chatbot.

Code:

```
#optional → !pip install transformers torch tensorflow
import tensorflow as tf
import transformers
from transformers import BertTokenizer
```

```
from transformers import TFBertForQuestionAnswering
from transformers import AutoTokenizer
from transformers import pipeline
print("Transformers version:", transformers.__version__)
print("TensorFlow version:", tf.__version__)
print("Model path or name:", "deepset/bert-base-cased-squad2")
# This is the given context
context = """
The Bengal tiger was chosen as the national animal in a meeting of the Indian wildlife board in 1972 and was adopted officially in April 1973. It was chosen over the Asiatic lion due to the wider presence of the tiger across India.
Indian peacock was designated as the national bird of India in February 1963.
Indian elephant is the largest terrestrial mammal in India and a cultural symbol throughout its range, appearing in various religious traditions and mythologies.
Indian banyan is a large tree native to the Indian subcontinent and produces aerial roots from the branches which grow downwards, eventually becoming trunks
Mango is a large fruit tree with many varieties, believed to have originated in northeast India.
Lotus is an aquatic plant adapted to grow in the flood plains. Lotus seeds can remain dormant and viable for many years, therefore the plant is regarded as a symbol of longevity.
"""
# These are some questions which we are going to ask to BERT model
question = [
    "What is the National animal of India?",
    "What is the National bird of India?",
    "What is the largest terrestrial mammal in India?",
    "Which tree produces aerial roots?",
    "Which fruit originated in northeast India?",
    "What is the symbol of longevity?",
]
# Importing the tokenizer of the BERT model
tokenizer = AutoTokenizer.from_pretrained("deepset/bert-base-cased-squad2")
# Importing the BERT model
model = TFBertForQuestionAnswering.from_pretrained("deepset/bert-base-cased-
squad2",from_pt=True)
# Tokenizing the first question and print the encoded output
tokenizer.encode(question[0],truncation = True, padding = True)
# Building the BERT "Question - Answering" pipeline
nlp = pipeline('question-answering',model = model,tokenizer = tokenizer)
# Making a chatbot function for question answering
def chatbot(question,context):
  output = nlp({
      "question": question,
      "context": context
  })
  return output['answer']
# Asking the first question to the chatbot
```

```
print("Question: ",question[1])
print("Answer: ",chatbot(question[1],context))
print("Question: ",question[2])
print("Answer: ",chatbot(question[2],context))
print("Question: ",question[3])
print("Answer: ",chatbot(question[3],context))
print("Question: ",question[4])
print("Answer: ",chatbot(question[4],context))
print("Question: ",question[5])
print("Answer: ",chatbot(question[5],context))
```

Output:

```
Transformers version: 4.55.4
TensorFlow version: 2.19.0
Model path or name: deepset/bert-base-cased-squad2

[101, 1327, 1110, 1103, 1305, 3724, 1104, 1726, 136, 102]
Question:  What is the National bird of India?
Answer:  Indian peacock
Question:  What is the largest terrestrial mammal in India?
Answer:  Indian elephant
Question:  Which tree produces aerial roots?
Answer:  Indian banyan
Question:  Which fruit originated in northeast India?
Answer:  Mango
Question:  What is the symbol of longevity?
Answer:  Lotus
```

Practical No. 9

Aim: Building a deep learning model for time series forecasting.

Writeup:

- **Import Libraries:** It imports numpy for numerical operations, tensorflow and Keras components (Sequential, Dense) for building the neural network, and MinMaxScaler from sklearn.preprocessing for data scaling.
- **Sample Time Series Data:** ***sales = np.array(...)*** defines your time series data. In this case, it's a simple sequence of sales values over time. The .reshape(-1, 1) is to make it a 2D array, which is often required for scikit-learn's MinMaxScaler.
- **Data Scaling:**
  - ***scaler = MinMaxScaler():*** An instance of MinMaxScaler is created. This scaler will transform the data so that all values are within a specific range, typically [0, 1]. Scaling is important for many neural networks as it can help with training stability and performance.
  - ***sales_scaled = scaler.fit_transform(sales).flatten():*** The scaler is fitted to your sales data (learning the min and max values) and then transforms the data. .flatten() converts the 2D scaled array back into a 1D array.
- **Prepare Supervised Data (Windowing):**
  - ***n_steps = 3:*** This defines the size of the "window" or the number of past time steps used to predict the next time step. In this code, it uses the last 3 time steps to predict the 4th.
  - The for loop iterates through the scaled time series data to create input-output pairs for supervised learning.
  - ***X.append(sales_scaled[i:i+n_steps]):*** For each position i, a sequence of n_steps values starting from i is extracted and added to the input list X. This represents the historical data points in the window.
  - ***y.append(sales_scaled[i+n_steps]):*** The value immediately following the window (i + n_steps) is extracted and added to the output list y. This is the value the model will try to predict.
  - ***X = np.array(X) and y = np.array(y):*** The lists of inputs and outputs are converted into NumPy arrays, which are required for training a Keras model. X will be a 2D array where each row is a window of past values, and y will be a 1D array of the corresponding next values.
- **Build Deep Learning Model:** ***model = Sequential([...])*** defines a simple feedforward neural network.
  - ***Dense(4, activation='relu', input_shape=(n_steps,)):*** The first dense layer has 4 neurons and uses the ReLU activation function. input_shape=(n_steps,) tells the model that the input to this layer will be sequences of length n_steps (the size of your window).
  - ***Dense(1):*** The output layer has 1 neuron, as you are predicting a single future value.
- **Compile Model:** ***model.compile(optimizer='adam', loss='mse'):*** The model is compiled with the adam optimizer and Mean Squared Error (mse) loss function, which is commonly used for regression tasks like forecasting.

- **Train Model:** ***model.fit(X, y, epochs=500, verbose=0):*** The model is trained on the prepared supervised data (X and y) for 500 epochs. verbose=0 keeps the training progress output silent.
- **Make Prediction:**
  - ***x_input = np.array([18, 17, 19]).reshape(-1, 1):*** A new input sequence (the last 3 original sales values) is defined for prediction. It's reshaped for scaling.
  - ***x_input_scaled = scaler.transform(x_input).flatten().reshape(1, n_steps):*** This new input is scaled using the *same* scaler fitted on the training data. It's then flattened and reshaped into the correct input shape for the model (1 sample, n_steps features).
  - ***y_pred_scaled = model.predict(x_input_scaled, verbose=0):*** The trained model makes a prediction on the scaled input. The output is a scaled prediction.
  - ***y_pred = scaler.inverse_transform([[y_pred_scaled[0][0]]])[0][0]:*** The predicted value is currently scaled. This line uses the inverse_transform method of the scaler to bring the predicted value back to the original sales scale. The nested indexing is to handle the shape of the scaler's input and output.
- **Print Prediction:** ***print(f"Predicted sales on day 11: {y_pred:.2f}"):*** The final predicted sales value on the original scale is printed.

Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.preprocessing import MinMaxScaler
# Original sales data
sales = np.array([10, 12, 13, 12, 14, 15, 16, 18, 17, 19]).reshape(-1, 1)
# Scale sales data to [0,1]
scaler = MinMaxScaler()
sales_scaled = scaler.fit_transform(sales).flatten()
# Prepare supervised data
X = []
y = []
n_steps = 3
for i in range(len(sales_scaled) - n_steps):
    X.append(sales_scaled[i:i+n_steps])
    y.append(sales_scaled[i+n_steps])
X = np.array(X)
y = np.array(y)
# Build model
model = Sequential([
    Dense(4, activation='relu', input_shape=(n_steps,)),
    Dense(1) ])
model.compile(optimizer='adam', loss='mse')
# Train model
model.fit(X, y, epochs=500, verbose=0)
```

```
# Predict the sales for day 11 using last 3 days [18, 17, 19]
x_input = np.array([18, 17, 19]).reshape(-1, 1)
x_input_scaled = scaler.transform(x_input).flatten().reshape(1, n_steps)
y_pred_scaled = model.predict(x_input_scaled, verbose=0)
y_pred_scaled
y_pred = scaler.inverse_transform([[y_pred_scaled[0][0]]])[0][0]
y_pred
print(f"Predicted sales on day 11: {y_pred:.2f}")
```

Output:

```
array([[0.34281263]], dtype=float32)
np.float64(13.08531364798546)
Predicted sales on day 11: 13.09
```

Practical No. 10

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

Writeup:

- **Import Libraries:** It imports random for generating random numbers needed for selection, crossover, and mutation.
- **Initial Population:**
  - ***population = [...]:*** This list represents the initial "population" of candidate solutions. Each dictionary in the list represents an "individual" in the population, and its keys (id, hidden, learning rate) represent its "genes" or the specific combination of hyperparameters being tested. The comments indicate assumed fitness values, although the code calculates fitness later.
- **Define Parameters:**
  - ***generations = 5:*** The number of iterations the evolutionary process will run. Each generation involves evaluating the current population, selecting parents, creating offspring, and forming a new population.
  - ***mutation_rate = 0.4:*** The probability that a gene (hyperparameter value) in an offspring will be randomly changed (mutated).
  - ***hidden_bounds = (10, 100)*** and ***learningrate_bounds = (0.001, 0.1):*** These define the valid range of values for the hidden and learning rate hyperparameters. Mutation and crossover results are constrained within these bounds.
- **Training Loop (Generations):** for gen in range(generations): The main loop runs for the specified number of generations.
- **Fitness Calculation:**
  - ***fitness_scores = []:*** An empty list to store the fitness score for each individual in the current population.
  - The inner loop iterates through each ind (individual) in the population.
  - ***fitness = 0.8 + (ind['hidden'] / 200) - abs(ind['learningrate'] - 0.03) * 2:*** This is the fitness function. In a real hyperparameter tuning scenario, this function would represent the performance of a model trained with these hyperparameters (e.g., accuracy on a validation set). Here, it's a simplified mathematical formula that rewards individuals with hidden values closer to 200 and learning rate values closer to 0.03. The goal of the genetic algorithm is to find individuals that maximize this fitness score.
  - ***fitness_scores.append(fitness):*** The calculated fitness for each individual is added to the list.
- **Selection:**
  - ***sorted_indices = sorted(...):*** The indices of the individuals are sorted based on their fitness_scores in descending order (highest fitness first).
  - ***p1 = population[sorted_indices[0]]*** and ***p2 = population[sorted_indices[1]]:*** The two individuals with the highest fitness scores (the "fittest") are selected as "parents" for the next generation. This is a simple form of selection called truncation selection.
- **Crossover (Recombination):**

- ***child_hidden = int((p1['hidden'] + p2['hidden']) / 2)*** and ***child_learningrate = round((p1['learningrate'] + p2['learningrate']) / 2, 4)***: A new individual (the "child") is created by combining the genes (hyperparameter values) of the two selected parents. Here, it's a simple averaging of the hidden and learning rate values. More complex crossover strategies exist in genetic algorithms.

- **Mutation:**
  - ***if random.random() < mutation_rate::*** With a probability determined by mutation_rate, the child's genes are subjected to mutation.
  - ***child_hidden += random.randint(-5, 5)*** and ***child_learningrate += round(random.uniform(-0.01, 0.01), 4):*** Random noise is added to the child's hyperparameter values within a small range.
  - ***max(min(..., bounds[1]), bounds[0]):*** The mutated values are then clamped to stay within the defined hidden_bounds and learningrate_bounds. Mutation introduces diversity into the population and helps the algorithm explore the search space to avoid getting stuck in local optima.
- **Evaluate Child: *child_fitness = 0.8 + (...):*** The fitness of the newly created child is calculated using the same fitness function.
- **Replacement (Forming the Next Generation):**
  - ***worst_idx = fitness_scores.index(min(fitness_scores)):*** The index of the individual with the lowest fitness in the current population is found.
  - ***if child_fitness > fitness_scores[worst_idx]:*** population[worst_idx] = {'id': '*', ...}: If the child's fitness is better than the worst individual in the current population, the child replaces the worst individual. This ensures that the population, on average, becomes fitter over generations.
  - ***else:*** print(...): If the child is not better than the worst, it is not included in the next generation.
- **Final Population:** After all generations, the code prints the hyperparameters and fitness of the individuals in the final population. The individual with the highest fitness in the final population represents the best combination of hyperparameters found by the algorithm.

Code:

```
import random
# Initial population (dataset)
population = [
   {'id': 'A', 'hidden': 20, 'learningrate': 0.01},  # Assume fitness = 0.82
   {'id': 'B', 'hidden': 50, 'learningrate': 0.05},  # Assume fitness = 0.87
   {'id': 'C', 'hidden': 80, 'learningrate': 0.02},  # Assume fitness = 0.78
]
population
# Fitness function calculation  0.8 + (hidden / 200) - abs(lr - 0.03) * 2
generations = 5
mutation_rate = 0.4
hidden_bounds = (10, 100)
learningrate_bounds = (0.001, 0.1)
for gen in range(generations):
   print(f"\n--- Generation {gen + 1} ---")
```

```
    # Calculate fitness
    fitness_scores = []
    for ind in population:
        fitness = 0.8 + (ind['hidden'] / 200) - abs(ind['learningrate'] - 0.03) * 2
        fitness_scores.append(fitness)
    # Print population
    for i, ind in enumerate(population):
        print(f"ID {ind['id']}: hidden={ind['hidden']}, learningrate={ind['learningrate']:.4f},
fitness={fitness_scores[i]:.4f}")
    # Select best two individuals
    sorted_indices = sorted(range(len(fitness_scores)), key=lambda i: fitness_scores[i],
reverse=True)
    p1 = population[sorted_indices[0]]
    p2 = population[sorted_indices[1]]
    print(f"\nSelected parents: {p1['id']} and {p2['id']}")
    # Crossover
    child_hidden = int((p1['hidden'] + p2['hidden']) / 2)
    child_learningrate = round((p1['learningrate'] + p2['learningrate']) / 2, 4)
    # Mutation
    if random.random() < mutation_rate:
        child_hidden += random.randint(-5, 5)
        child_hidden = max(min(child_hidden, hidden_bounds[1]), hidden_bounds[0])
    if random.random() < mutation_rate:
        child_learningrate += round(random.uniform(-0.01, 0.01), 4)
        child_learningrate = max(min(child_learningrate, learningrate_bounds[1]),
learningrate_bounds[0])
    # Evaluate child
    child_fitness = 0.8 + (child_hidden / 200) - abs(child_learningrate - 0.03) * 2
    print(f"\nChild: hidden={child_hidden}, learningrate={child_learningrate:.4f},
fitness={child_fitness:.4f}")
    # Replace worst if better
    worst_idx = fitness_scores.index(min(fitness_scores))
    if child_fitness > fitness_scores[worst_idx]:
        print(f"Child replaces individual {population[worst_idx]['id']}")
        population[worst_idx] = {'id': '*', 'hidden': child_hidden, 'learningrate': child_learningrate}
    else:
        print("Child is not better than the worst individual. No replacement.")
# Final population
print("\n=== Final Population ===")
for ind in population:
    final_fitness = 0.8 + (ind['hidden'] / 200) - abs(ind['learningrate'] - 0.03) * 2
    print(f"ID {ind['id']}: hidden={ind['hidden']}, learningrate={ind['learningrate']:.4f},
fitness={final_fitness:.4f}")
```

Output:

```
[{'id': 'A', 'hidden': 20, 'learningrate': 0.01},
 {'id': 'B', 'hidden': 50, 'learningrate': 0.05},
 {'id': 'C', 'hidden': 80, 'learningrate': 0.02}]

--- Generation 1 ---
ID A: hidden=20, learningrate=0.0100, fitness=0.8600
ID B: hidden=50, learningrate=0.0500, fitness=1.0100
ID C: hidden=80, learningrate=0.0200, fitness=1.1800

Selected parents: C and B

Child: hidden=65, learningrate=0.0350, fitness=1.1150
Child replaces individual A

--- Generation 2 ---
ID *: hidden=65, learningrate=0.0350, fitness=1.1150
ID B: hidden=50, learningrate=0.0500, fitness=1.0100
ID C: hidden=80, learningrate=0.0200, fitness=1.1800

Selected parents: C and *

Child: hidden=72, learningrate=0.0312, fitness=1.1576
Child replaces individual B

--- Generation 3 ---
ID *: hidden=65, learningrate=0.0350, fitness=1.1150
ID *: hidden=72, learningrate=0.0312, fitness=1.1576
ID C: hidden=80, learningrate=0.0200, fitness=1.1800

Selected parents: C and *

Child: hidden=72, learningrate=0.0256, fitness=1.1512
Child replaces individual *

--- Generation 4 ---
ID *: hidden=72, learningrate=0.0256, fitness=1.1512
ID *: hidden=72, learningrate=0.0312, fitness=1.1576
ID C: hidden=80, learningrate=0.0200, fitness=1.1800

Selected parents: C and *

Child: hidden=76, learningrate=0.0256, fitness=1.1712
Child replaces individual *

--- Generation 5 ---
```

ID *: hidden=76, learningrate=0.0256, fitness=1.1712
ID *: hidden=72, learningrate=0.0312, fitness=1.1576
ID C: hidden=80, learningrate=0.0200, fitness=1.1800

Selected parents: C and *

Child: hidden=78, learningrate=0.0228, fitness=1.1756
Child replaces individual *

=== Final Population ===
ID *: hidden=76, learningrate=0.0256, fitness=1.1712
ID *: hidden=78, learningrate=0.0228, fitness=1.1756
ID C: hidden=80, learningrate=0.0200, fitness=1.1800

Practical No. 11

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.

Writeup:

- **Import Libraries:** It imports numpy, pandas, tensorflow, and warnings (though these aren't directly used in the text generation itself) and crucially, components from the transformers library: GPT2LMHeadModel, GPT2Tokenizer, and pipeline. These are necessary for working with pre-trained transformer models like GPT-2.
- **Define Prompt:** ***text = "Once upon a time, in a magical forest, there lived a curious elf named Elara. She had":*** This line defines the initial string of text that the GPT-2 model will use as a starting point for generating new text.
- **Define generate_text Function:** This function encapsulates the text generation process.
  - ***generator = pipeline('text-generation', model='gpt2'):*** This line is key. It utilizes the pipeline utility from the transformers library to set up a text generation task. It automatically loads the pre-trained gpt2 model and its corresponding tokenizer. This simplifies the process of using the model for generation.
  - ***generated_text = generator(prompt, max_length=200, num_return_sequences=1, truncation=True):*** This line performs the text generation.
  - ***prompt:*** The starting text provided to the model.
  - ***max_length=200:*** Specifies the maximum length of the generated text (including the prompt).
  - ***num_return_sequences=1:*** Requests only one generated output sequence.
  - ***truncation=True:*** Ensures that the prompt is truncated if it exceeds the model's maximum input length.
  - ***return generated_text[0]['generated_text']:*** The function extracts and returns the generated text string from the pipeline's output.
- **Generate and Display Text:**
  - ***output = generate_text(text):*** The generate_text function is called with the defined starting text, and the generated output is stored in the output variable.
  - ***output:*** In a Colab code cell, having a variable name as the last line displays its value, showing the generated text.

Code:

```
import numpy as np
import pandas as pd
import tensorflow as tf
import warnings
from transformers import GPT2LMHeadModel, GPT2Tokenizer, pipeline
text = "Once upon a time, in a magical forest, there lived a curious elf named Elara. She had"
#text= input("Enter your sentence here\n")
def generate_text(prompt):
    generator = pipeline('text-generation', model='gpt2')
```

```
    generated_text = generator(prompt, max_length=200, num_return_sequences=1,
truncation=True)
    return generated_text[0]['generated_text']
    #print(generated_text[0]['generated_text'])
output=generate_text(text)
output
```

Output:

Once upon a time, in a magical forest, there lived a curious elf named Elara. She had a strange, seemingly random nature and, when she came across the young man she suddenly felt as if she were on a journey to find out more about him.

Elara had long since become a magical girl and was known for being a very good at reading and writing. By the time she was twenty-two years old, she had found a place to live in a small village called Welt. The village was located on an island in the middle of the continent and Elara had been taken there by the spirit of the elves she had met on the island.

In the short time she had been there, Elara had learned that the village was haunted. The spirit of the elves she had met on the island had been the one that had made it so that no one found her.

So she was sent to the village of Welt, and to the world she had come to live in. The spirit of the elves she had met on the island had been that of a young boy named Elara.

The magical elves were not the only ones who had witnessed the spirit of the elves on the island. It was obvious that the spirit of the elves had also been in the village. The elves had come to the village as a group.

Practical No. 12

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.

Writeup:

- **Import Libraries**: It imports tensorflow for building and training the neural networks, numpy for numerical operations, and matplotlib.pyplot for visualizing the generated images.
- **Load and Preprocess Data**: The MNIST dataset of handwritten digits is loaded. The images are reshaped to have a channel dimension and normalized to the range [-1, 1]. Normalizing to this range is common in GANs, especially when using tanh activation in the generator's output layer. A tf.data.Dataset is created for efficient batching and shuffling of the training data.
- **Define Hyperparameters:** batch_size, noise_dim (the size of the random noise vector fed to the generator), and epochs (the number of training cycles) are defined.
- **Define the Generator Model:** ***generator = tf.keras.Sequential([...])*** defines the generator network. Its purpose is to take a random noise vector (noise_dim) and transform it into an image that looks like the training data.
    - It starts with a Dense layer to project the noise into a higher dimension.
    - Reshape changes the output shape to a 3D tensor (like a small image with many channels).
    - BatchNormalization helps stabilize training.
    - LeakyReLU is an activation function commonly used in GANs.
    - Conv2DTranspose (also known as deconvolution or upsampling) layers are used to increase the spatial dimensions of the data, gradually building up an image from the smaller representation. The strides greater than 1 (strides=2) are key to upsampling.
    - The final Conv2DTranspose layer outputs a single-channel image (grayscale) with a tanh activation function, which outputs values in the range [-1, 1], matching the normalized input data.
- **Define the Discriminator Model:** ***discriminator = tf.keras.Sequential([...])*** defines the discriminator network. Its purpose is to take an image (either a real image from the dataset or a fake image from the generator) and output a single value indicating whether it thinks the image is real (closer to 1) or fake (closer to 0).
    - Conv2D layers with strides greater than 1 (strides=2) are used to downsample the image and extract features.
    - LeakyReLU and Dropout (to prevent overfitting) are used.
    - Flatten converts the 2D feature maps into a 1D vector.
    - The final Dense layer outputs a single value.
- **Define Loss Function and Optimizers:**
    - ***loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True):*** Binary crossentropy is used as the loss function. from_logits=True is important because the discriminator's final Dense layer does not have an activation function applied (it outputs logits).

- ***gen_optimizer*** and ***disc_optimizer:*** Separate Adam optimizers are defined for the generator and discriminator, as they are trained adversarially.

- **Training Loop**: The code enters a loop for the specified number of epochs.
  - Inside the epoch loop, it iterates through batches of real images from the dataset.
  - For each batch, random noise is generated.
  - **Training the Discriminator:**
    - A tf.GradientTape is used to record operations for automatic differentiation.
    - Fake images are generated by the generator.
    - The discriminator is called on both real and fake images.
    - Losses are calculated: real_loss (discriminator trying to classify real images as real) and fake_loss (discriminator trying to classify fake images as fake). The total disc_loss is the sum of these.
    - Gradients of the disc_loss with respect to the discriminator's trainable variables are computed and applied using the disc_optimizer.
  - **Training the Generator:**
    - Another tf.GradientTape is used.
    - Fake images are generated.
    - The discriminator is called on the fake images.
    - gen_loss is calculated: the generator tries to make the discriminator classify the fake images as real (tf.ones_like(fake_output)).
    - Gradients of the gen_loss with respect to the generator's trainable variables are computed and applied using the gen_optimizer.
- **Visualization:** After each epoch, the generator is used to generate a sample of images from a fixed seed noise vector (to see how the generation improves over epochs). These images are rescaled back to [0, 1] and displayed using matplotlib.pyplot.

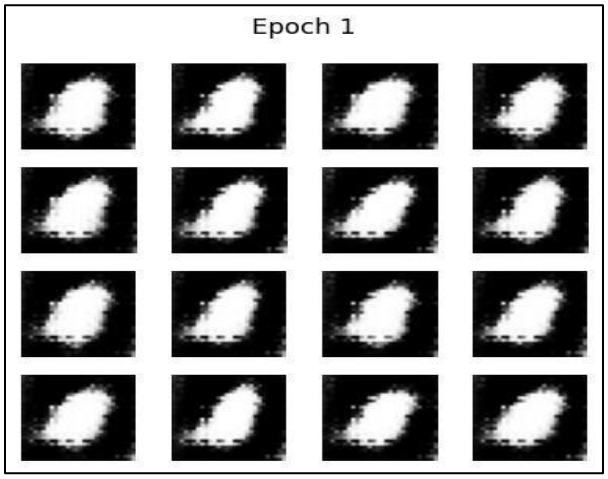
Code:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
(x_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape(-1, 28, 28, 1).astype("float32")
x_train = (x_train - 127.5) / 127.5
batch_size = 128
dataset = tf.data.Dataset.from_tensor_slices(x_train).shuffle(60000).batch(batch_size)
noise_dim = 100
generator = tf.keras.Sequential([
    tf.keras.layers.Dense(7*7*256, input_shape=(noise_dim,)),
    tf.keras.layers.Reshape((7, 7, 256)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2DTranspose(128, 5, strides=1, padding='same'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2DTranspose(64, 5, strides=2, padding='same'),
    tf.keras.layers.BatchNormalization(),
```

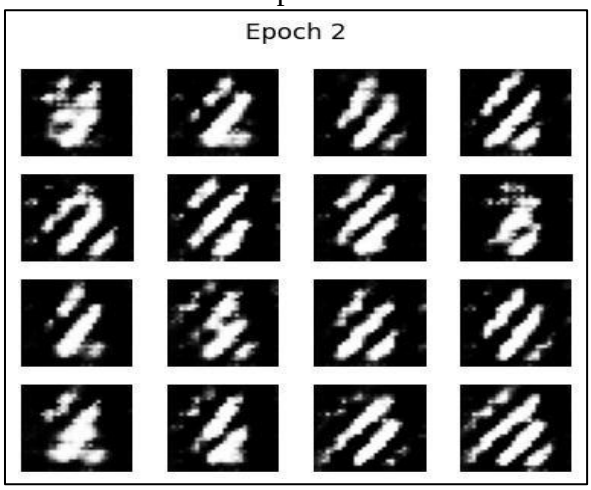
```
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2DTranspose(1, 5, strides=2, padding='same', activation='tanh')
])
discriminator = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, 5, strides=2, padding='same', input_shape=(28, 28, 1)),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Conv2D(128, 5, strides=2, padding='same'),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1)
])
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)
gen_optimizer = tf.keras.optimizers.Adam(1e-4)
disc_optimizer = tf.keras.optimizers.Adam(1e-4)
epochs = 30
seed = tf.random.normal([16, noise_dim])
for epoch in range(epochs):
    print(f"Epoch {epoch+1}/{epochs}")
    for real_images in dataset:
        noise = tf.random.normal([batch_size, noise_dim])
        # Generate fake images
        with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
            fake_images = generator(noise, training=True)
            real_output = discriminator(real_images, training=True)
            fake_output = discriminator(fake_images, training=True)
            gen_loss = loss_fn(tf.ones_like(fake_output), fake_output)
            real_loss = loss_fn(tf.ones_like(real_output), real_output)
            fake_loss = loss_fn(tf.zeros_like(fake_output), fake_output)
            disc_loss = real_loss + fake_loss
        gradients_gen = gen_tape.gradient(gen_loss, generator.trainable_variables)
        gradients_disc = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
        gen_optimizer.apply_gradients(zip(gradients_gen, generator.trainable_variables))
        disc_optimizer.apply_gradients(zip(gradients_disc, discriminator.trainable_variables))
    generated_images = generator(seed, training=False)
    generated_images = (generated_images + 1) / 2.0
    fig = plt.figure(figsize=(4, 4))
    for i in range(16):
        plt.subplot(4, 4, i+1)
        plt.imshow(generated_images[i, :, :, 0], cmap='gray')
        plt.axis('off')
    plt.suptitle(f'Epoch {epoch+1}')
    plt.tight_layout()
    plt.show()
```

Output:

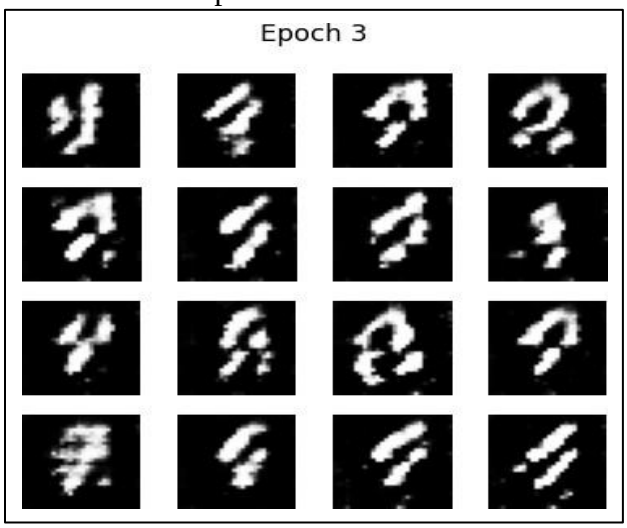
Epoch 1/30



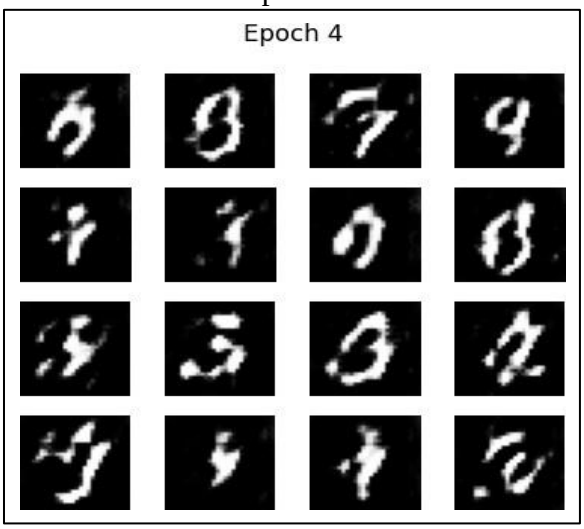
Epoch 2/30



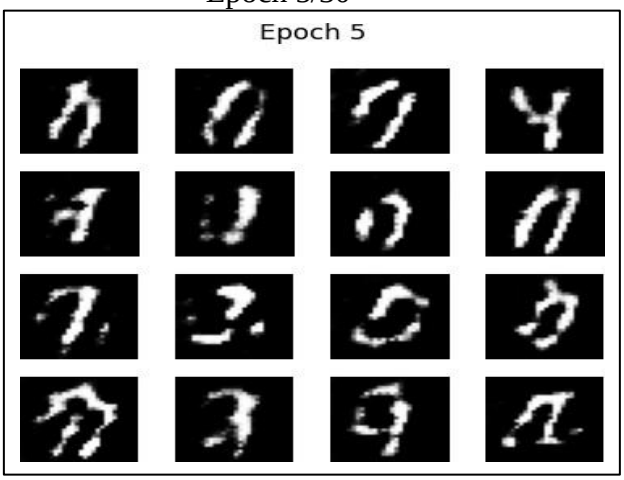
Epoch 3/30



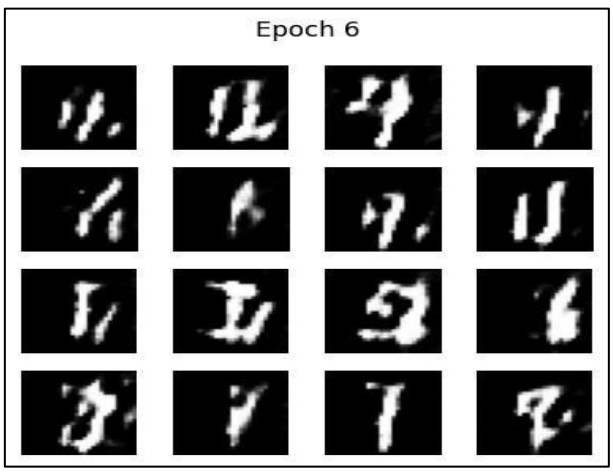
Epoch 4/30



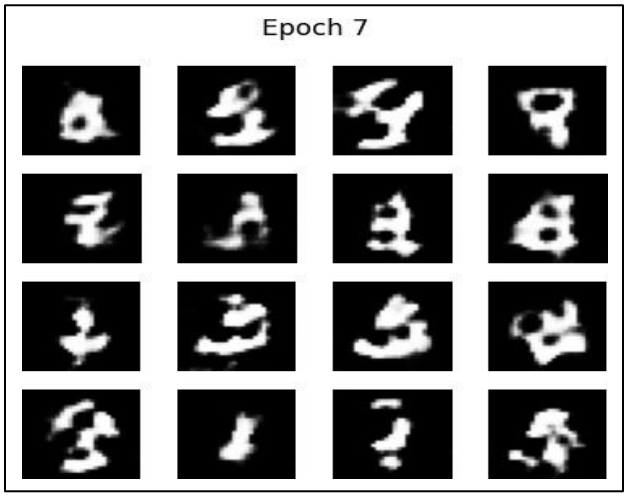
Epoch 5/30



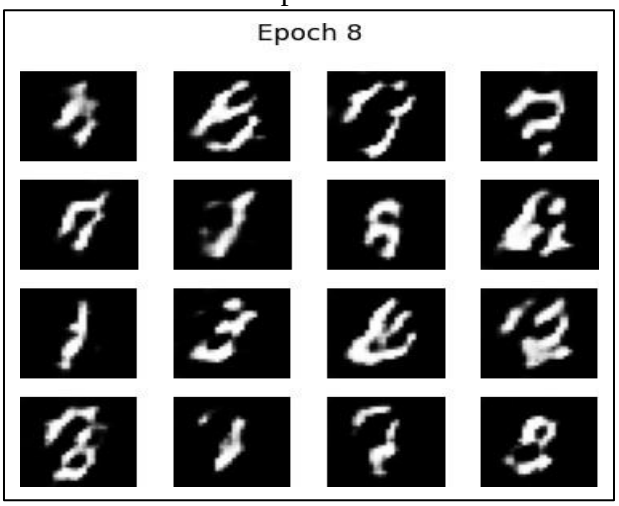
Epoch 6/30



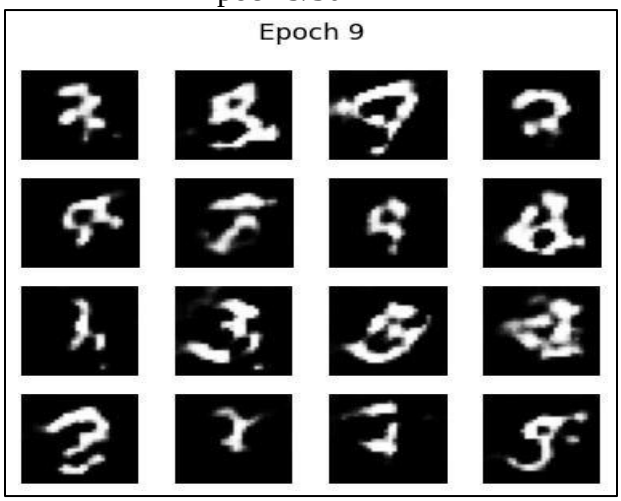
Epoch 7/30



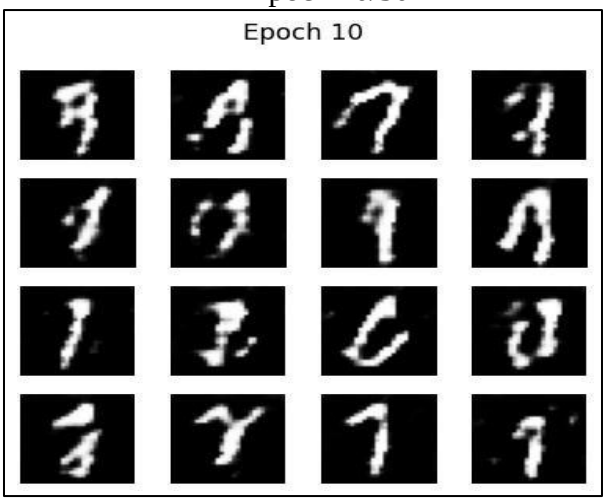
Epoch 8/30



Epoch 9/30



Epoch 10/30



NURTURING POTENTIAL

SAKET GYANPEETH'S

SAKET COLLEGE OF ARTS, SCIENCE AND COMMERCE

(Permanently Affiliated to University of Mumbai)

NAAC Accredited

Saket Vidyanagri Marg, Chinchpada Road, Katemanivali,
Kalyan (East) -421306(Mah)

Department of Information Technology

This is to certify that

Mr./Ms. YASH PRAMOD GAIKWAD Seat No. 254337

of MACHINE LEARNING

M.Sc. Information Technology

Part II NEP 2020 Semester III

has satisfactory carried out the required practical in the subject
of
For the Academic year 2025 – 2026

Practical In-Charge

Head of the Department

External Examiner

College Seal

INDEX

Sr.no.	Practical	Date	Sign
1.	**Data Pre-processing and Exploration** **a.** Load a CSV dataset. Handle missing values, inconsistent formatting, and outliers. **b.** 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 **c.** Create or Explore datasets to use all pre-processing routines like label encoding, scaling, and binarization.		
2.	**Testing Hypothesis** **a.** 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** **a.** 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 **b.** Multiple Linear Regression Extend linear regression to multiple feature. Handle feature selection and potential multicollinearity **c.** Regualarized Linear Models Implement Regression variants like LASSO and Ridge on any generated dataset		

4.	**Discriminative Models** **a.** Logistic Regression : Perform binary classification using logistic regression. Calculate accuracy, precision, recall, and understand the ROC curve." **b.** 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. **c.** Build a decision tree classifier or regressor. Control hyperparameters like tree depth to avoid overfitting. Visualize the tree. **d.** Implement a Support Vector Machine for any relevant dataset. **e.** Train a random forest ensemble. Experiment with the number of trees and feature sampling. Compare performance to a single decision tree. **f.** Implement a gradient boosting machine (e.g., XGBoost). Tune hyperparameters and explore feature importance.		
5.	**Generative Models** **a.** Implement and demonstrate the working of a Naive Bayesian classifier using a sample data set. Build the model to classify a test sample. **b.** Implement Hidden Markov Models using hmmlearn		
6.	**Probabilistic Models** **a.** Implement Bayesian Linear Regression to explore prior and posterior distribution. **b.** Implement Gaussian Mixture Models for density estimation and unsupervised clustering.		
7.	**Model Evaluation and Hyperparameter Tuning** **a.** Implement cross-validation techniques (k-fold, stratified, etc.) for robust model evaluation		

	b. Systematically explore combinations of hyperparameters to optimize model performance.(use grid and randomized search)		
8.	**Bayesian Learning** **a.** 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
```

2. **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())
```

3. **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())
```

4. **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())
```

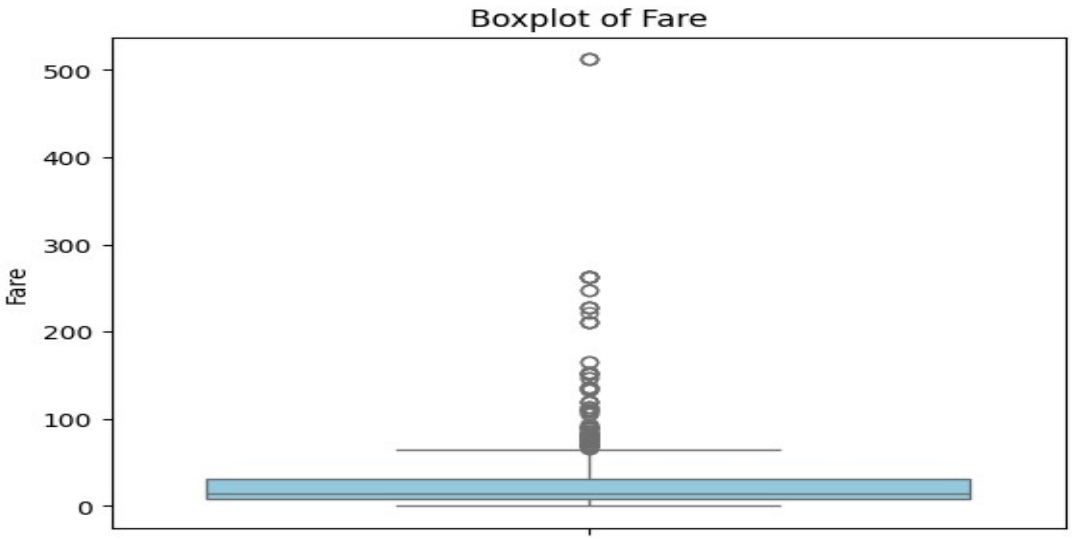
5. **Detect and Handle Outliers** `# Boxplot for the 'Fare' column sns.boxplot(data['Fare'], color='skyblue') plt.title('Boxplot of Fare') plt.show()`

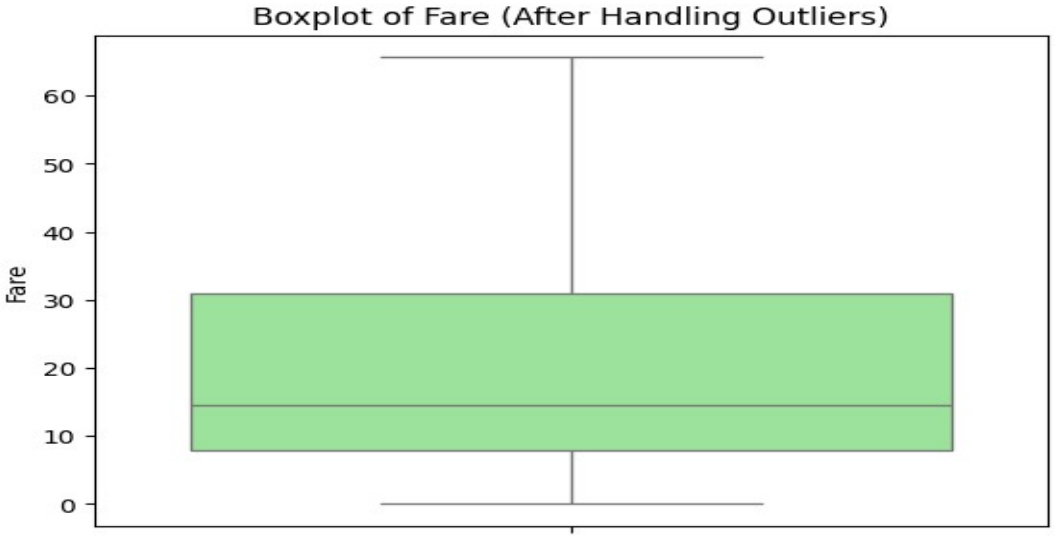
```
# Detect outliers using the IQR method
Q1 = data['Fare'].quantile(0.25)
Q3 = data['Fare'].quantile(0.75)
IQR = Q3 - Q1  lower_bound =
Q1 - 1.5 * IQR  upper_bound =
Q3 + 1.5 * IQR
# Capping outliers
data['Fare'] = np.where(data['Fare'] > upper_bound, upper_bound,   np.where(data['Fare'] <
lower_bound, 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()
```

6. **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 :


Boxplot of Fare
Fare
500
400
300
200
100
0


Boxplot of Fare (After Handling Outliers)
Fare
60
50
40
30
20
10
0

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
```

2. 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())
```

3. 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())
```

4. 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()
```

5. 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()
```

6. 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

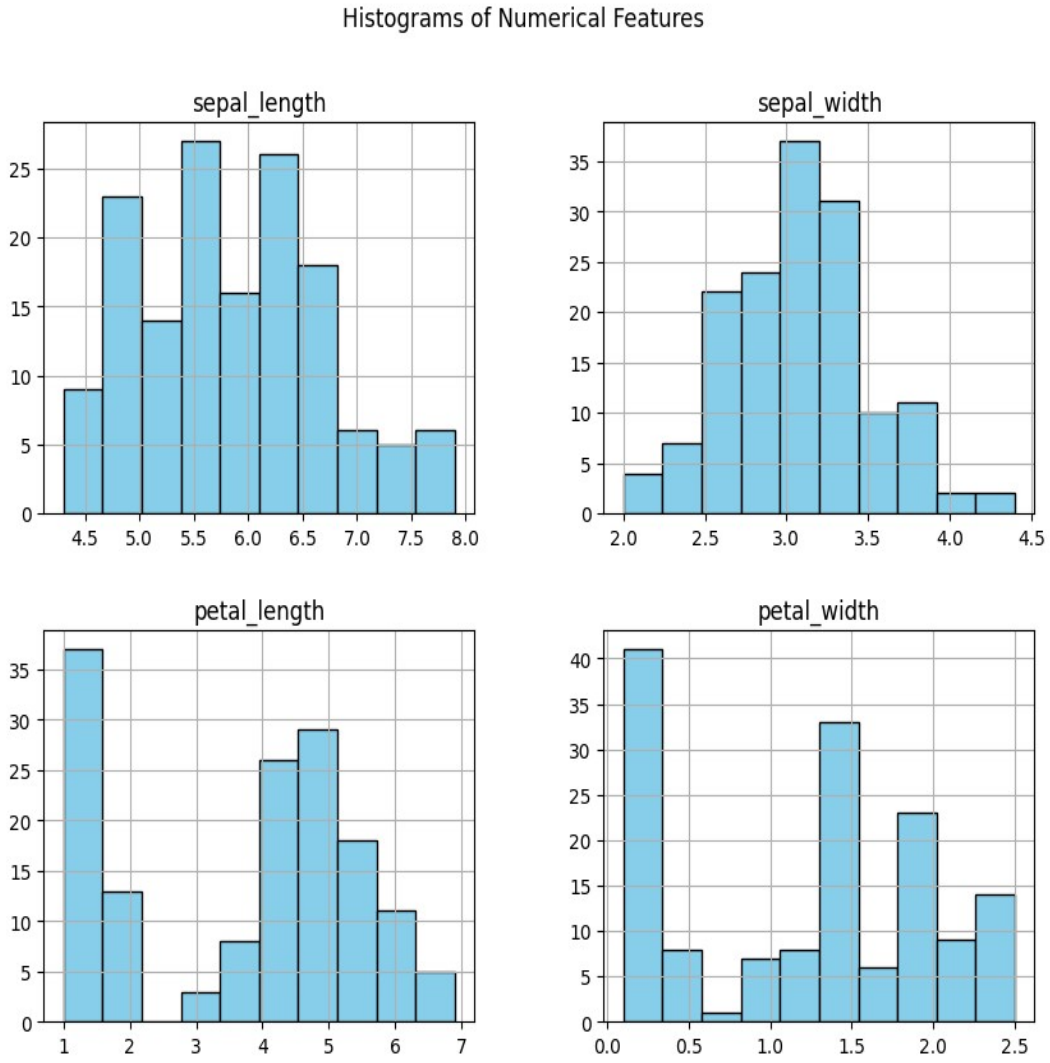
```
sns.countplot(x=target, palette='viridis')
plt.title("Target Variable Distribution")

plt.show()
```

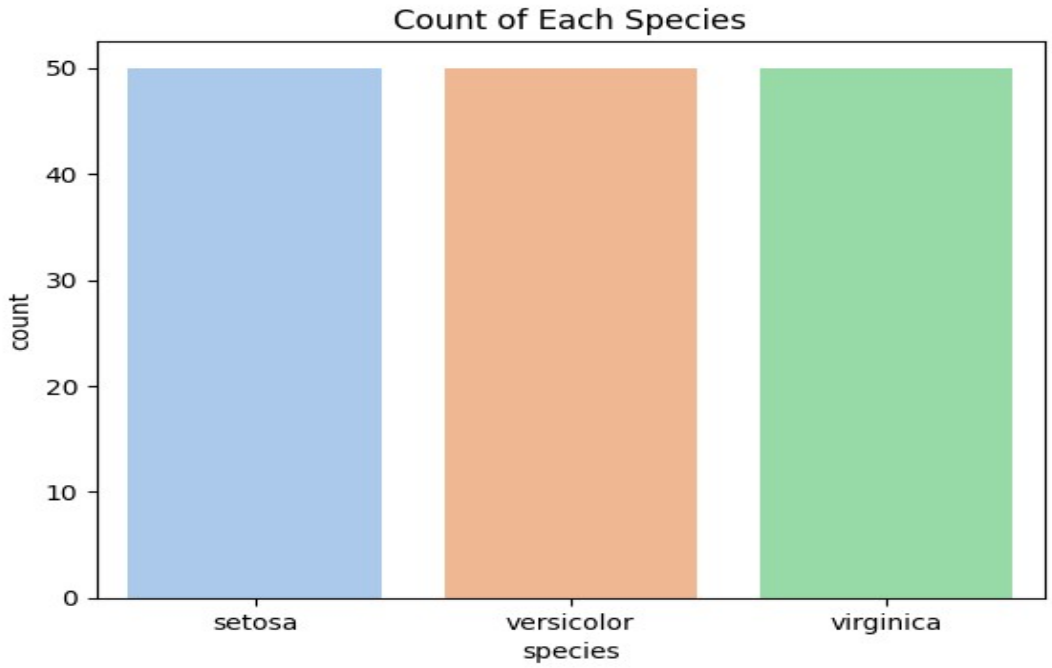
7. Save the Cleaned and Processed Dataset

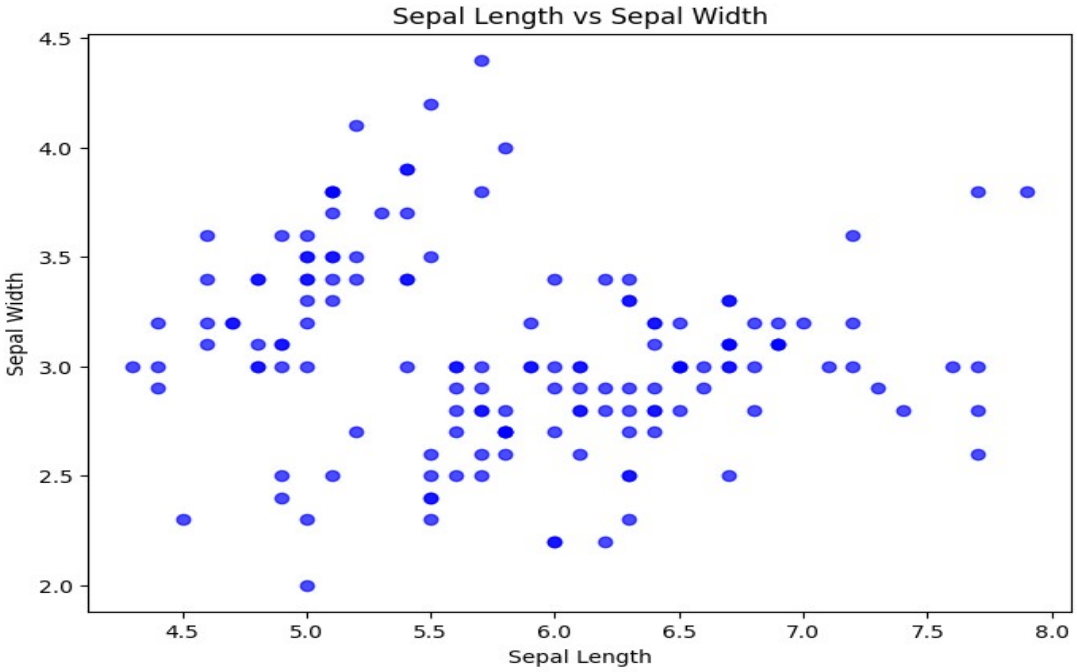
 # Save the dataset

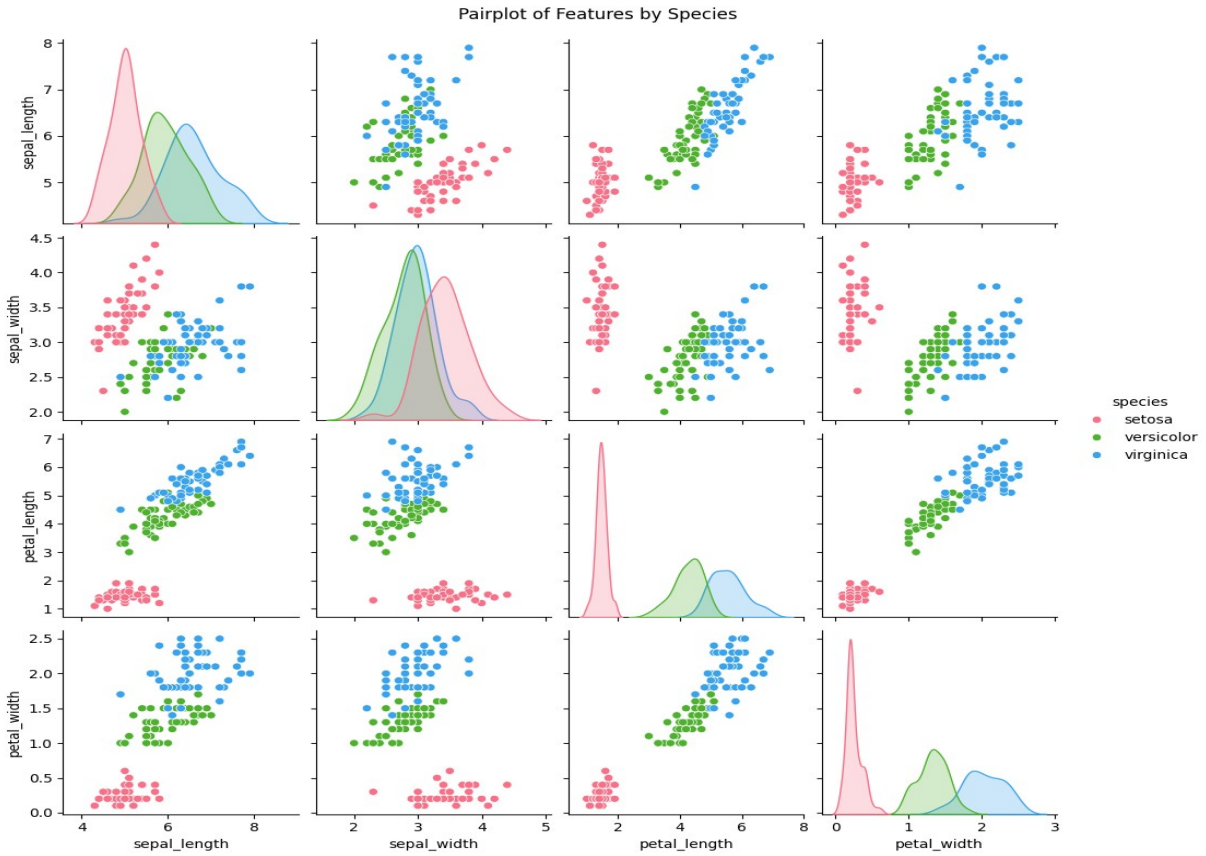
```
data.to_csv('processed_iris.csv', index=False)

print("\nProcessed dataset saved as 'processed_iris.csv'")
```

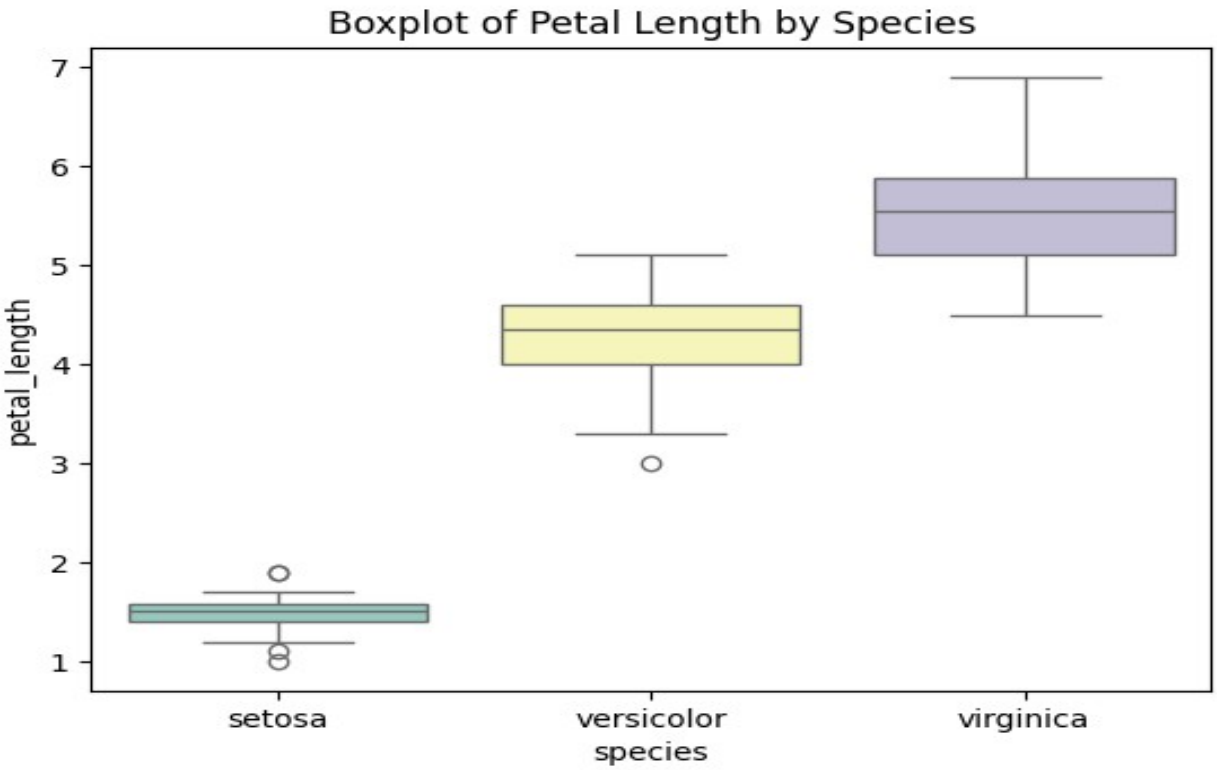
Output :




Count of Each Species
count
50
40
30
20
10
0
setosa
versicolor
virginica
species


Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
Sepal Length


Pairplot of Features by Species
sepal_length
sepal_width
petal_length
petal_width
species
setosa
versicolor
virginica


Boxplot of Petal Length by Species
petal_length
setosa
versicolor
virginica
species

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
```

2. 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)
```

3. 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)
```

4. 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 :

```
Sample Dataset:
  Category  Age  Income Has_Car
0        A   23   50000     Yes
1        B   45   60000      No
2        C   31   70000     Yes
3        A   22   80000      No
4        B   35   90000     Yes
5        C   30  100000      No
```

```
After Label Encoding:
  Category  Age  Income Has_Car  Category_Encoded  Has_Car_Encoded
0        A   23   50000     Yes                 0                1
1        B   45   60000      No                 1                0
2        C   31   70000     Yes                 2                1
3        A   22   80000      No                 0                0
4        B   35   90000     Yes                 1                1
5        C   30  100000      No                 2                0
```

```
After Scaling:
  Category  Age  Income Has_Car  Category_Encoded  Has_Car_Encoded  \
0        A   23   50000     Yes                 0                1
1        B   45   60000      No                 1                0
2        C   31   70000     Yes                 2                1
3        A   22   80000      No                 0                0
4        B   35   90000     Yes                 1                1
5        C   30  100000      No                 2                0

   Income_MinMax  Age_Standardized
0            0.0         -1.035676
1            0.2          1.812434
2            0.4          0.000000
3            0.6         -1.165136
4            0.8          0.517838
5            1.0         -0.129460
```

```
After Binarization:
  Category  Age  Income Has_Car  Category_Encoded  Has_Car_Encoded  \
0        A   23   50000     Yes                 0                1
1        B   45   60000      No                 1                0
2        C   31   70000     Yes                 2                1
3        A   22   80000      No                 0                0
4        B   35   90000     Yes                 1                1
5        C   30  100000      No                 2                0

   Income_MinMax  Age_Standardized  Income_Binary
0            0.0         -1.035676              0
1            0.2          1.812434              0
2            0.4          0.000000              0
3            0.6         -1.165136              1
4            0.8          0.517838              1
5            1.0         -0.129460              1
```

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 :

```
Dataset:
      Sky Temperature Humidity    Wind Water Forecast Condition
0  Sunny        Warm   Normal  Strong  Warm     Same       Yes
1  Sunny        Cold     High  Strong  Warm     Same        No
2  Rainy        Warm     High    Weak  Cool   Change        No
3  Sunny        Warm   Normal  Strong  Warm     Same       Yes
4  Rainy        Cold   Normal    Weak  Cool   Change        No
```

```
Loaded Dataset:
      Sky Temperature Humidity    Wind Water Forecast Condition
0  Sunny        Warm   Normal  Strong  Warm     Same       Yes
1  Sunny        Cold     High  Strong  Warm     Same        No
2  Rainy        Warm     High    Weak  Cool   Change        No
3  Sunny        Warm   Normal  Strong  Warm     Same       Yes
4  Rainy        Cold   Normal    Weak  Cool   Change        No
```

```
Final Specific Hypothesis:
['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same']
```

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")

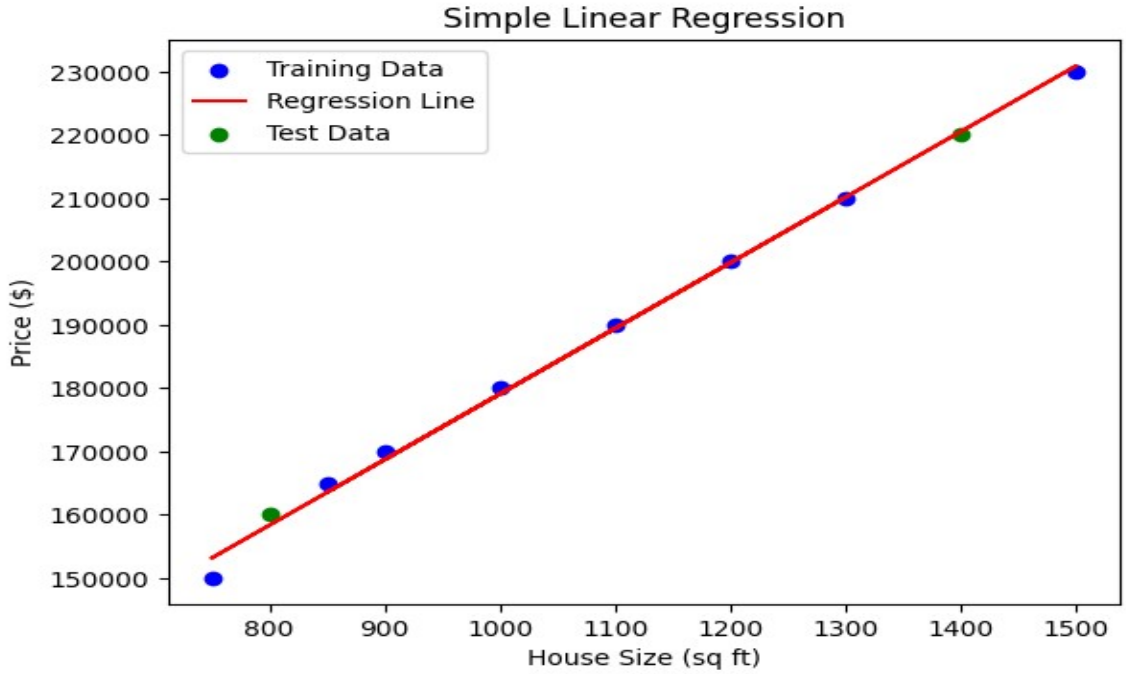
plt.xlabel("House Size (sq ft)")

plt.ylabel("Price ($)")

plt.legend()

plt.show()

Output :



3b. Multiple Linear Regression :

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

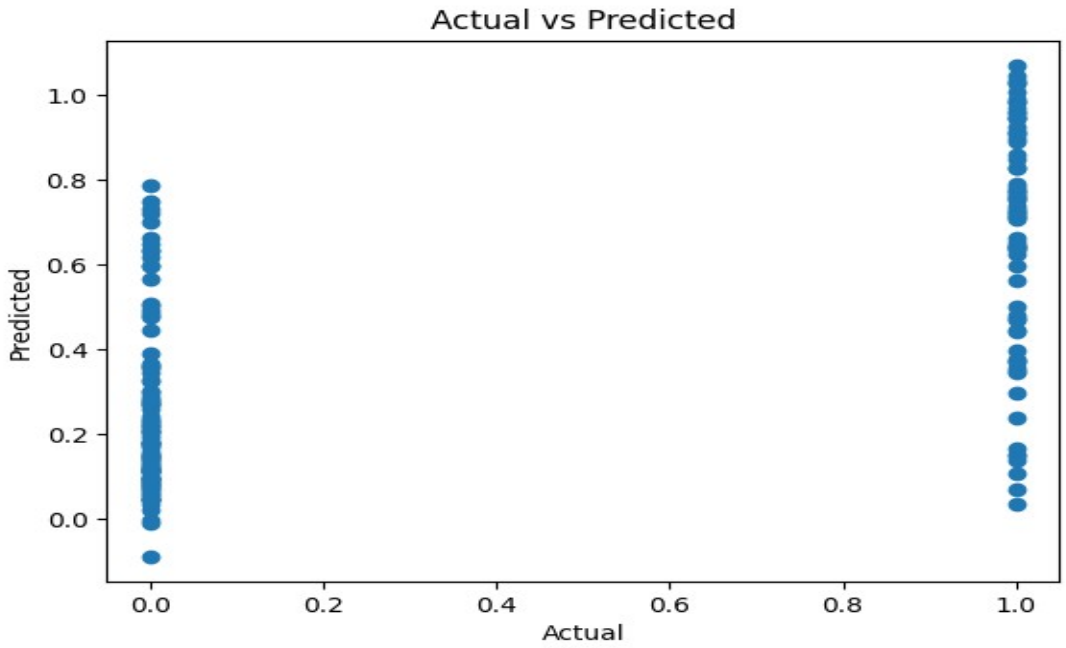
Code :

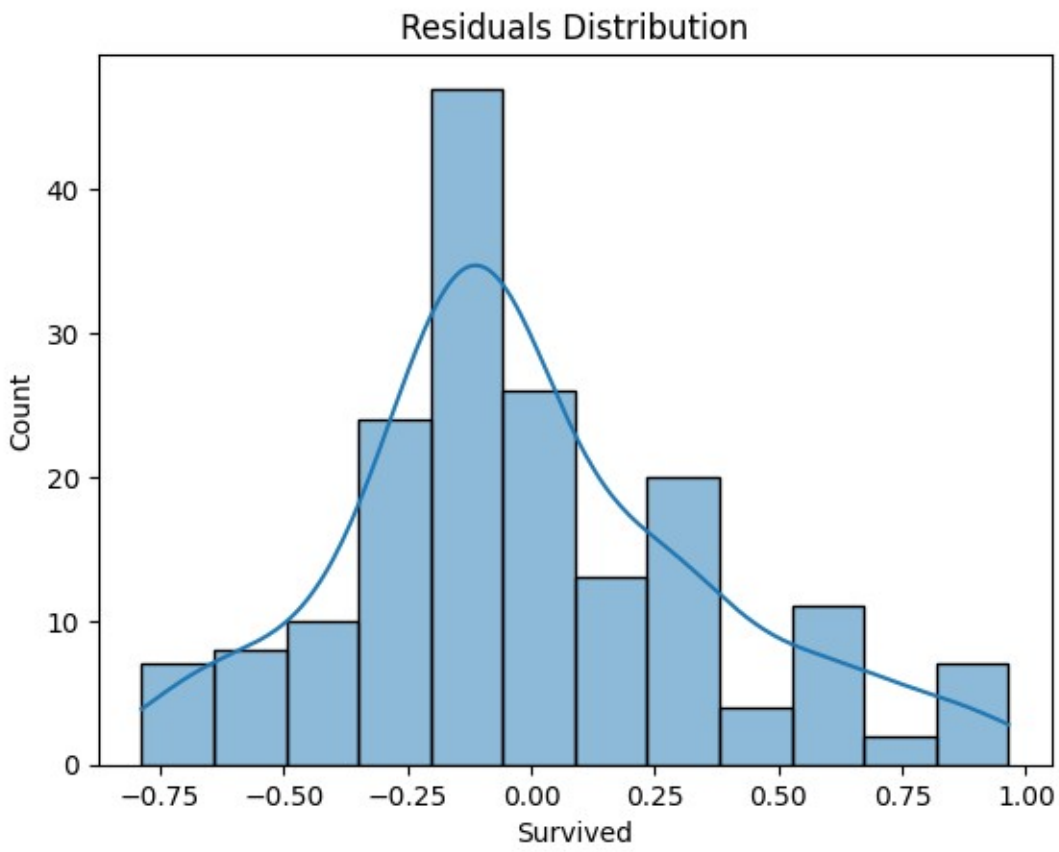
```
import pandas as pd

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 :




Residuals Distribution
Count
Survived
0
10
20
30
40
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

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}")
```

5. **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_)

6. 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}")

7. Compare Results

 # Collect metrics

metrics = pd.DataFrame({

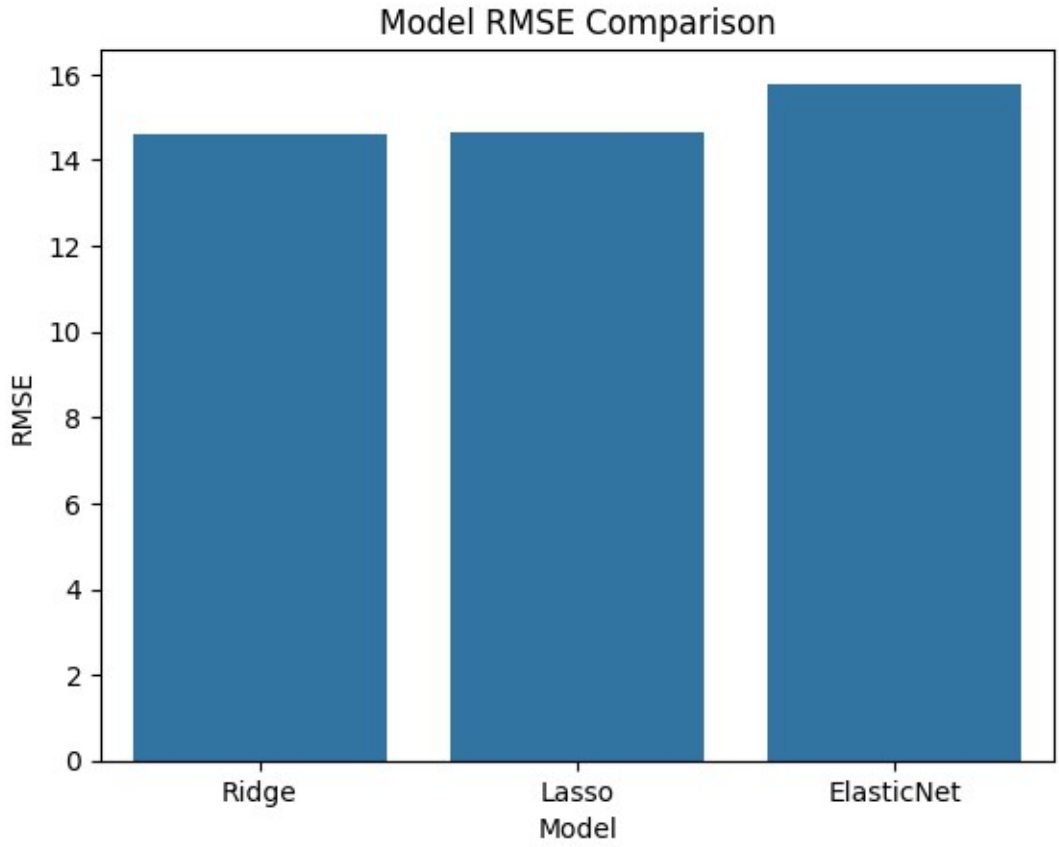
"Model": ["Ridge", "Lasso", "ElasticNet"],

"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

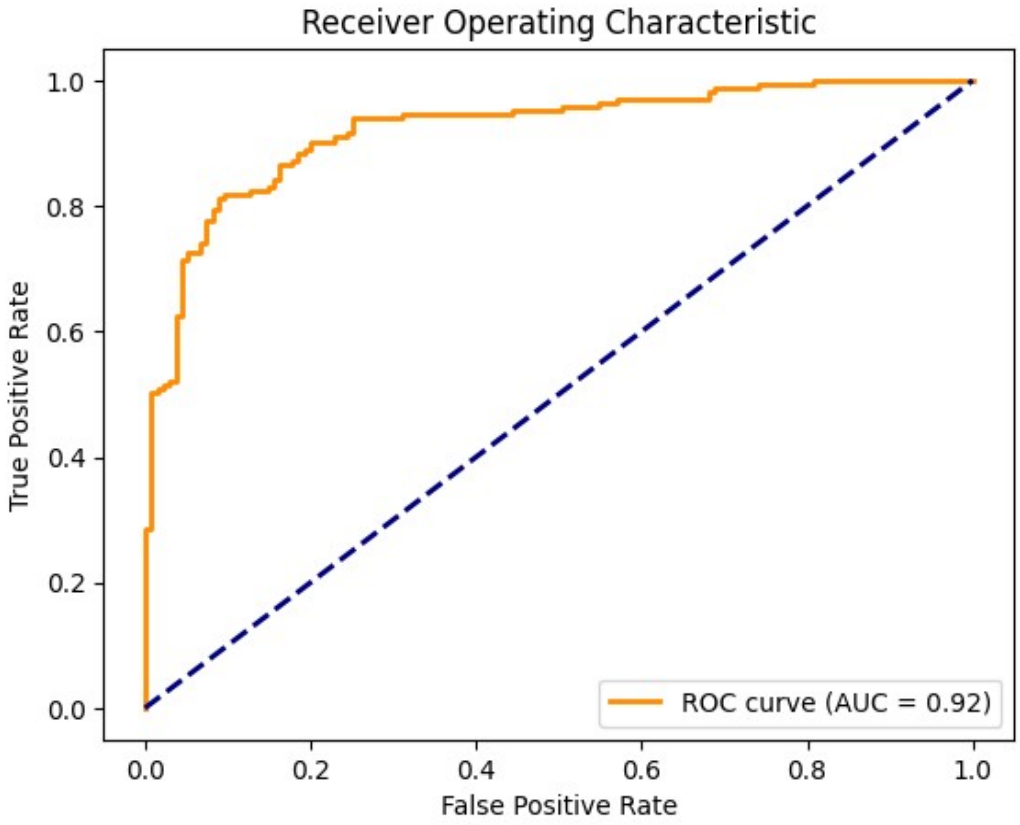
accuracy = accuracy_score(y_test, y_pred)

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 :

```
[ ]
    Model Accuracy: 0.88

    Correct Predictions:
    Sample 0: Predicted=0, Actual=0
    Sample 1: Predicted=1, Actual=1
    Sample 2: Predicted=1, Actual=1
    Sample 3: Predicted=0, Actual=0
    Sample 4: Predicted=1, Actual=1
    Sample 5: Predicted=1, Actual=1
    Sample 6: Predicted=0, Actual=0
    Sample 7: Predicted=0, Actual=0
    Sample 9: Predicted=1, Actual=1
    Sample 10: Predicted=1, Actual=1
    Sample 11: Predicted=1, Actual=1
    Sample 12: Predicted=0, Actual=0
    Sample 13: Predicted=0, Actual=0
    Sample 14: Predicted=0, Actual=0
    Sample 15: Predicted=0, Actual=0
    Sample 16: Predicted=0, Actual=0
    Sample 17: Predicted=1, Actual=1
    Sample 18: Predicted=1, Actual=1
    Sample 19: Predicted=0, Actual=0
    Sample 20: Predicted=0, Actual=0
    Sample 22: Predicted=1, Actual=1
    Sample 23: Predicted=1, Actual=1
    Sample 24: Predicted=1, Actual=1
    Sample 25: Predicted=1, Actual=1
    Sample 26: Predicted=1, Actual=1
    Sample 27: Predicted=0, Actual=0
    Sample 28: Predicted=0, Actual=0
    Sample 30: Predicted=1, Actual=1
    Sample 31: Predicted=1, Actual=1
    Sample 32: Predicted=1, Actual=1
    Sample 34: Predicted=0, Actual=0
    Sample 35: Predicted=1, Actual=1
    Sample 36: Predicted=1, Actual=1
    Sample 38: Predicted=1, Actual=1
    Sample 39: Predicted=1, Actual=1
```

```
    Incorrect Predictions:
    Sample 8: Predicted=1, Actual=0
    Sample 21: Predicted=1, Actual=0
    Sample 29: Predicted=0, Actual=1
    Sample 33: Predicted=1, Actual=0
    Sample 37: Predicted=1, Actual=0
```

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

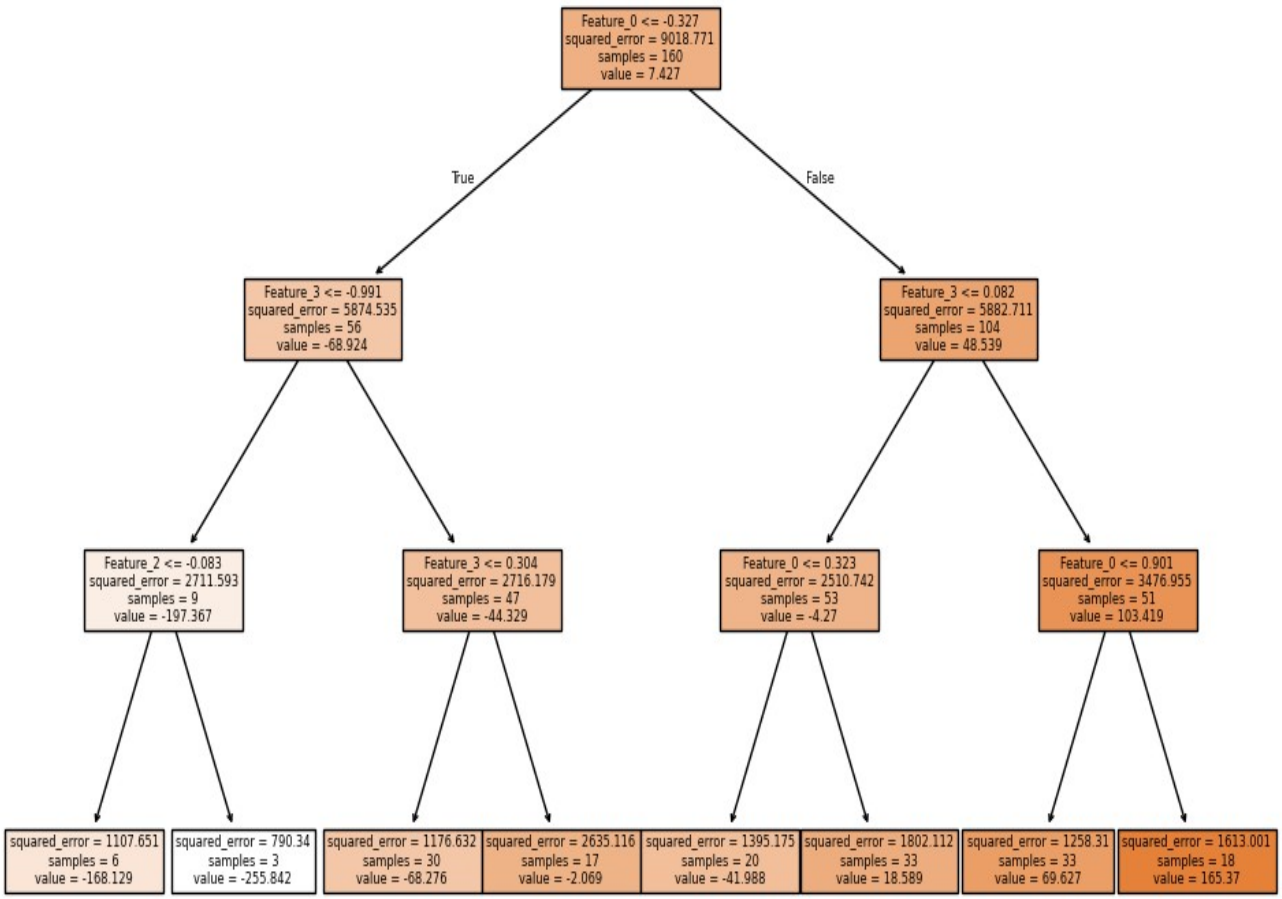
```
if task_type == "classification":

   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 :


Decision Tree Visualization
Feature_0 <= -0.327
squared_error = 9018.771
samples = 160
value = 7.427
True
False
Feature_3 <= -0.991
squared_error = 5874.535
samples = 56
value = -68.924
Feature_3 <= 0.082
squared_error = 5882.711
samples = 104
value = 48.539
Feature_2 <= -0.083
squared_error = 2711.593
samples = 9
value = -197.367
Feature_3 <= 0.304
squared_error = 2716.179
samples = 47
value = -44.329
Feature_0 <= 0.323
squared_error = 2510.742
samples = 53
value = -4.27
Feature_0 <= 0.901
squared_error = 3476.955
samples = 51
value = 103.419
squared_error = 1107.651
samples = 6
value = -168.129
squared_error = 790.34
samples = 3
value = -255.842
squared_error = 1176.632
samples = 30
value = -68.276
squared_error = 2635.116
samples = 17
value = -2.069
squared_error = 1395.175
samples = 20
value = -41.988
squared_error = 1802.112
samples = 33
value = 18.589
squared_error = 1258.31
samples = 33
value = 69.627
squared_error = 1613.001
samples = 18
value = 165.37

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')

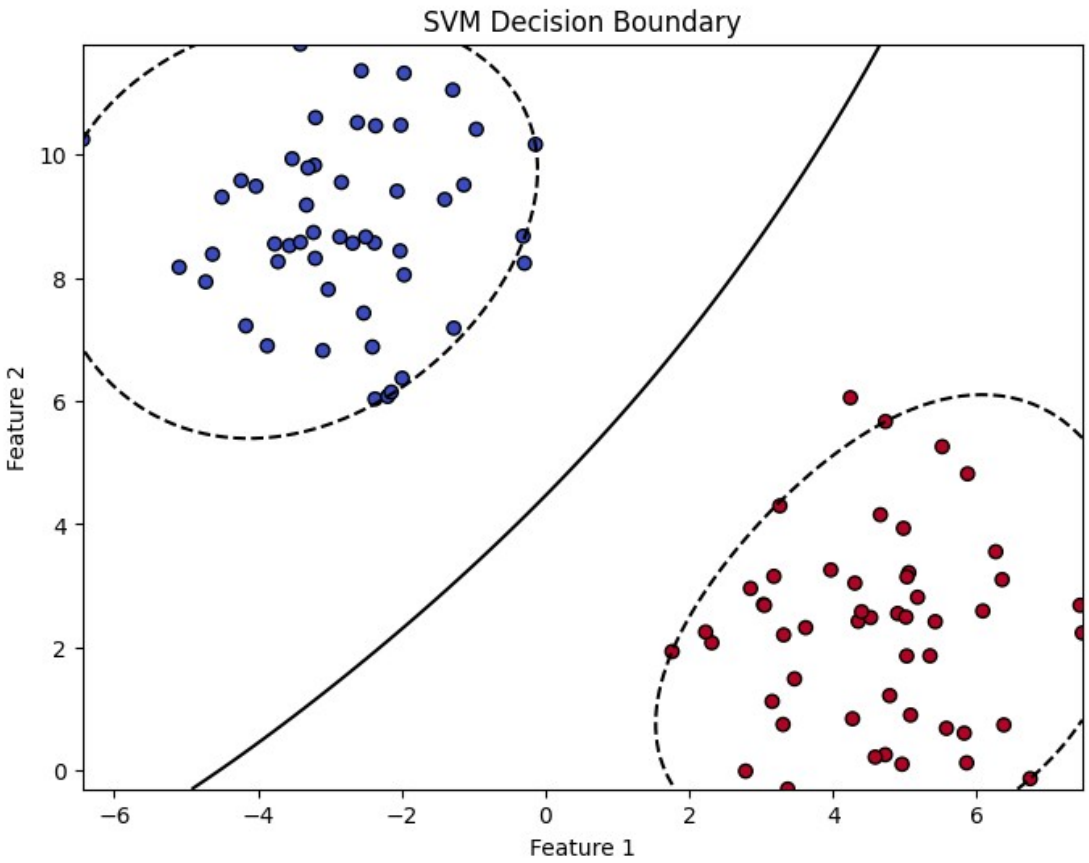
plt.title("SVM Decision Boundary")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

Output :


SVM Decision Boundary
Feature 2
Feature 1
10
8
6
4
2
0
−6
−4
−2
0
2
4
6

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))

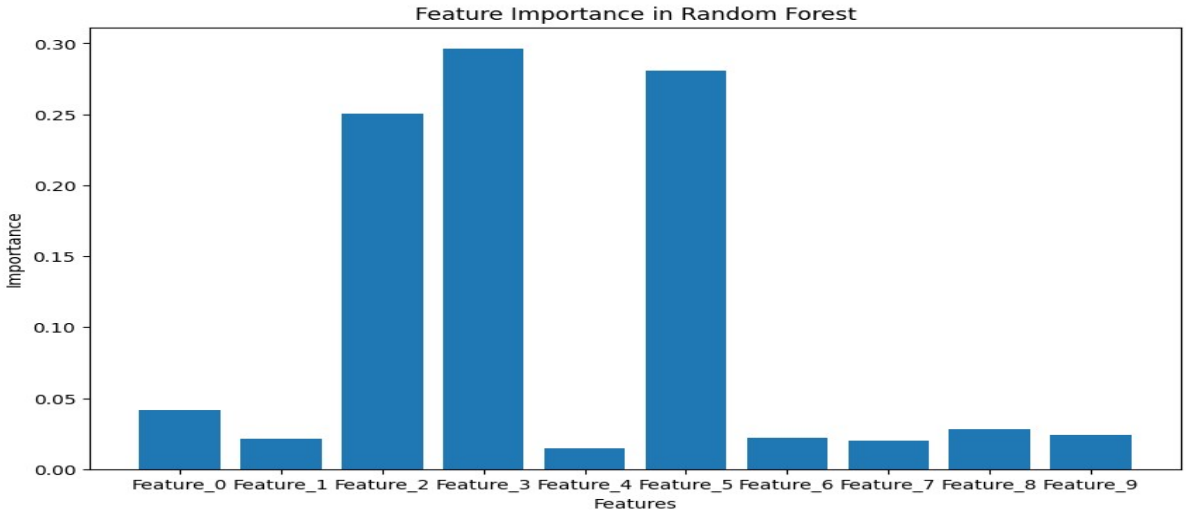
plt.bar(range(len(feature_importances)), feature_importances, tick_label=data.columns[:-1])
plt.title("Feature Importance in Random Forest")

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")

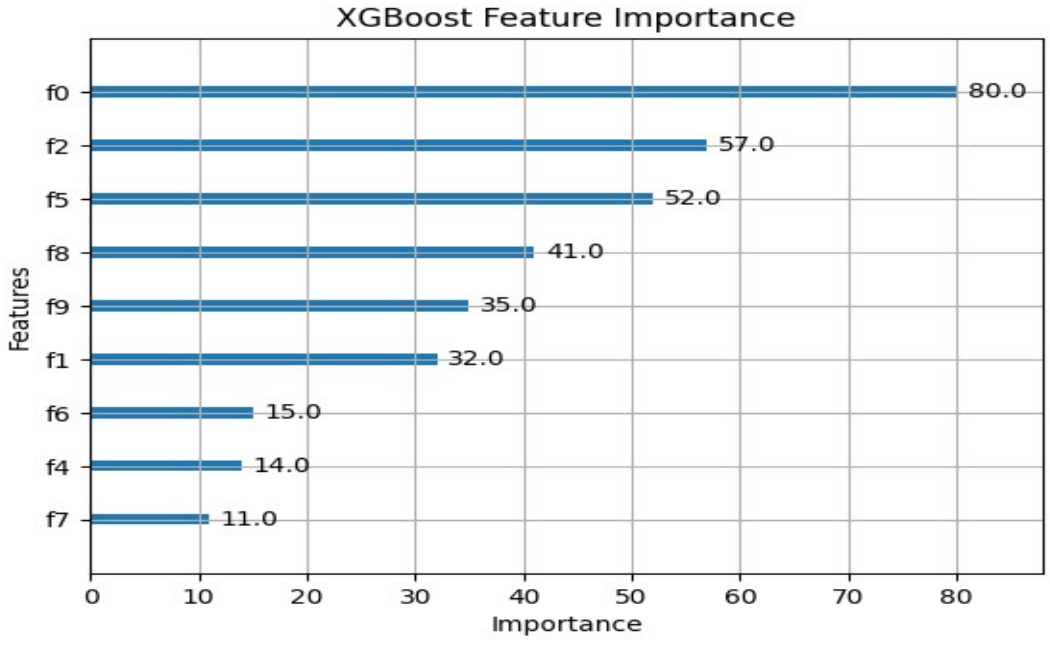
plt.title("XGBoost Feature Importance")

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 :

```
Dataset Preview:
   Feature_0  Feature_1  Feature_2  Feature_3  Feature_4  Feature_5  \
0  -1.274158   1.317988  -2.423879   0.906946  -1.583903  -0.331811
1   1.607963  -1.649959   0.299293  -0.891720   1.301741   1.508502
2  -0.154167   0.161033   2.210523   0.139400  -0.557492   0.087713
3  -0.920991   0.949136  -1.613561   0.588410   1.471170  -0.529287
4   1.013304  -1.038578  -0.305225  -0.539334  -0.609512   1.048078

   Feature_6  Feature_7  Target
0  -0.452306   0.760415       1
1   0.742095   1.561511       0
2   0.963879  -1.369803       0
3  -1.371901  -0.209324       0
4  -1.065114  -0.186971       0
```

```
Test Sample: [array([-0.90320608,  0.9220511 , -1.32308979,  0.41081065,  1.64201516,
       -1.23559176, -0.63896175,  1.00981709])]
Predicted Class: 1
```

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()

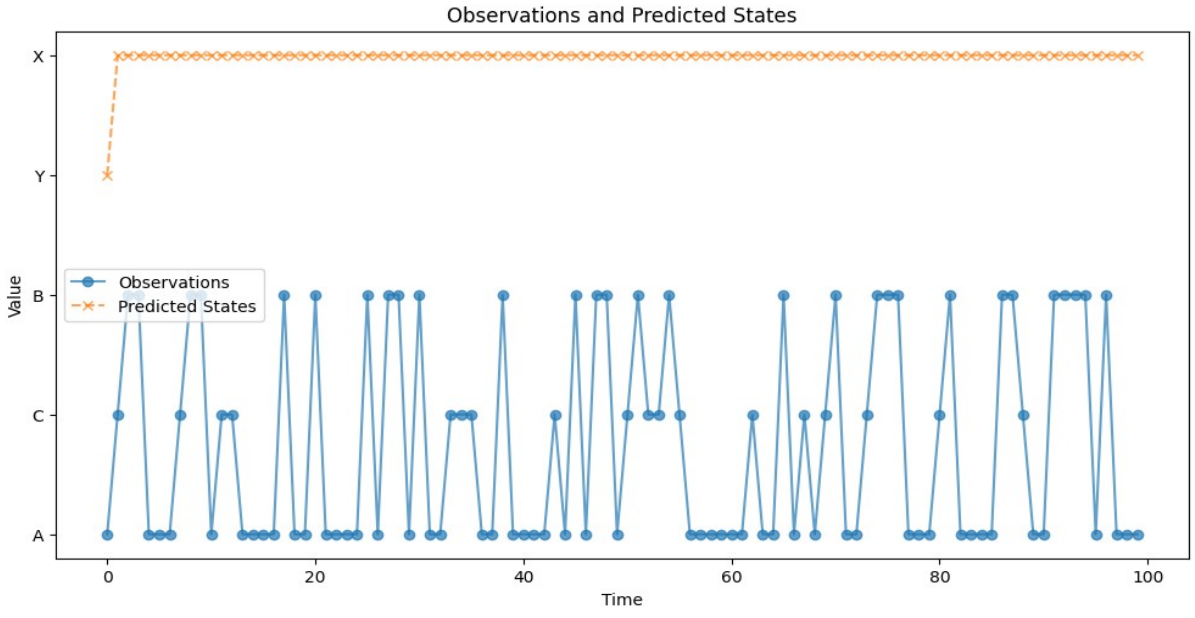
plt.title("Observations and Predicted States")

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")

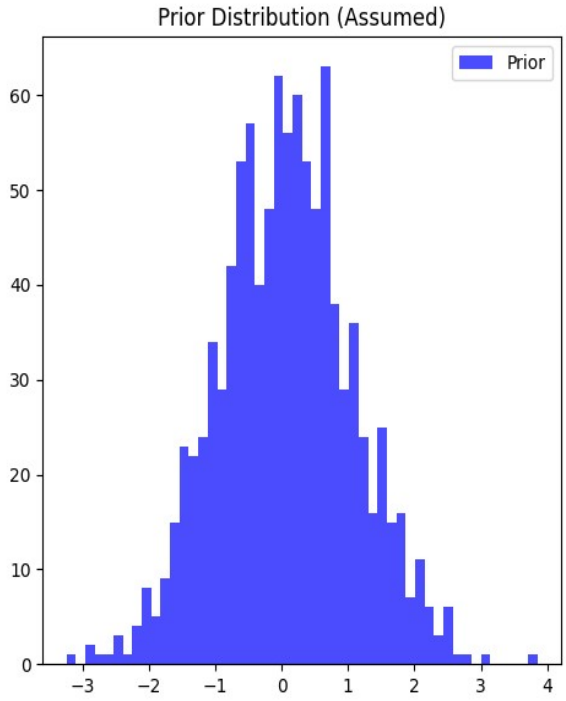
plt.ylabel("y")

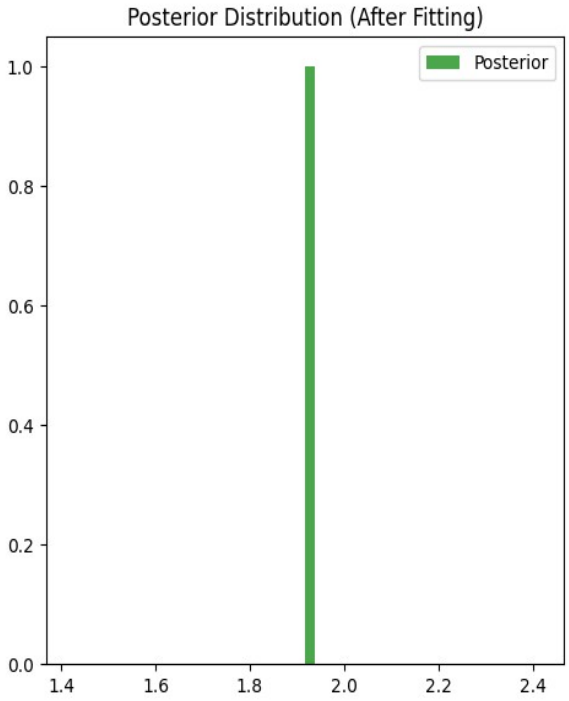
plt.legend()

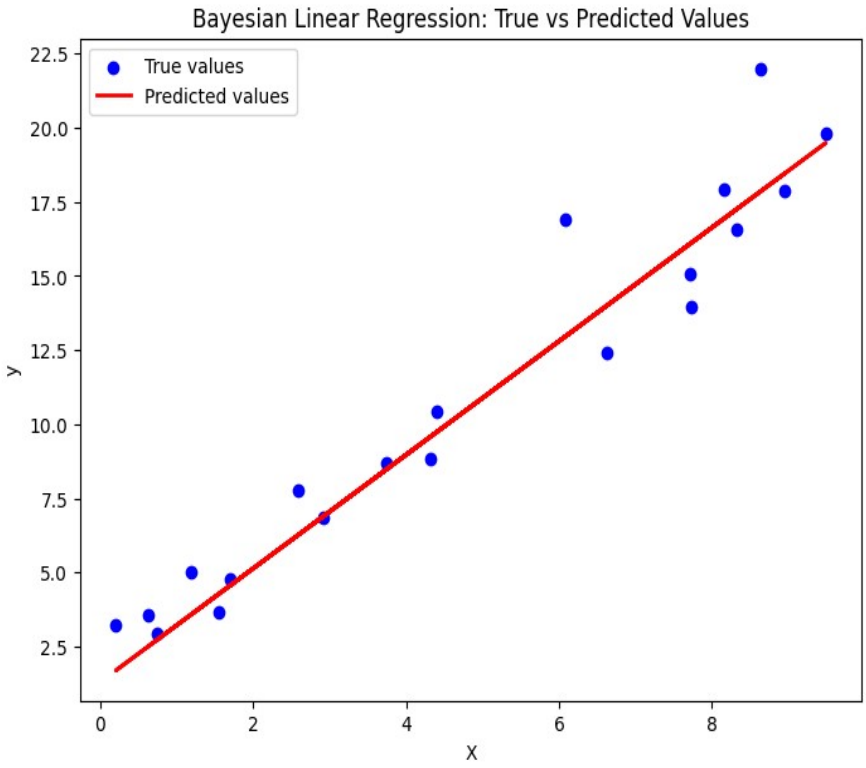
plt.show()

Mean Squared Error (MSE): 3.9

Output :


Prior Distribution (Assumed)
Prior
60
50
40
30
20
10
0
−3
−2
−1
0
1
2
3
4


Posterior Distribution (After Fitting)
Posterior
1.0
0.8
0.6
0.4
0.2
0.0
1.4
1.6
1.8
2.0
2.2
2.4


Bayesian Linear Regression: True vs Predicted Values
True values
Predicted values
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
y
0
2
4
6
8
X

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()
```

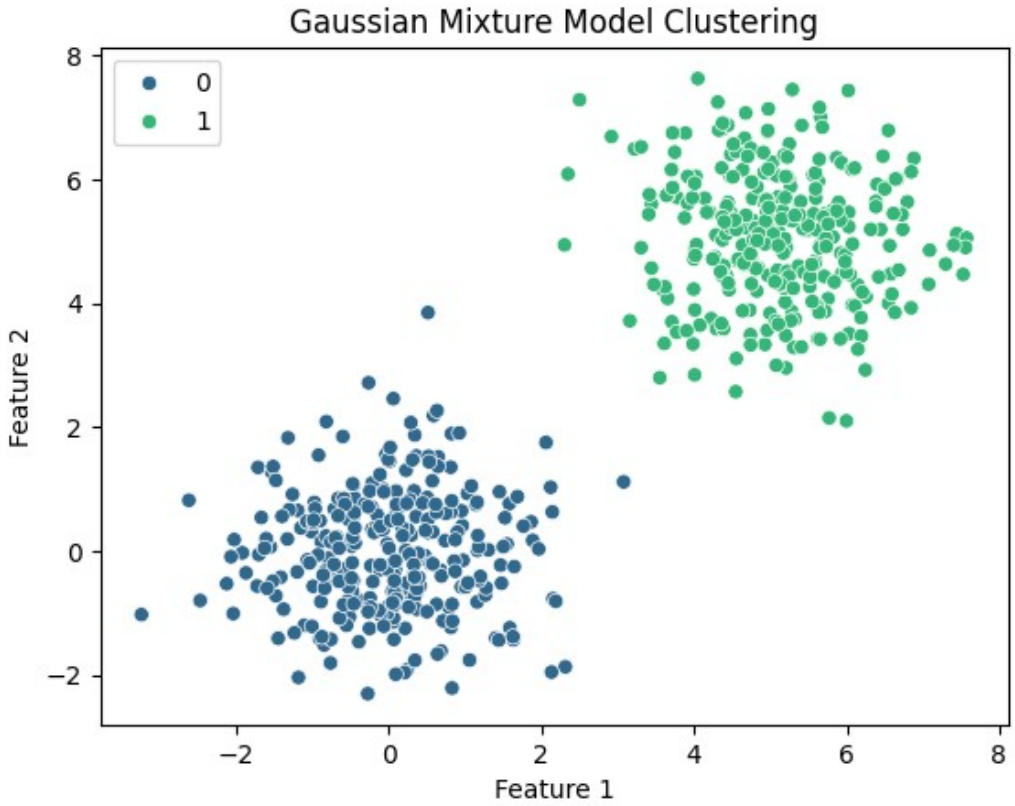
Step 7: Model Evaluation (Optional)

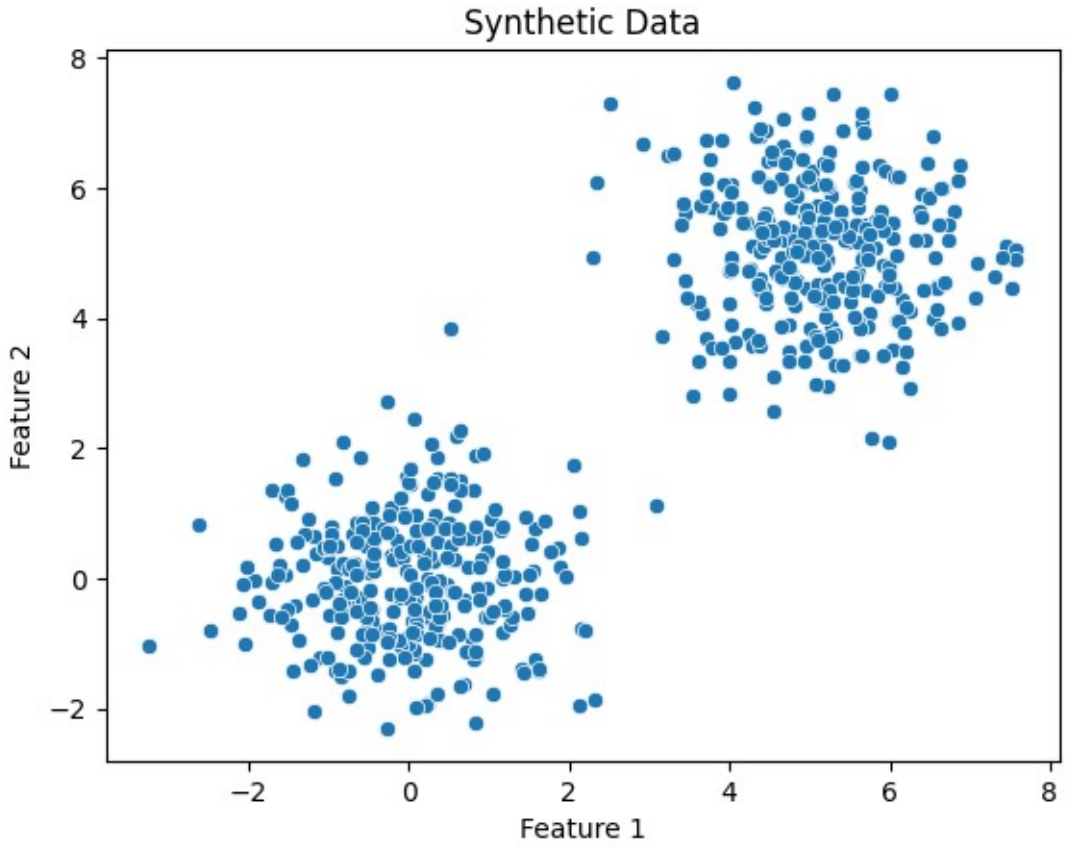
```
# Compute the log-likelihood of the data under the fitted GMM model
log_likelihood = gmm.score(data)
print(f"Log-Likelihood of the data: {log_likelihood:.2f}")
```

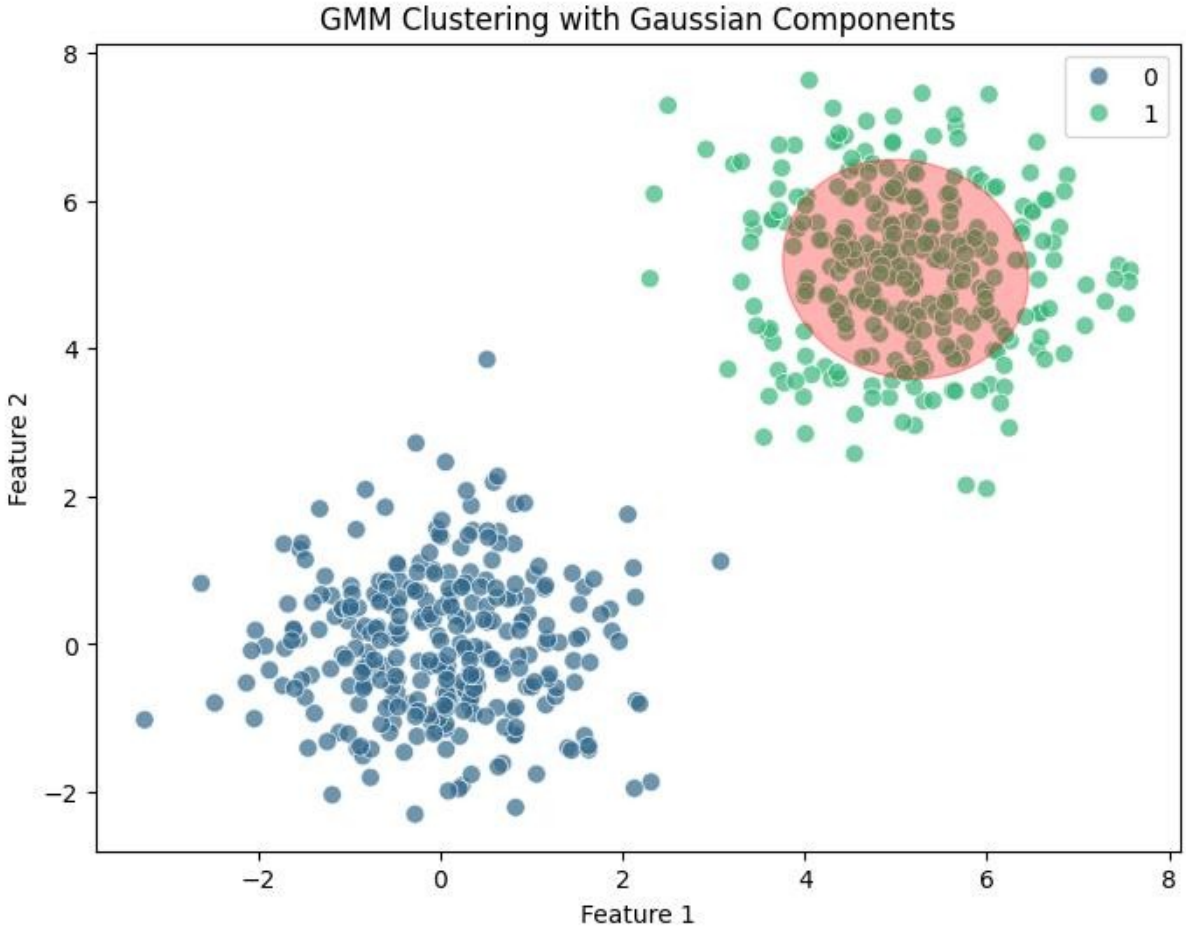
Step 8: Predict New Data Points

```
# 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 :




Synthetic Data
Feature 2
Feature 1
8
6
4
2
0
−2
−2
0
2
4
6
8


GMM Clustering with Gaussian Components
Feature 2
Feature 1
0
1
8
6
4
2
0
−2
−2
0
2
4
6
8

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 data
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')

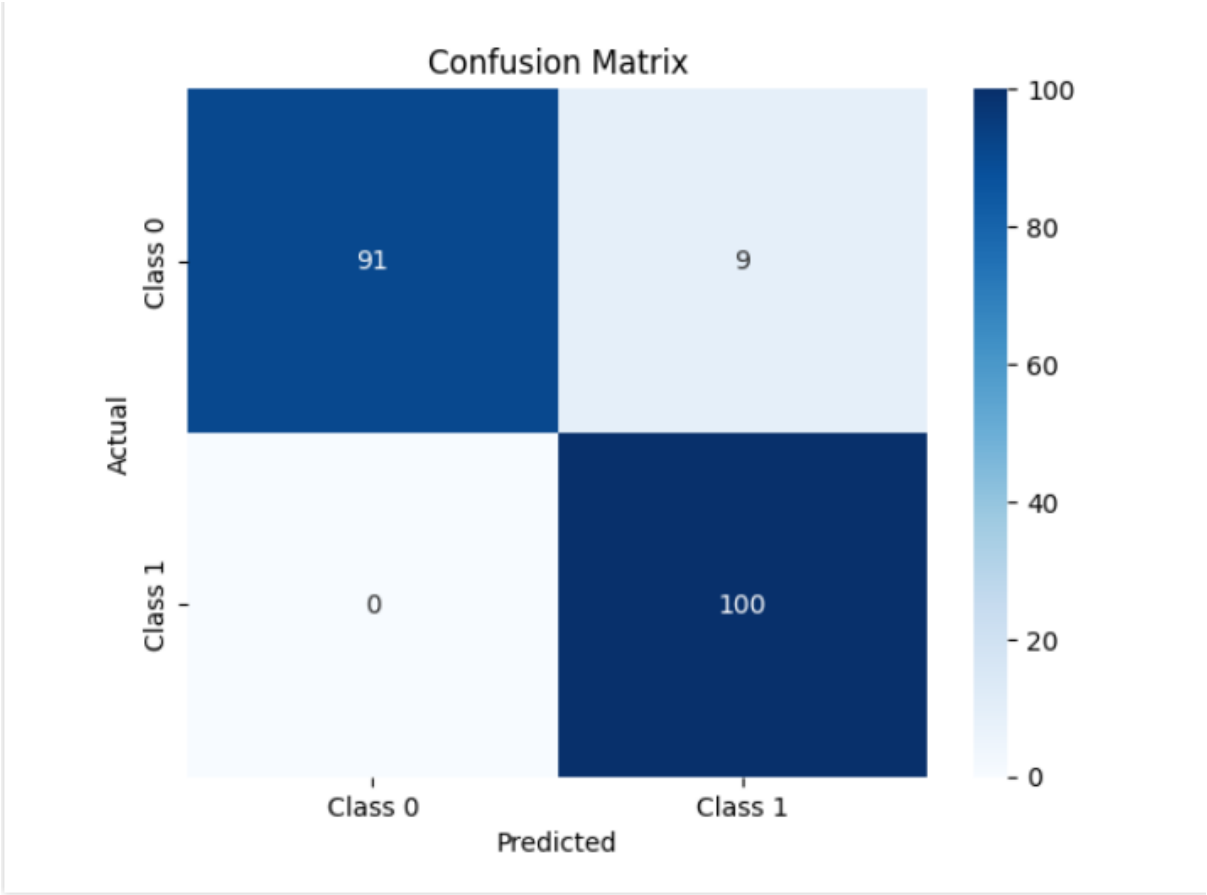
plt.show()

Output :

```
   Feature_1  Feature_2  Feature_3  Feature_4  Feature_5  Feature_6  \
0  -3.358483   3.159918   0.827163   0.069638  -6.715639  -2.708559
1   2.071819  -4.055419  -2.615940  -2.599432   3.053752   0.366795
2  -0.633460   0.712482   2.024390  -0.432639  -1.307929   0.419320
3  -0.464478   0.892442   2.521010   2.766580  -1.933734  -1.418018
4   1.042426  -1.192605  -2.071386  -0.131231   0.545377   0.379060

   Feature_7  Feature_8  Feature_9  Feature_10  Target
0   0.183206   1.113502   1.730759    1.228394       1
1  -0.392171  -1.191720  -1.220516    1.899925       0
2  -1.469510  -0.719051   1.155005    2.018026       0
3   1.391760  -2.430279   1.308295   -0.270896       1
4  -0.062978  -1.325591   2.037936    0.115414       0
k-Fold Cross-Validation:
TRAIN: [  0   1   3   4   5   6   8   9  11  12  13  14  15  16  17  18  19  20
  21  22  24  25  26  27  28  32  34  35  36  37  38  40  41  42  43  44
  45  46  47  48  50  51  52  53  55  56  57  58  59  60  61  62  64  68
  69  70  71  73  74  75  79  80  82  83  85  87  88  89  90  91  92  93
  94  95  98  99 100 102 103 104 105 106 107 108 111 112 113 114 115 116
 117 119 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136
 138 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 156 157
 158 159 160 161 162 163 164 165 166 167 169 170 171 172 173 175 176 177
 178 179 180 181 182 183 184 185 186 187 188 189 190 191 193 194 195 196
 197 200 201 202 203 205 206 207 212 213 214 216 217 219 220 221 222 223
 224 225 226 227 228 229 230 232 233 234 236 237 238 239 240 241 242 243
 245 246 247 248 249 251 252 253 255 256 257 258 259 261 262 263 264 267
 268 269 270 271 272 273 274 276 277 278 279 280 282 283 284 285 287 288
 289 290 291 292 293 295 297 298 299 300 301 303 304 305 307 308 309 310
 311 312 313 315 317 318 319 320 321 322 324 325 328 329 330 331 332 334
```

```
674 675 676 677 678 679 680 681 682 683 684 685 686 687 689 690 691 693
 694 695 697 698 699 700 701 702 703 704 706 707 708 709 710 711 712 713
 714 716 717 718 722 724 725 726 727 728 729 730 731 732 733 734 735 736
 737 739 740 742 743 745 747 748 749 751 752 753 755 756 757 759 761 762
 763 765 766 767 768 769 770 771 772 773 774 775 778 779 780 781 782 784
 785 788 789 790 791 792 793 794 797 799] VALIDATION: [  2   7  10  23  29  30  31  33  39  49  54  63  65  66  67  72  76  77
  78  81  84  86  96  97 101 109 110 118 120 137 139 155 168 174 192 198
 199 204 208 209 210 211 215 218 231 235 244 250 254 260 265 266 275 281
 286 294 296 302 306 314 316 323 326 327 333 336 346 352 357 360 361 365
 367 368 377 383 388 393 395 398 409 422 423 425 428 432 446 456 464 481
 483 486 490 506 512 513 515 519 521 525 526 529 532 533 534 537 545 568
 570 578 589 591 594 595 596 604 608 610 621 622 628 635 637 640 641 644
 652 655 656 658 662 666 667 670 688 692 696 705 715 719 720 721 723 738
 741 744 746 750 754 758 760 764 776 777 783 786 787 795 796 798]
TRAIN: [  0   1   2   3   4   5   7   8   9  10  12  13  14  15  16  17  19  20
  21  22  23  25  26  27  29  30  31  32  33  34  35  36  37  38  39  40
  45  46  47  48  49  50  52  53  54  57  58  59  62  63  64  65  66  67
  68  71  72  75  76  77  78  80  81  84  85  86  87  88  91  93  94  95
  96  97  98  99 100 101 102 103 105 106 107 109 110 111 112 113 115 116
 117 118 119 120 121 122 123 124 125 126 127 128 129 130 134 137 138 139
 141 142 143 144 146 147 149 150 151 152 153 154 155 156 157 159 160 161
 162 166 168 169 170 171 172 173 174 175 179 180 183 184 185 186 187 188
 189 190 191 192 194 195 197 198 199 200 201 202 203 204 205 206 207 208
 209 210 211 214 215 216 217 218 219 221 222 224 225 226 228 229 230 231
 232 233 235 236 237 238 240 241 242 243 244 245 246 249 250 251 252 253
 254 255 256 257 258 260 261 262 263 265 266 267 268 269 270 271 272 273
 274 275 276 277 278 279 280 281 282 283 284 286 287 288 289 293 294 295
 296 297 298 301 302 303 304 305 306 307 308 310 312 313 314 315 316 317
 318 320 321 322 323 324 325 326 327 330 333 335 336 337 338 339 340 341
```

```
 450 455 459 460 463 470 472 475 480 481 483 494 496 502 503 505 514 519
 521 523 529 533 553 555 560 563 565 579 585 590 594 600 603 620 630 631
 634 635 637 638 644 646 648 649 651 673 675 686 691 693 708 709 715 720
 729 730 738 747 753 759 767 771 774 776 777 780 782 783 784 796]
Average Accuracy from k-Fold: 0.95
Fitting 5 folds for each of 36 candidates, totalling 180 fits
Best Parameters: {'max_depth': None, 'min_samples_split': 5, 'n_estimators': 50}
Best Cross-Validation Accuracy: 0.96

Test Accuracy: 0.955

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.91      0.95       100
           1       0.92      1.00      0.96       100

    accuracy                           0.95       200
   macro avg       0.96      0.96      0.95       200
weighted avg       0.96      0.95      0.95       200
```


Confusion Matrix
Class 0
Class 1
Actual
91
9
0
100
Class 0
Class 1
Predicted
100
80
60
40
20
0

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')

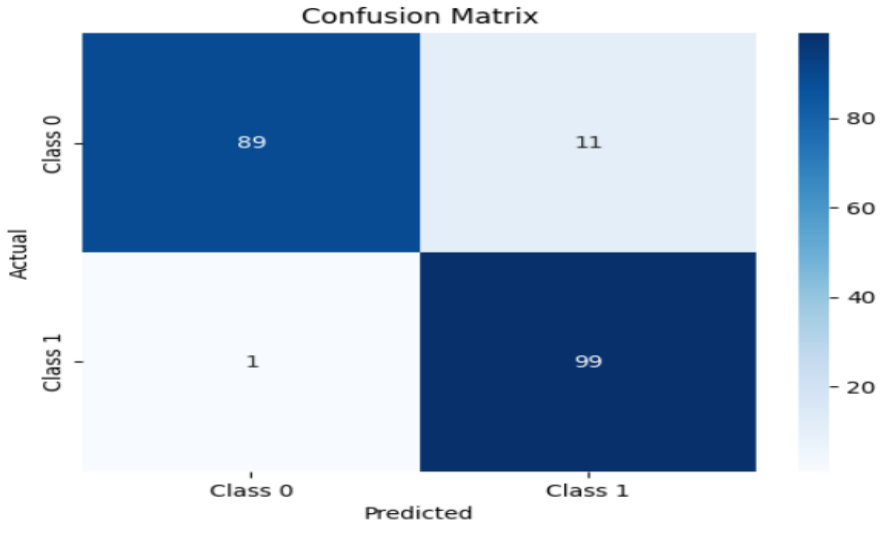
plt.title('Confusion Matrix')

plt.show()

Output :

```
   Feature_1  Feature_2  Feature_3  Feature_4  Feature_5  Feature_6  \
0   0.013650   0.607473  -2.096916   2.867232   2.504360   0.784101
1   0.107199   0.105735  -3.843343   1.524052  -1.619824   0.778334
2  -1.779086  -5.219831  -0.738488   2.108084  -0.803833  -3.431122
3  -4.310656  -2.268569   1.864943  -1.246116   1.260794  -2.007664
4  -3.195179  -0.671327   3.720485   0.356661   0.819486   2.670230

   Feature_7  Feature_8  Feature_9  Feature_10  Feature_11  Feature_12  Target
0  -0.497744  -0.482072   1.112773    1.641637   -2.689832   -0.480311       1
1   0.551177  -1.843583  -0.110132   -0.494739   -0.985276   -0.978400       0
2   1.346120  -0.858351  -0.792415   -2.260815    0.238780    3.029952       0
3  -0.824133  -2.277449   0.936206    1.255903    1.386278   -0.321200       0
4   1.857477  -3.410944  -1.773719    0.656476    3.534189   -1.704889       0
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Best Parameters from Grid Search: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
Best Cross-Validation Accuracy from Grid Search: 0.9487499999999999
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best Parameters from Randomized Search: {'max_depth': None, 'min_samples_leaf': 2, 'min_samples_split': 9, 'n_estimators': 285}
Best Cross-Validation Accuracy from Randomized Search: 0.9487499999999999

Test Accuracy: 0.94
```

```
Classification Report:
                precision    recall  f1-score   support

           0       0.99      0.89      0.94       100
           1       0.90      0.99      0.94       100

    accuracy                           0.94       200
   macro avg       0.94      0.94      0.94       200
weighted avg       0.94      0.94      0.94       200
```



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:

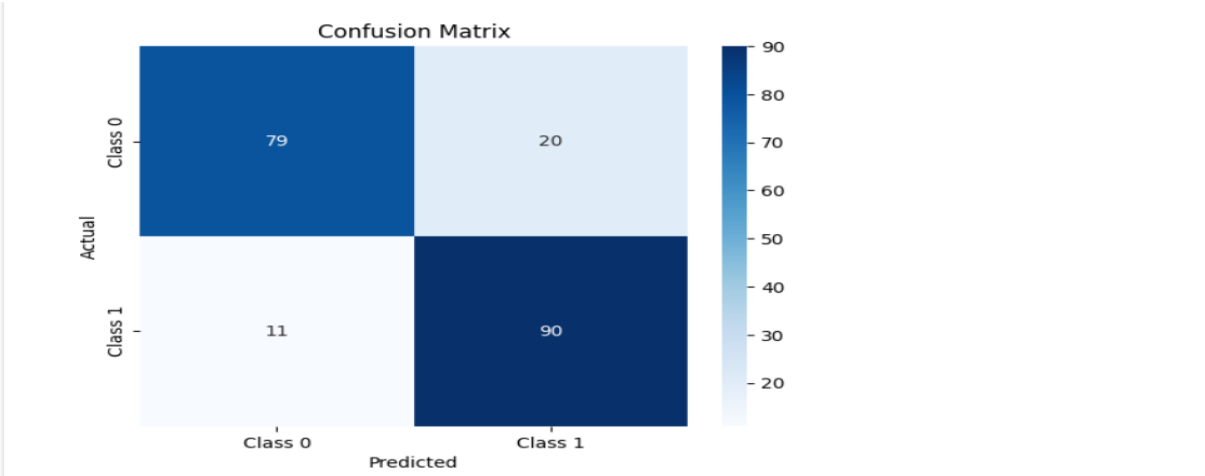
- **Prior Probability (P(C)P(C)P(C)):** The likelihood of each class based on historical data.
- **Likelihood (P(X|C)P(X|C)P(X|C)):** The probability of the data given a class.
- **Posterior Probability (P(C|X)P(C|X)P(C|X)):** Calculated using Bayes' theorem: P(C|X)=P(X|C)·P(C)P(X)P(C|X) = \frac{P(X|C) \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 :

```
   Feature_1  Feature_2  Feature_3  Feature_4  Feature_5  Feature_6  \
0  -1.732538   5.260112  -2.952194  -4.603768   2.235848   1.928893
1   2.072914   2.240572  -1.385104  -2.514962  -0.984756   1.436260
2  -0.263106   1.527781  -1.872414  -0.028009   1.612809   3.264194
3  -0.164349  -0.550131  -0.019503  -0.765000   2.273523   2.084217
4  -1.419423   1.015324  -0.864441  -0.009297   0.385404   0.449093

   Feature_7  Feature_8  Target
0  -0.101845   3.193487       0
1  -1.255271   2.089872       0
2  -1.296421   1.537870       0
3  -0.321931   0.426253       0
4  -0.029007  -1.902917       1
Test Accuracy: 0.84

Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.80      0.84        99
           1       0.82      0.89      0.85       101

    accuracy                           0.84       200
   macro avg       0.85      0.84      0.84       200
weighted avg       0.85      0.84      0.84       200
```




```
Sample Features: [[ 1.94397502  0.33753289 -0.77968371  2.63667244 -1.62323864  1.87859635
  -2.20522563  2.09632056]]
Posterior Probabilities: [[0.07067404 0.92932596]]
Predicted Class: [1]
```

NURTURING POTENTIAL

SAKET GYANPEETH'S

SAKET COLLEGE OF ARTS, SCIENCE AND COMMERCE

(Permanently Affiliated to University of Mumbai)

NAAC Accredited

Saket Vidyanagri Marg, Chinchpada Road, Katemanivali,
Kalyan (East) -421306(Mah)

Department of Information Technology

This is to certify that

Mr./Ms. YASH PRAMOD GAIKWAD Seat No. 254337

of Server Virtualization on VMWare platform

M.Sc. Information Technology

Part II NEP 2020 Semester III

has satisfactory carried out the required practical in the subject
of
For the Academic year 2025 – 2026

Practical In-Charge

Head of the Department

External Examiner

College Seal

INDEX

Sr.no	Practical	Date	Sign
1	Deploying and Configuring Virtual Machines		
2	Working with vCenter Server Appliance		
3	Users, Groups, and Permissions		
4	Using Standard Switches		
5	Accessing iSCSI Storage		
6	Managing VMFS Datastores		
7	Accessing NFS Storage		
8	Using Templates and Clones		
9	Modifying Virtual Machines		
10	Migrating Virtual Machines (vMotion)		
11	Managing Virtual Machines		
12	Managing Resource Pools		

Practical 1: Deploying and Configuring Virtual Machines

Objective:

To access the student desktop, create a virtual machine, install VMware Tools, and copy files to the desktop.

Procedure:

1. Open the **VMware HOL portal** and launch the assigned lab.
2. Log in to the **Student Desktop** using the credentials provided in the lab manual.
3. Open **vSphere Client** from the desktop.
4. Log in to the vCenter Server using administrator credentials.
5. Right-click on the **Datacenter / Host** → Select **New Virtual Machine**.
6. Choose **Create a new virtual machine** and click **Next**.
7. Enter the **Virtual Machine name** and select the **Datacenter**.
8. Select the **Compute Resource (ESXi Host)**.
9. Choose the **Datastore** for VM storage.
10. Select the **Guest OS type and version**.
11. Customize **CPU, Memory, and Disk** settings.
12. Finish the wizard and power on the VM.
13. Right-click the VM → **Install VMware Tools**.
14. Complete the VMware Tools installation inside the guest OS.
15. Copy sample files from shared storage to the VM desktop.

Result:

A virtual machine was successfully created, configured, VMware Tools installed, and files copied to the desktop.

Practical 2: Working with vCenter Server Appliance

Objective:

To configure vCenter Server Appliance, datacenter, hosts, folders, and navigation.

Procedure:

1. Log in to the **vSphere Client**.
2. Navigate to **Administration → Licensing** and assign available licenses.
3. Go to **Administration → Single Sign-On → Configuration**.
4. Create a **Datacenter Object** under vCenter.
5. Right-click Datacenter → **Add Host**.
6. Enter ESXi host IP, credentials, and complete the wizard.
7. Select the host → **Configure → System → Time Configuration**.
8. Enable **NTP Client** and add NTP server details.
9. Create **Host and Cluster folders**.
10. Create **VM and Template folders** for organization.
11. Explore the **vSphere Client navigation pane**.

Result:

vCenter Server Appliance was configured with hosts, datacenter, NTP, and folders.

Practical 3: Users, Groups, and Permissions

Objective:

To integrate Active Directory with vCenter and assign permissions.

Procedure:

1. Navigate to **Administration → System Configuration**.
2. Join the vCenter Server Appliance to **vclass.local domain**.
3. Add **vclass.local** as an **Identity Source**.
4. Browse and view **Active Directory users and groups**.
5. Select a vCenter object → **Add Permission**.
6. Assign permissions to an AD user.
7. Configure **Root-Level Global Permissions**.
8. Log out and log in using **Windows Session Authentication**.
9. Manage a VM using an AD user account.

Result:

Active Directory integration and permission management were successfully completed.

Practical 4: Using Standard Switches

Objective:

To configure standard switches and virtual machine port groups.

Procedure:

1. Select ESXi Host → **Networking → Virtual Switches**.
2. View existing **Standard Switch configuration**.
3. Click **Add Standard Virtual Switch**.
4. Configure uplinks and MTU.
5. Create a **Virtual Machine Port Group**.
6. Edit VM settings → Change network to new port group.

Result:

Standard Switch and VM Port Group were created and attached to virtual machines.

Practical 5: Accessing iSCSI Storage

Objective:

To configure and connect iSCSI storage.

Procedure:

1. Verify existing iSCSI configuration under **Storage Adapters**.
2. Create a **VMkernel Port Group** for iSCSI.
3. Enable **iSCSI Software Adapter**.

4. Add **Dynamic/Static Discovery targets**.
5. Rescan storage adapters.

Result:

iSCSI storage was successfully connected to the ESXi host.

Practical 6: Managing VMFS Datastores

Objective:

To create, extend, expand, and remove VMFS datastores.

Procedure:

1. Navigate to **Storage → New Datastore**.
2. Select **VMFS** and choose the disk.
3. Create VMFS datastore.
4. Expand datastore to consume unused LUN space.
5. Extend VMFS datastore.
6. Remove VMFS datastore safely.
7. Create second shared VMFS datastore using iSCSI.

Result:

VMFS datastores were created and managed successfully.

Practical 7: Accessing NFS Storage

Objective:

To configure and verify NFS storage.

Procedure:

1. Go to **Storage → New Datastore**.
2. Select **NFS**.
3. Enter NFS server IP and shared folder path.
4. Mount the datastore.
5. View NFS datastore details.

Result:

NFS storage was successfully configured and accessed.

Practical 8: Using Templates and Clones

Objective:

To create VM templates and deploy VMs.

Procedure:

1. Power off a VM → Convert to **Template**.
2. Create **Customization Specifications**.
3. Deploy a VM from the template.

Result:

Virtual machines were deployed using templates.

Practical 9: Modifying Virtual Machines

Objective:

To modify VM hardware and settings.

Procedure:

1. Clone a powered-on VM.
2. Increase **VMDK disk size**.
3. Adjust **Memory allocation**.
4. Rename the VM.
5. Add and remove **Raw LUN**.

Result:

Virtual machine configuration changes were successfully applied.

Practical 10: Migrating Virtual Machines (vMotion)

Objective:

To migrate VMs using vSphere vMotion.

Procedure:

1. Create VMkernel port for **vMotion**.
2. Prepare VMs for migration.
3. Perform **vMotion migration**.
4. Perform **compute and storage migration**.

Result:

Virtual machines were migrated successfully using vMotion.

Practical 11: Managing Virtual Machines

Objective:

To manage VM registration and snapshots.

Procedure:

1. Unregister and register VMs.
2. Delete VM from datastore.
3. Take VM snapshot.
4. Modify VM and take another snapshot.
5. Revert to snapshot.
6. Delete individual and all snapshots.

Result:

Virtual machine lifecycle and snapshots were managed successfully.

Practical 12: Managing Resource Pools

Objective:

To create and verify resource pools.

Procedure:

1. Create CPU contention.
2. Create resource pools.
3. Verify resource allocation.

Result:

Resource pools were created and tested successfully.

Practical 13: Monitoring VM Performance

Objective:

To monitor VM CPU performance.

Procedure:

1. Create CPU workload.
2. Use performance charts to monitor CPU.
3. Undo configuration changes.

Result:

Virtual machine performance was monitored successfully.

Practical 14: Using vSphere HA

Objective:

To configure and test vSphere High Availability.

Procedure:

1. Create HA-enabled cluster.
2. Add ESXi hosts.
3. Test HA functionality.
4. View resource usage.
5. Configure slot size.
6. Enable strict admission control.

Result:

vSphere HA cluster was configured and tested successfully.