**PRACTICAL 01(A) (TITANIC.csv)**

**1(A) Load a CSV dataset handle missing values, inconsistent formatting, outliers.**

**!pip install pandas**

**!pip install numpy**

**!pip install seaborn**

**!pip install matplotlib**

**# Import necessary libraries**

**import pandas as pd**

**import numpy as np**

**import seaborn as sns**

**import matplotlib.pyplot as plt**

**#Importing the dataset**

**dataset = pd.read\_csv('/content/titanic.csv')**

**dataset.describe()**

**# Display the first few rows**

**print(dataset.head())**

**# Check for missing values**

**print("Missing values in each column:")**

**print(dataset.isnull().sum())**

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

**dataset['age'].fillna(dataset['age'].mean(), inplace=True)**

**# Fill missing values in 'Embarked' with the most common value**

**dataset['embarked'].fillna(dataset['embarked'].mode()[0], inplace=True)**

**# Verify missing values are handled**

**print("\nAfter handling missing values:")**

**print(dataset.isnull().sum())**

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

**dataset['sex'] = dataset['sex'].str.lower().str.strip()**

**# Verify unique values**

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

**print(dataset['sex'].unique())**

**# Boxplot for the 'Fare' column**

**sns.boxplot(dataset['fare'],color='skyblue')**

**plt.title('Boxplot of Fare')**

**plt.show()**

**# Detect outliers using the IQR method**

**Q1 = dataset['fare'].quantile(0.25)**

**Q3 = dataset['fare'].quantile(0.75)**

**IQR = Q3 - Q1**

**lower\_bound = Q1 - 1.5 \* IQR**

**upper\_bound = Q3 + 1.5 \* IQR**

**# Capping outliers**

**# Replacing 'data' with 'dataset' to ensure consistency**

**dataset['fare'] = np.where(dataset['fare'] > upper\_bound, upper\_bound, np.where(dataset['fare'] < lower\_bound, lower\_bound, dataset['fare']))**

**# Verify with an updated boxplot**

**# Replacing 'data' with 'dataset' to ensure consistency**

**sns.boxplot(dataset['fare'], color='lightgreen')**

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

**plt.show()**

**# Capping outliers - This section seems redundant, removing it**

**# 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 - This section seems redundant, removing it**

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

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

**# plt.show()**

**# Save the cleaned dataset**

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

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

**PRACTICAL 01-(C)**

**1(C)Create or explore datasets to used all preprocessing routines like label encoding, scaling and binarization.**

**!pip install pandas**

**!pip install numpy**

**!pip install sklearn**

**!pip install matplotlib**

**!pip install sklearn.preprocessing**

**!pip install LabelEncoder**

**!pip install MinMaxScaler**

**!pip install StandardScaler**

**!pip install Binarizer**

**import pandas as pd**

**import numpy as np**

**from sklearn.preprocessing import LabelEncoder**

**from sklearn.preprocessing import MinMaxScaler**

**from sklearn.preprocessing import StandardScaler**

**from sklearn.preprocessing import Binarizer**

**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':**

**'Has\_Car': ['Yes', 'No', 'Yes', 'No', 'Yes', 'No']**

**# Binary categorical variable**

**})**

**# Display the dataset**

**print("Sample Dataset:")**

**print(data)**

**# Display the dataset**

**print("Sample Dataset:")**

**print(data)**

**# 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)**

**# Save the processed dataset**

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

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

**PRACTICAL 02-(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 a .CSV file and generate the final hypothesis.**

import pandas as pd

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):

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

  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:

      import pandas as pd

      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']

}

df = pd.DataFrame(data)

# Step 2: Implementing the FIND-S Algorithm

def find\_s\_algorithm(data):

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

  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

hypothesis = find\_s\_algorithm(df)

print("The most specific hypothesis is:")

print(hypothesis)

print(hypothesis)

**PRACTICAL 03-(A)(HOUSE-PRICE.csv)**

**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.txt**

!pip install pandas

!pip install numpy

!pip install matplotlib

!pip install sklearn

!pip install sklearn.model\_selection

!pip install sklearn.linear\_model

!pip install sklearn.metrics

!pip install r2\_score

!pip install mean\_squared\_error

!pip install train\_test\_split

!pip install LinearRegression

# 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

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('/content/house-prices.csv', index=False)

# Display the dataset

print("Dataset:")

print(df)

dataset = pd.read\_csv('/content/house-prices.csv')

# Display the first few rows

print("\nLoaded Dataset:")

print(dataset.head())

# 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])

# 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\_)

# 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)

# 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)

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()

**PRACTICAL 03-(B)(TITANIC.csv)**

3b Multiple Linear Regression Extend linear regression to multiple feature. Handle feature selection and potential multicollinearity.txt

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

#Importing the dataset

dataset = pd.read\_csv('/content/titanic.csv')

# Read the CSV file, replace 'titanic.csv' with the correct path if needed

data = pd.read\_csv('/content/titanic.csv')

# Display the first few rows

print(data.head())

# Check for null values and basic statistics

print(data.info())

print(data.describe())

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, but keep at least one row

  # If all rows would be dropped, keep the first row

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

  if numeric\_df.dropna().empty:

    numeric\_df = numeric\_df.iloc[[0]].fillna(0) # Keep at least one row

  else:

    numeric\_df = numeric\_df.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

  X = data.drop("Survived", axis=1)

  y = data["Survived"] # Assign the target variable 'Survived' to 'y'

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

  le = LabelEncoder()

  X[col] = le.fit\_transform(X[col])

  imputer = SimpleImputer(strategy='mean')

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

  print("VIF before handling multicollinearity:")

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

if 'X1' in X.columns and len(X.select\_dtypes(include=np.number).columns) > 1:

  X = X.drop("X1", axis=1)

else:

  print("Column 'X1' not found or dropping it would leave no numeric features.")

  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)

  model = LinearRegression()

  model.fit(X\_train, y\_train)

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

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

  y\_pred = model.predict(X\_test)

  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()

**PRACTICAL 03-(C)**

3(C)Regualarized Linear Models Implement Regression variants like LASSO and Ridge on any generated dataset.txt

!pip install pandas

!pip install numpy

!pip install matplotlib

!pip install seaborn

!pip install sklearn

!pip install scikit-learn

!pip install sklearn\_linear\_model

!pip install sklearn\_metrics

!pip install sklearn\_datasets

!pip install sklearn\_model\_selection

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

import numpy as np

np.random.seed(42)

# Generate synthetic data

from sklearn.datasets import make\_regression

import pandas as pd

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())

# 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

)

# 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}")

# 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\_)

# 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

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}")

# 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()

**PRACTICAL 04-(b)**

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..txt

!pip install pandas

!pip install numpy

!pip install sklearn

!pip install google.colab

!pip install sklearn\_model\_selection

!pip install sklearn\_neighbors

!pip install sklearn\_metrics

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

# 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()

 # Handle potential errors and extract filename

 if uploaded:  # Check if upload was successful

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

 else:

   print("Upload failed or canceled.")

   filename = None  # Or handle the failure appropriately

else:

  # Create a synthetic dataset

  from sklearn.datasets import make\_classification

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

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

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

  data['Target'] = y

  filename = "synthetic\_data.csv"

  data.to\_csv(filename, index=False)

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

  X = data.iloc[:, :-1].values

  y = data.iloc[:, -1].values

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

  knn = KNeighborsClassifier(n\_neighbors=3)

  knn.fit(X\_train, y\_train)

  y\_pred = knn.predict(X\_test)

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

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

  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]}")

**PRACTICAL 04-(C)(IRIS.csv)**

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

**!pip install pandas**

**!pip install numpy**

**!pip install sklearn**

**!pip install matplotlib**

**!pip install sklearn\_model\_selection**

**!pip install sklearn\_tree**

**!pip install sklearn\_metrics**

**!pip install matplotlib\_pyplot**

**!pip install google\_colab**

**# Import necessary libraries**

**import pandas as pd  # This line imports the pandas library and assigns it to the alias '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**

**# 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**

**# The error was here, need to separate the DataFrame creation and column assignment**

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

**data['Target'] = y # Assign the Target column on a separate line**

**# 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}.")**

**# Load the dataset**

**data =pd.read\_csv('/content/iris.csv')**

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

**print("Dataset Preview:")**

**print(data.head())**

**# 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)**

**# 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)**

**# 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}")**

**# Visualize the decision tree**

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

**plot\_tree(model, feature\_names=data.columns[:-1],**

**class\_names=np.unique(y) if task\_type == "classification" else None, # Changed this line**

**filled=True)**

**plt.title("Decision Tree Visualization")**

**plt.show()**

**PRACTICAL 04-(D)**

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

**!pip install pandas**

**!pip install numpy**

**!pip install sklearn**

**!pip install matplotlib**

**!pip install google.colab**

**!pip install google.colab.files**

**!pip install scikit-learn**

**!pip install sklearn\_model\_selection**

**!pip install sklearn\_svm**

**!pip install sklearn\_metrics**

**!pip install sklearn\_datasets**

**!pip install sklearn\_classification\_report**

**!pip install accuracy\_score**

**# 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**

**# 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])]) # pd is used here to call pandas functions**

**data['Target'] = y**

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

**filename="titanic.csv"**

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

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

**# 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())**

**# 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)**

**# 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)**

**# Predict the labels for the test set**

**y\_pred = svm\_model.predict(X\_test)**

**# 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))**

**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()**

**PRACTICAL 04-(E)**

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

!pip install pandas

!pip install numpy

!pip install sklearn

!pip install matplotlib

!pip install google.colab

!pip install sklearn\_model\_selection

!pip install sklearn\_tree

!pip install sklearn\_ensemble

!pip install sklearn\_metrics

# Import necessarylibraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

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

# 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 = "titanic.csv"

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

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

# Load the dataset

data =pd.read\_csv(filename)

# Display the first few rows of the dataset

print("Dataset Preview:")

print(data.head())

# 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)

# 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}")

# 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}")

# 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}")

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}")

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()

**PRACTICAL 05-(A)**

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

!pip install pandas

!pip install numpy

!pip install sklearn

!pip install matplotlib

!pip install seaborn

!pip install google.colab

!pip install sklearn.metrics

!pip install sklearn.model\_selection

!pip install sklearn.naive\_bayes

!pip install GaussianNB

# 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

# 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="titanic.csv"

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

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

# Load the dataset

data =pd.read\_csv(filename)

# Display the first few rows of the dataset

print("Dataset Preview:")

print(data.head())

# 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)

# Initialize the Gaussian Naive Bayes classifier

naive\_bayes = GaussianNB()

# Train the model

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

# 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))

# 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]}")

**PRACTICAL 05-(B)**

5b Implement Hidden Markov Models using hmmlearn.txt

!pip install hmmlearn

!pip install numpy

!pip install pandas

!pip install matplotlib

# Import necessary libraries

import numpy as np

import pandas as pd

from hmmlearn import hmm

import matplotlib.pyplot as plt

# 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())

# 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)

# 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], [0.4, 0.6]])  # Corrected transition probabilities

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\_)

# 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)

# 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()

**PRACTICAL 06-(A)**

**6a Implement Bayesian Linear Regression to explore prior and posterior distribution..txt**

**!pip install pandas**

**!pip install numpy**

**!pip install matplotlib**

**!pip install seaborn**

**!pip install sklearn**

**!pip install google.colab**

**!pip install sklearn.linear\_model**

**!pip install sklearn.model\_selection**

**!pip install sklearn.metrics**

**# 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**

**# 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="titanic.csv"**

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

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

**# Load the dataset (for CSV file)**

**data = pd.read\_csv(filename)**

**# Display first few rows**

**print("Dataset Preview:")**

**print(data.head())**

**# 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)**

**# 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)**

**# 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()**

**#Calculate the Mean Squared Error (MSE)**

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

**print(f"Mean Squared Error (MSE):{mse:.2f}")**

**# 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()**

**PRACTICAL 06-(B)**

**6b Implement Gaussian Mixture Models for density estimation and unsupervised clustering..txt**

**!pip install numpy**

**!pip install pandas**

**!pip install matplotlib**

**!pip install seaborn**

**!pip install sklearn**

**!pip install google.colab**

**!pip install sklearn\_mixture**

**!pip install sklearn\_model\_selection**

**!pip install sklearn\_mixture\_GaussianMixture**

**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**

**#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 = "titanic.csv"**

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

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

**# 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()**

**# 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()**

**# Extract the means and covariances of the Gaussian components**

**means = gmm.means\_**

**covariances = gmm.covariances\_  # Assign covariances correctly**

**# 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):  # Uncomment and correct the loop**

**# Plot the Gaussian components as ellipses**

**v, w = np.linalg.eigh(covar)  # Use covar instead of covariances**

**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()**

**# Compute the log-likelihood of the data under the fitted GMM model**

**# Select only the features used during training**

**log\_likelihood = gmm.score(data[['Feature\_1', 'Feature\_2']])**

**print(f"Log-Likelihood of the data: {log\_likelihood:.2f}")**

**# 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}")**

**PRACTICAL 07-(A)**

**7a Implement cross-validation techniques (k-fold, stratified etc.) for robust model evaluation.txt**

**!pip install numpy**

**!pip install pandas**

**!pip install scikit-learn**

**!pip install matplotlib**

**!pip install seaborn**

**!pip install sklearn\_datasets**

**!pip install sklearn\_model\_selection**

**!pip install sklearn\_ensemble**

**!pip install sklearn\_metrics**

**!pip install matplotlib\_pyplot**

**!pip install seaborn\_sns**

**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**

**# 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())**

**# 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)**

**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)**

**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)**

**# 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}")**

**# 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\_)**

**# 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', 'Class1'], yticklabels=['Class 0', 'Class 1'])**

**plt.xlabel('Predicted')**

**plt.ylabel('Actual')**

**plt.title('Confusion Matrix')**

**plt.show()**

**PRACTICAL 07-(B)**

**7b Systematically explore combinations of hyperparameters to optimize model performance.(use grid and randomized search).txt**

**!pip install numpy**

**!pip install pandas**

**!pip install scikit-learn**

**!pip install matplotlib**

**!pip install seaborn**

**!pip install sklearn\_model\_selection**

**!pip install sklearn\_ensemble**

**!pip install sklearn\_metrics**

**!pip install sklearn\_datasets**

**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**

**# 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())**

**# 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)**

**# Initialize a Random Forest classifier**

**model = RandomForestClassifier(random\_state=42)**

**# 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\_)**

**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\_)**

**# 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()**

**PRACTICAL 08**

8 Implement Bayesian Learning using inferences.txt

!pip install pandas

!pip install numpy

!pip install sklearn

!pip install seaborn

!pip install matplotlib

!pip install sklearn.datasets

!pip install sklearn.model\_selection

!pip install sklearn.naive\_bayes

!pip install sklearn.metrics

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

# 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())

# 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)

# 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)

# 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)

# Added a line break here to separate the statements

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Class 0', 'Class1'], yticklabels=['Class 0', 'Class 1'])

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()

# 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)}")