Practical No. 1
Data Pre-processing and Exploration

(A). Load a CSV dataset. Handle missing values, inconsistent formatting, and outliers. Here are the general steps for cleaning a dataset:

Solution:

Load the CSV: Read the dataset from a CSV file.
Handle Missing Values: Identify and either fill or drop missing values.
Handle Inconsistent Formatting: Standardize formats (e.g., date formats, numerical columns).
Handle Outliers: Detect and deal with outliers, either by removing or transforming them.
Let's create a sample CSV and process it step by step. I'll simulate the data and demonstrate how you can clean it.

Step 1: Generate a Sample CSV Dataset
Here is a sample dataset that contains missing values, inconsistent formats, and outliers:
Now, let's load and clean this dataset:

Step 2: Handle Missing Values
Missing Age: We could either fill the missing value with the mean or median age or drop that row.
Missing Income: We could fill it with the mean or median income.
Missing Name: We could either drop the row or replace it with a placeholder value.

Step 3: Handle Inconsistent Formatting
The date format should be consistent (e.g., all yyyy-mm-dd).
If there are mixed types in a column (such as numbers and text in the Age column), they need to be cleaned.

Step 4: Handle Outliers
The Income column might have extreme outliers (e.g., David's 300,000), which we can handle by capping or removing.
Steps Explained:
Handling Missing Values: We use fillna() to replace missing values in Age, Income, and Name. Age and Income are filled with their median values, while missing Name values are replaced with "Unknown".
Handling Inconsistent Formatting: We ensure the Join_Date is in a consistent format (YYYY-MM-DD) by converting it using pd.to_datetime().
Handling Outliers: Outliers in the Income column are capped using the 95th percentile. This helps prevent extreme values from skewing the analysis.

Code:

```
import pandas as pd
import numpy as np
from scipy import stats

# Load Titanic dataset (replace with your actual file path)
df = pd.read_csv('titanic.csv')  # Adjust file name if necessary

# Display first few rows to understand the structure
print("Initial Data:")
print(df.head())

# 1. Handle Missing Values
```

```
# For simplicity, we will fill missing values in numeric columns with the column mean.
# For categorical columns, we will fill missing values with the mode (most frequent value).

# Fill missing values in numeric columns with the mean
df['Age'] = df['Age'].fillna(df['Age'].mean())
df['Fare'] = df['Fare'].fillna(df['Fare'].mean())

# Fill missing values in categorical columns with the mode
df['Embarked'] = df['Embarked'].fillna(df['Embarked'].mode()[0])
df['Cabin'] = df['Cabin'].fillna('Unknown')  # Fill Cabin with a default string if missing

# 2. Handle Inconsistent Formatting
# Convert 'Sex' column to lowercase to standardize the text
df['Sex'] = df['Sex'].str.lower()

# Normalize 'Embarked' column by stripping extra spaces and ensuring consistency in uppercase
df['Embarked'] = df['Embarked'].str.strip().str.upper()

# 3. Handle Outliers using Z-score method
# We'll apply outlier detection on the numeric columns: 'Age' and 'Fare'

numeric_columns = ['Age', 'Fare']

# Calculate Z-scores for numeric columns to detect outliers
z_scores = np.abs(stats.zscore(df[numeric_columns]))

# Set a threshold for Z-scores to identify outliers (e.g., Z > 3)
threshold = 3
outliers = (z_scores > threshold)

# Remove outliers by filtering out rows where any numeric column exceeds the threshold
df_no_outliers = df[(z_scores < threshold).all(axis=1)]

# Display cleaned data
print("\nCleaned Data (after handling missing values, inconsistent formatting, and outliers):")
print(df_no_outliers.head())

# Save the cleaned dataset if needed
df_no_outliers.to_csv('cleaned_titanic_data.csv', index=False)
```

Output:

```
Initial Data:
   PassengerId  Survived  Pclass  ...     Fare Cabin  Embarked
0            1         0       3  ...   7.2500   NaN         S
1            2         1       1  ...  71.2833   C85         C
2            3         1       3  ...   7.9250   NaN         S
3            4         1       1  ...  53.1000  C123         S
4            5         0       3  ...   8.0500   NaN         S

[5 rows x 12 columns]

Cleaned Data (after handling missing values, inconsistent formatting, and outliers):
   PassengerId  Survived  Pclass  ...     Fare    Cabin  Embarked
0            1         0       3  ...   7.2500  Unknown         S
1            2         1       1  ...  71.2833      C85         C
2            3         1       3  ...   7.9250  Unknown         S
3            4         1       1  ...  53.1000     C123         S
4            5         0       3  ...   8.0500  Unknown         S

[5 rows x 12 columns]
```

(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.

To explore a dataset, calculate descriptive statistics, and visualize the data using univariate and bivariate graphs, we'll use Python's pandas, matplotlib, and seaborn libraries. Below, I'll walk you through the steps, including:

1. Loading the dataset
2. Calculating descriptive summary statistics
3. Creating univariate and bivariate visualizations
4. Identifying potential features and target variables

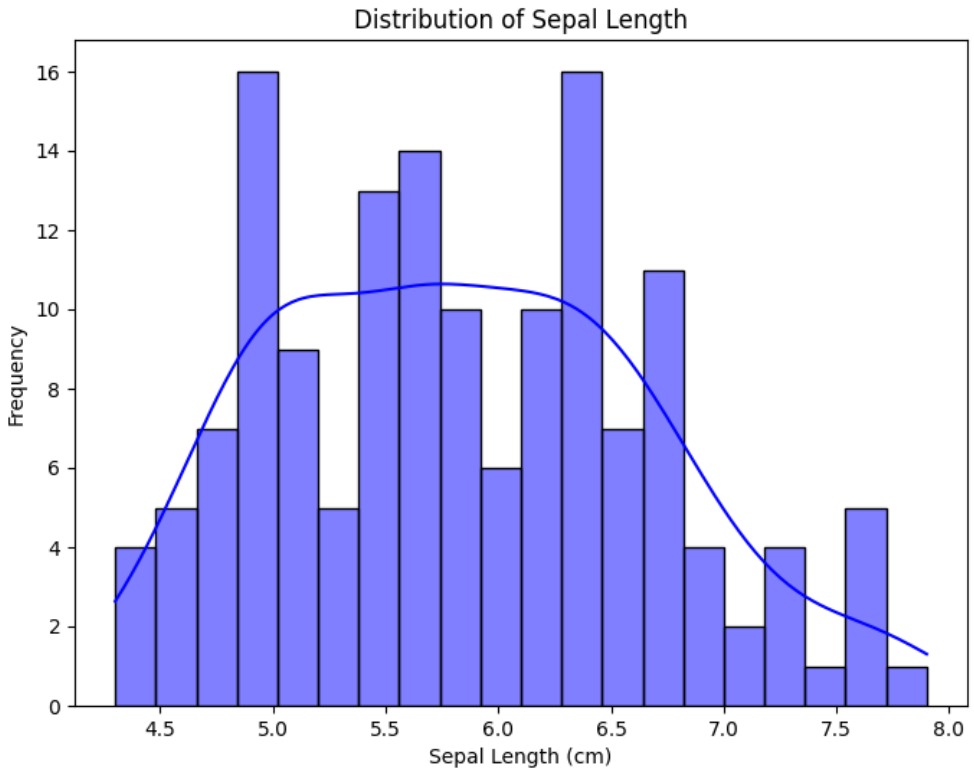
Code:

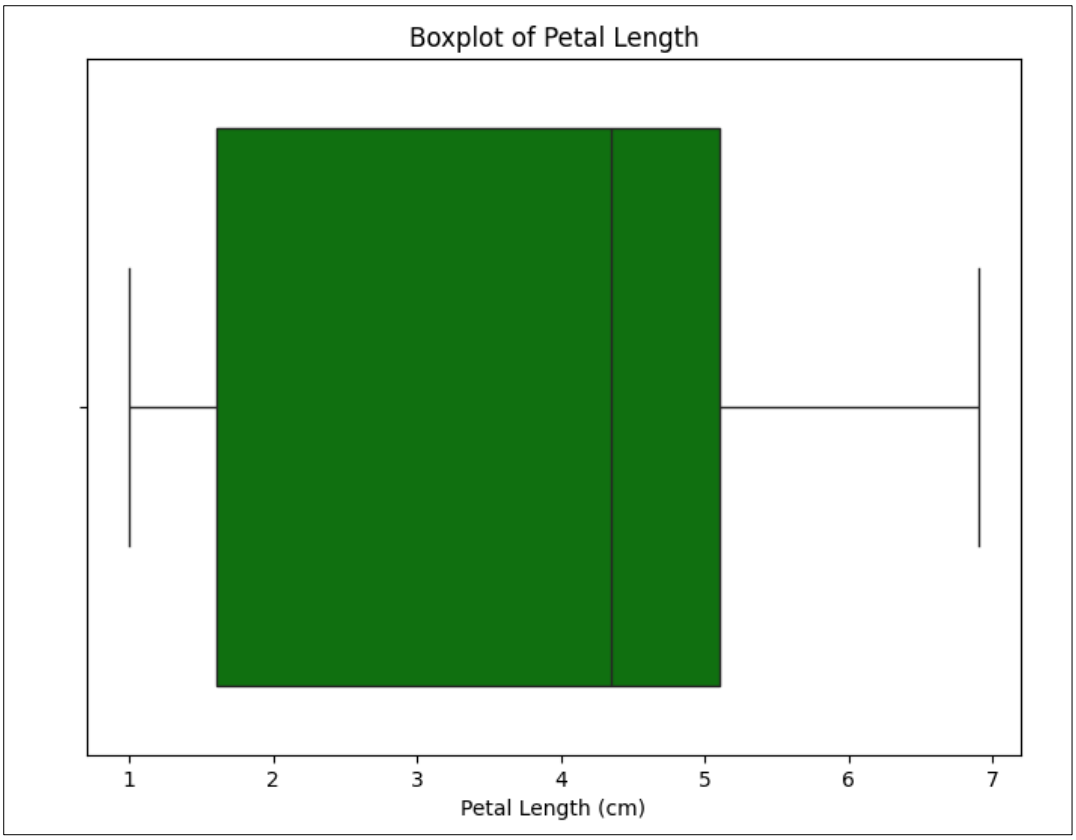
```
import matplotlib.pyplot as plt
import seaborn as sns

# Load the Iris dataset
df = sns.load_dataset("iris")

# 1. Descriptive Statistics
print(df.describe())

# 2. Univariate Visualizations
# a. Histogram for 'sepal_length'
plt.figure(figsize=(8, 6))
sns.histplot(df['sepal_length'], kde=True, bins=20, color='blue')
plt.title('Distribution of Sepal Length')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Frequency')
plt.show()

# b. Boxplot for 'petal_length'
plt.figure(figsize=(8, 6))
sns.boxplot(x=df['petal_length'], color='green')
plt.title('Boxplot of Petal Length')
plt.xlabel('Petal Length (cm)')
plt.show()

# c. Density plot for 'sepal_width'
plt.figure(figsize=(8, 6))
sns.kdeplot(df['sepal_width'], shade=True, color='red')
plt.title('Density Plot of Sepal Width')
plt.xlabel('Sepal Width (cm)')
plt.ylabel('Density')
plt.show()

# 3. Bivariate Visualizations
# a. Scatter plot for Sepal Length vs Sepal Width
```

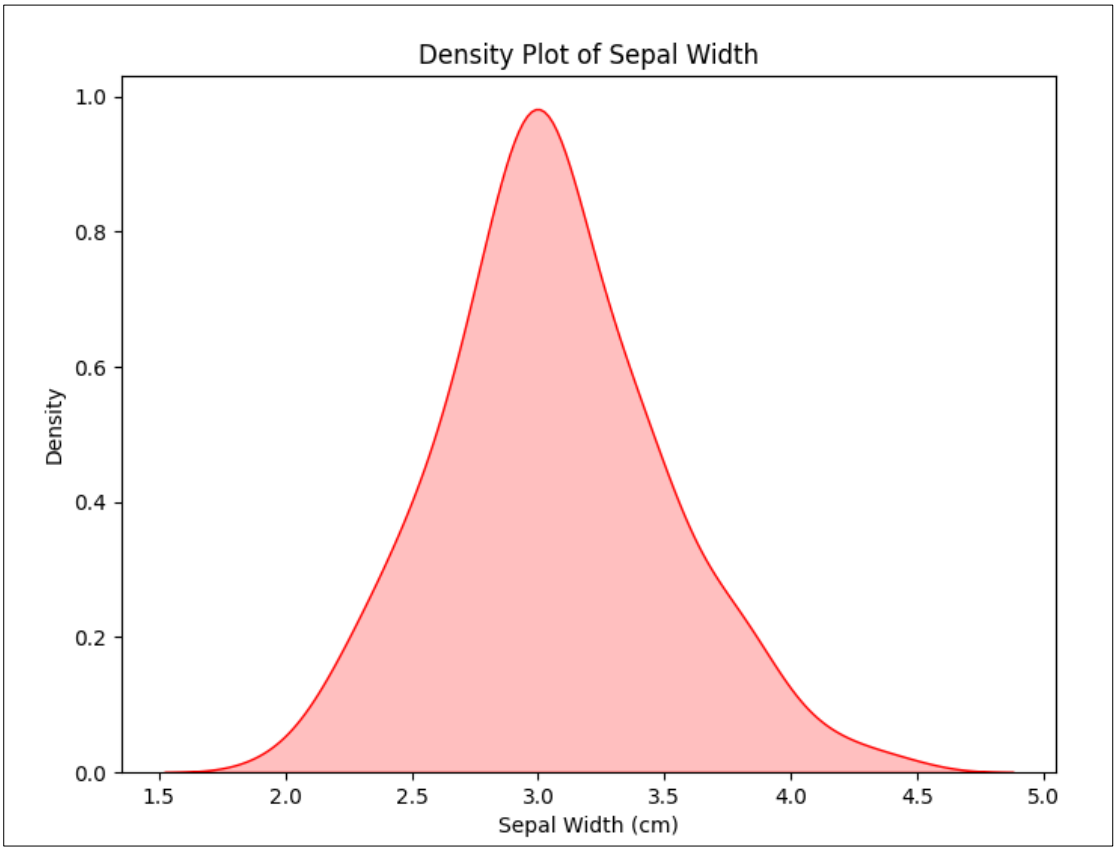
```
plt.figure(figsize=(8, 6))
sns.scatterplot(x='sepal_length', y='sepal_width', data=df, hue='species')
plt.title('Sepal Length vs Sepal Width')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.show()
# b. Pairplot of all numerical variables
sns.pairplot(df, hue='species')
plt.show()
# c. Correlation Heatmap
corr_matrix = df.corr()
plt.figure(figsize=(8, 6))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5)
plt.title('Correlation Heatmap of Numerical Features')
plt.show()
```

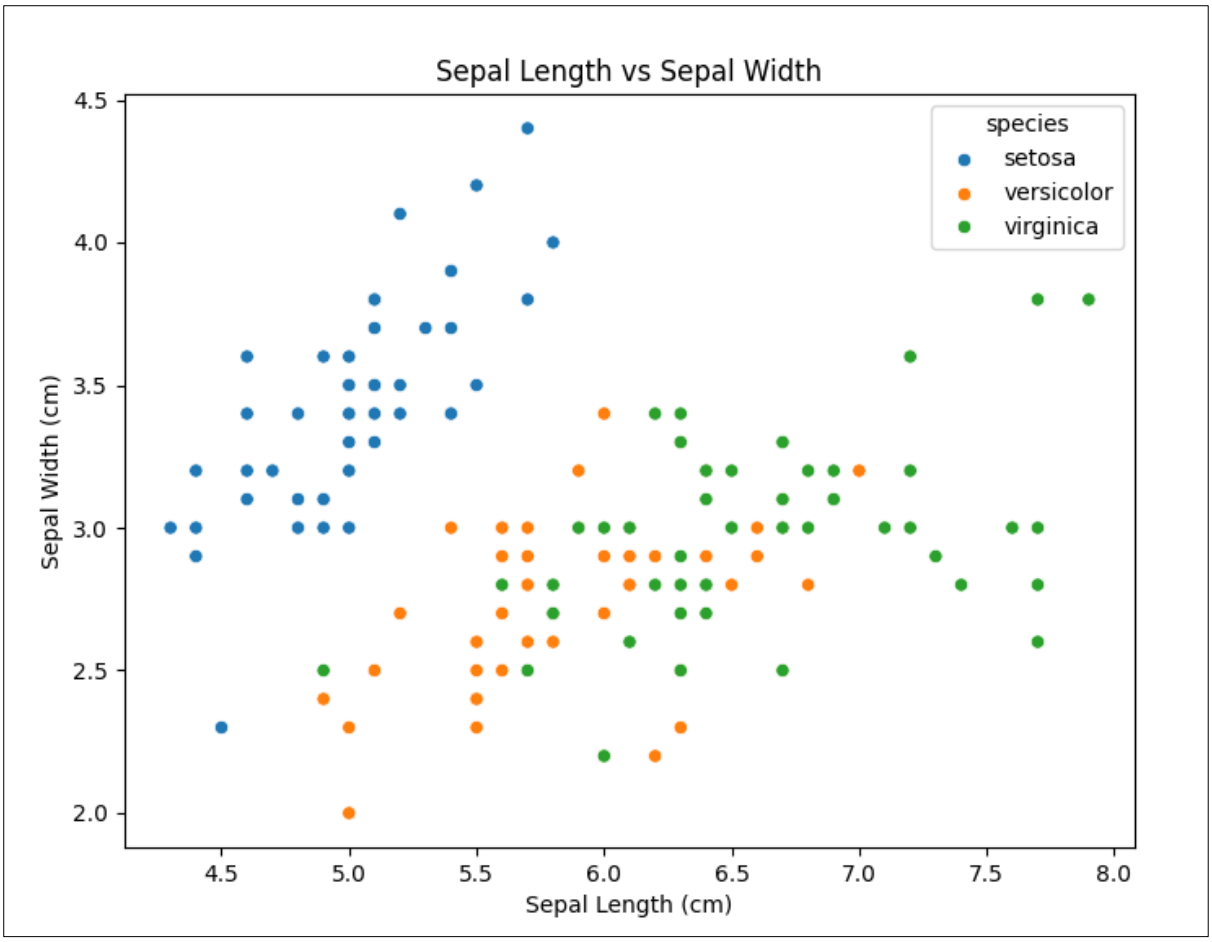
Output:

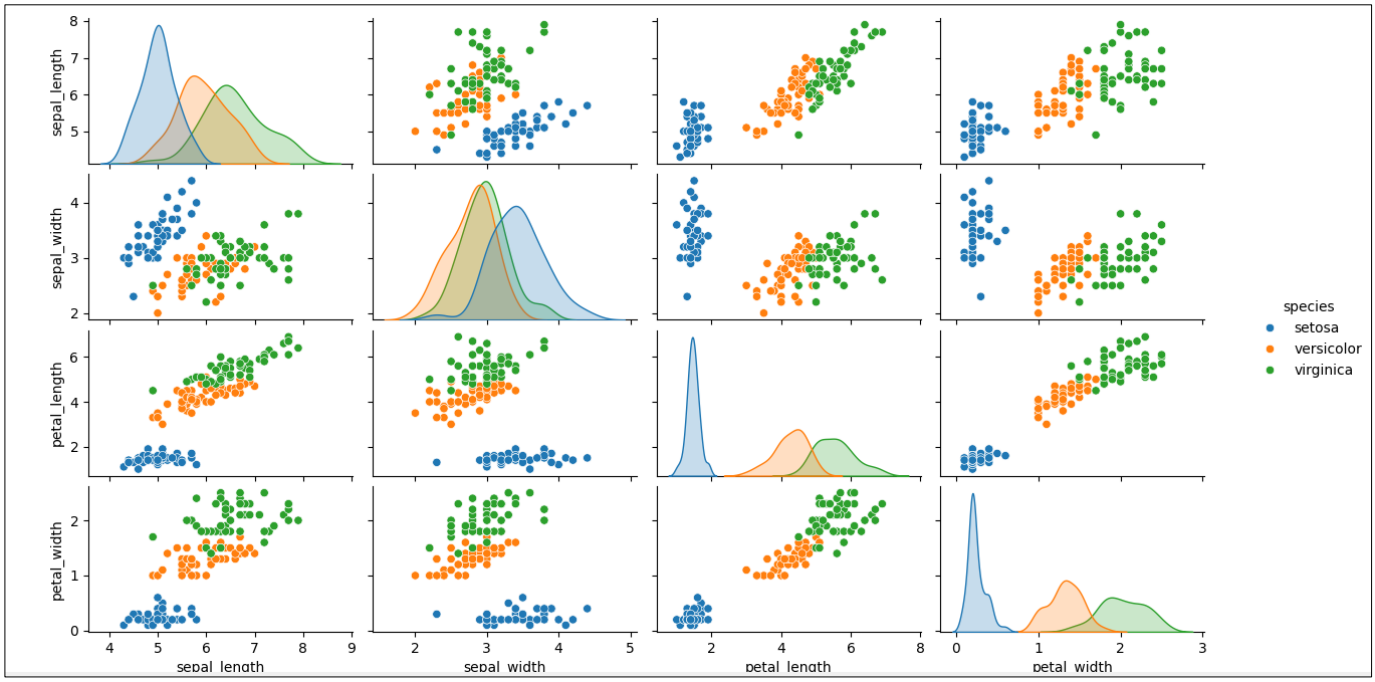
	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000




Boxplot of Petal Length
1
2
3
4
5
6
7
Petal Length (cm)


Density Plot of Sepal Width
1.0
0.8
0.6
0.4
0.2
0.0
Density
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Sepal Width (cm)


Sepal Length vs Sepal Width
Sepal Width (cm)
Sepal Length (cm)
species
setosa
versicolor
virginica
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
sepal_width
petal_length
petal_width
species
setosa
versicolor
virginica

(C). Create or Explore datasets to use all pre-processing routines like label encoding, scaling, and binarization.

Solution:

```
# Step 1: Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder, StandardScaler, MinMaxScaler, Binarizer

# Step 2: Create a synthetic dataset
data = {
   "Numerical_Column": [10, 20, 30, 40, 50],
   "Categorical_Column": ["Low", "Medium", "High", "Medium", "Low"],
   "Binary_Column": [1, 0, 1, 0, 1]
}

# Convert to a DataFrame
df = pd.DataFrame(data)

# Step 3: Handle missing values (if any)
# Here we simulate missing values and fill them
df.loc[1, "Numerical_Column"] = np.nan
df["Numerical_Column"].fillna(df["Numerical_Column"].mean(), inplace=True)

# Step 4: Apply Label Encoding to the categorical column
label_encoder = LabelEncoder()
df["Categorical_Column_Encoded"] = label_encoder.fit_transform(df["Categorical_Column"])

# Step 5: Scale numerical columns
scaler = StandardScaler()
df["Numerical_Scaled"] = scaler.fit_transform(df[["Numerical_Column"]])

# Min-Max Scaling for comparison
min_max_scaler = MinMaxScaler()
df["Numerical_MinMax_Scaled"] = min_max_scaler.fit_transform(df[["Numerical_Column"]])

# Step 6: Apply Binarization to numerical columns
binarizer = Binarizer(threshold=25)
df["Numerical_Binarized"] = binarizer.fit_transform(df[["Numerical_Column"]])

# Print the resulting DataFrame
print(df)
```

Output:

```
   Numerical_Column Categorical_Column  Binary_Column  Categorical_Column_Encoded  Numerical_Scaled  Numerical_MinMax_Scaled  Numerical_Binarized
0              10.0                Low              1                           1         -1.700840                   0.0000                  0.0
1              32.5             Medium              0                           2          0.000000                   0.5625                  1.0
2              30.0               High              1                           0         -0.188982                   0.5000                  1.0
3              40.0             Medium              0                           2          0.566947                   0.7500                  1.0
4              50.0                Low              1                           1          1.322876                   1.0000                  1.0
```

Practical No. 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 a. CSV file and generate the final specific hypothesis. (Create your dataset)

The **FIND-S** algorithm is a simple method for finding the most specific hypothesis that fits a set of positive training examples in a classification problem. The algorithm focuses on the positive examples and iteratively refines the hypothesis by setting the hypothesis to match the features of each positive example.

Steps of the FIND-S Algorithm:

1. **Initialize the hypothesis**: Start with the most specific hypothesis possible. In the case of an attribute-based classifier, this is usually a hypothesis with all features set to the most specific values (e.g., all attributes are set to 'None' or an impossible value).

2. **Iterate over positive examples**: For each positive example in the training set:
   - If the current hypothesis matches the example, do nothing.
   - If the hypothesis does not match the example, update the hypothesis to match the features of the current example.

3. **Final hypothesis**: The final hypothesis after processing all positive examples is the most specific hypothesis that fits the positive training samples.

Dataset:

Feature1	Feature2	Feature3	Class
Sunny	Hot	High	Yes
Sunny	Hot	Low	Yes
Overcast	Hot	High	Yes
Rainy	Mild	High	No
Rainy	Cool	High	No
Overcast	Mild	High	Yes
Sunny	Cool	High	No

Code:

```
import pandas as pd
# Step 1: Create a simple dataset and save it to a CSV file
data = {
   "Sky": ["Sunny", "Sunny", "Rainy", "Sunny", "Sunny"],
   "AirTemp": ["Warm", "Warm", "Cold", "Warm", "Warm"],
   "Humidity": ["Normal", "High", "High", "High", "Normal"],
   "Wind": ["Strong", "Strong", "Strong", "Strong", "Strong"],
   "Water": ["Warm", "Warm", "Warm", "Cool", "Warm"],
   "Forecast": ["Same", "Same", "Change", "Same", "Same"],
   "EnjoySport": ["Yes", "Yes", "No", "Yes", "Yes"]
}

# Save dataset to CSV for demonstration purposes
df = pd.DataFrame(data)
```

```
df.to_csv("training_data.csv", index=False)

# Step 2: Load the training data from CSV
training_data = pd.read_csv("training_data.csv")

# Step 3: Implement FIND-S algorithm
def find_s_algorithm(data):
    # Extract features and target
    features = data.iloc[:, :-1].values  # All columns except the last
    target = data.iloc[:, -1].values     # The last column (target)

    # Initialize the most specific hypothesis
    hypothesis = ["ϕ"] * features.shape[1]  # Start with the most specific hypothesis

    # Update hypothesis for positive examples
    for i, row in enumerate(features):
        if target[i] == "Yes":  # Consider only positive examples
            for j in range(len(hypothesis)):
                if hypothesis[j] == "ϕ":  # Update if still specific
                    hypothesis[j] = row[j]
                elif hypothesis[j] != row[j]:  # Generalize if mismatch
                    hypothesis[j] = "?"

    return hypothesis

# Step 4: Run the FIND-S algorithm
final_hypothesis = find_s_algorithm(training_data)

# Step 5: Display the results
print("Final Specific Hypothesis:", final_hypothesis)
```

Explanation of the Code:

1. **Read the data**: We read the CSV file using pandas.read_csv() and store it in the data variable. The feature columns are stored in X, and the target labels are stored in y.

2. **Initialize the hypothesis**: We start by setting the hypothesis to match the first positive example in the dataset (i.e., the first row where Class is 'Yes').

3. **Refine the hypothesis**: We loop through the remaining positive examples. For each positive example, if the value of a feature in the hypothesis does not match the feature in the current example, we generalize the hypothesis by setting that feature to ? (i.e., the most general value).

4. **Final Hypothesis**: After processing all the positive examples, the hypothesis represents the most specific classifier that fits the training data.

Output:

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

Important Notes:

- **CSV File Structure**: Make sure the CSV file is structured correctly, where the target column is labeled (e.g., Class), and the features are categorical (e.g., 'Sunny', 'Hot', 'High').
- **Generalization**: The generalization in FIND-S works by replacing any attribute that doesn't match across all positive examples with ?. This means the hypothesis becomes more general as we encounter examples with differing features.

This simple algorithm is particularly useful for small datasets with easily identifiable patterns, especially when the goal is to create a highly specific classifier based on positive examples.
Let me know if you need further clarification!

Practical No. 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

Linear Regression model on a dataset, interpreting the coefficients, making predictions, and evaluating performance using metrics like R-squared and Mean Squared Error (MSE).

We'll generate a synthetic dataset and perform the following steps:

1. Generate a synthetic dataset using sklearn.datasets.make_regression.
2. Fit a Linear Regression model.
3. Interpret the model coefficients.
4. Make predictions.
5. Evaluate the model performance using R-squared and Mean Squared Error (MSE).

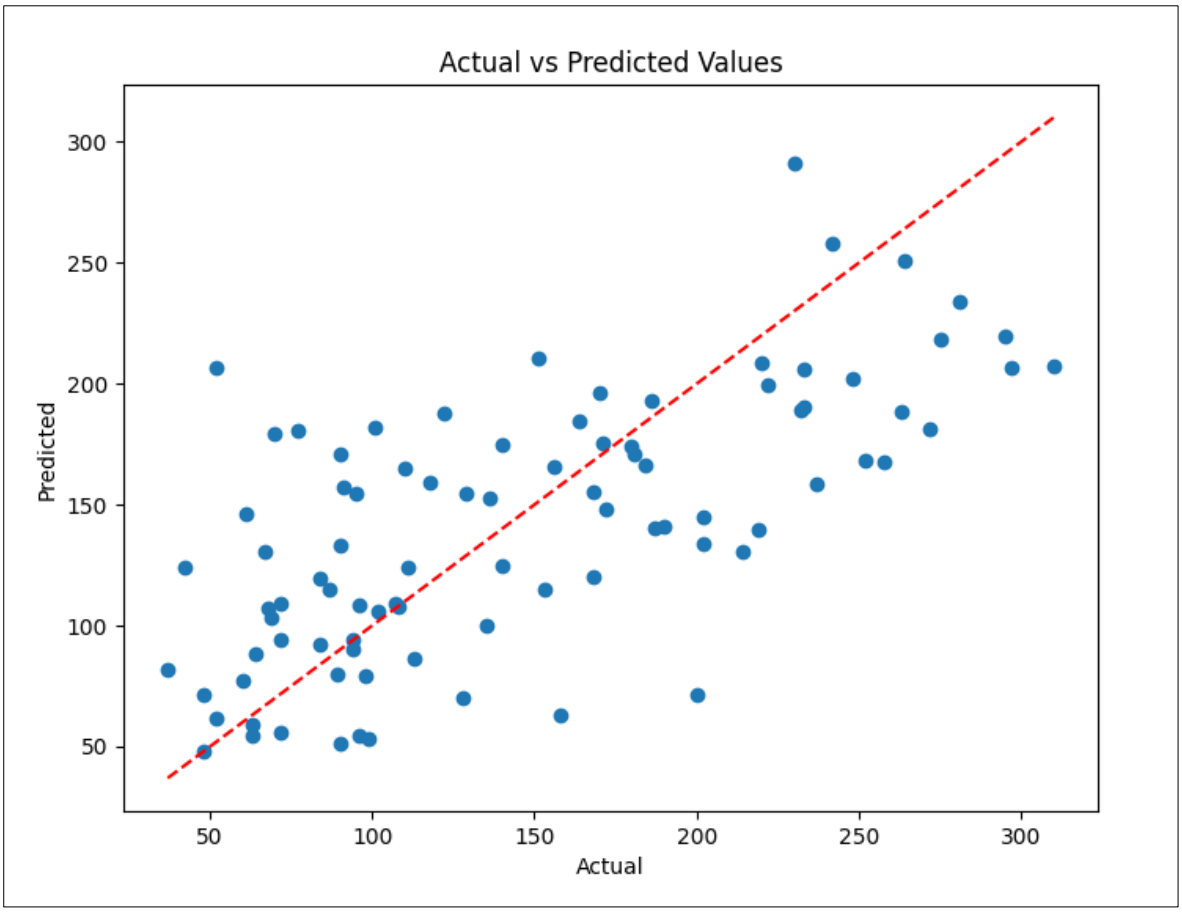
Code:

```
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 sklearn.datasets import load_diabetes

# Load the diabetes dataset
data = load_diabetes()
# Convert the dataset to a pandas DataFrame
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target
# Show the first few rows of the dataframe
print(df.head())
# Split the data into features (X) and target (y)
X = df.drop('target', axis=1)
y = df['target']
# Split the dataset 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)
# Create a linear regression model
model = LinearRegression()
# Train the model
model.fit(X_train, y_train)
# Make predictions
y_pred = model.predict(X_test)
# Calculate the mean squared error and R2 score
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

```
print(f"Mean Squared Error: {mse}")
print(f"R2 Score: {r2}")
# Visualize the predicted vs. actual values
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red', linestyle='--')
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.title("Actual vs Predicted Values")
plt.show()
```

Output:

```
        age       sex       bmi        bp  ...        s4        s5        s6  target
0  0.038076  0.050680  0.061696  0.021872  ... -0.002592  0.019907 -0.017646   151.0
1 -0.001882 -0.044642 -0.051474 -0.026328  ... -0.039493 -0.068332 -0.092204    75.0
2  0.085299  0.050680  0.044451 -0.005670  ... -0.002592  0.002861 -0.025930   141.0
3 -0.089063 -0.044642 -0.011595 -0.036656  ...  0.034309  0.022688 -0.009362   206.0
4  0.005383 -0.044642 -0.036385  0.021872  ... -0.002592 -0.031988 -0.046641   135.0

[5 rows x 11 columns]
Mean Squared Error: 2900.19362849348
R2 Score: 0.4526027629719197
```



(B). Multiple Linear Regression
Extend linear regression to multiple features. Handle feature selection and potential multicollinearity.

Extending **Linear Regression** to handle multiple features is straightforward, as **Linear Regression** can naturally handle multiple features (multiple linear regression). However, when working with multiple features, we need to be cautious about issues like **feature selection** and **multicollinearity**.

Key Concepts:

1. **Multiple Linear Regression**: Linear regression with more than one independent variable.
2. **Feature Selection**: Selecting the most relevant features to include in the model to improve accuracy and reduce complexity.
3. **Multicollinearity**: Occurs when two or more features are highly correlated with each other. This can cause instability in the coefficient estimates.

Steps:

1. Extend Linear Regression to multiple features.
2. Handle feature selection using methods like Recursive Feature Elimination (RFE) or L1 regularization (Lasso).
3. Detect and handle multicollinearity using the Variance Inflation Factor (VIF) and removing highly correlated features.

Code:

```
# Import necessary libraries
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, Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score
from statsmodels.stats.outliers_influence import variance_inflation_factor
from statsmodels.tools.tools import add_constant
from sklearn.datasets import load_diabetes

# Load the Diabetes dataset
diabetes = load_diabetes()

# Convert it into a pandas DataFrame
df = pd.DataFrame(diabetes.data, columns=diabetes.feature_names)

# Add the target variable (diabetes progression) to the dataframe
df['PROGRESSION'] = diabetes.target
```

```
# Display the first few rows of the dataset
print("Dataset Overview:")
print(df.head(), "\n")

# Check for multicollinearity using the correlation matrix
corr_matrix = df.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f", linewidths=0.5)
plt.title("Correlation Matrix")
plt.show()

# Check Variance Inflation Factor (VIF) to detect multicollinearity
X = df.drop(columns=['PROGRESSION'])
X_with_const = add_constant(X)  # Add constant to the features
vif_data = pd.DataFrame()
vif_data["Feature"] = X_with_const.columns
vif_data["VIF"] = [variance_inflation_factor(X_with_const.values, i) for i in
range(X_with_const.shape[1])]

print("\nVariance Inflation Factor (VIF):")
print(vif_data, "\n")

# Remove features with high VIF (>10), or use Lasso Regression to handle feature selection
X_selected = X.drop(columns=['s5', 's4'])  # Drop highly correlated features, based on VIF
analysis

# Split the dataset into features (X) and target (y)
X = X_selected
y = df['PROGRESSION']

# Standardize the features (important for regularization)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Initialize and train the Multiple Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Get the model's coefficients and intercept
intercept = model.intercept_
coefficients = model.coef_

print(f"\nIntercept: {intercept}")
print(f"Coefficients: {coefficients}\n")
```

```
# Make predictions using the test set
y_pred = model.predict(X_test)

# Compare actual vs predicted values
comparison_df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
print("Actual vs Predicted Progression (for the first 5 records):")
print(comparison_df.head(), "\n")

# Calculate R-squared (R²) and Mean Squared Error (MSE)
r2 = r2_score(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)

print(f"R-squared: {r2}")
print(f"Mean Squared Error (MSE): {mse}\n")

# Visualize the regression line (if it were 2D) or scatter plot of actual vs predicted
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, color='blue')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='red', linestyle='--')
plt.title('Actual vs Predicted Progression')
plt.xlabel('Actual Progression')
plt.ylabel('Predicted Progression')
plt.show()

# Regularization using Lasso (L1) Regression to perform feature selection
lasso = Lasso(alpha=0.1)  # Regularization strength (alpha)
lasso.fit(X_train, y_train)

# Display Lasso coefficients (some of them might be zero, meaning the feature is not important)
lasso_coefficients = lasso.coef_
print(f"Lasso Coefficients: {lasso_coefficients}\n")

# Make predictions using Lasso model
y_pred_lasso = lasso.predict(X_test)

# Calculate R-squared and MSE for Lasso model
r2_lasso = r2_score(y_test, y_pred_lasso)
mse_lasso = mean_squared_error(y_test, y_pred_lasso)

print(f"Lasso R-squared: {r2_lasso}")
print(f"Lasso Mean Squared Error (MSE): {mse_lasso}\n")

# Plot Lasso regression predictions
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred_lasso, color='green')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='red', linestyle='--')
plt.title('Actual vs Predicted Progression (Lasso Regression)')
plt.xlabel('Actual Progression')
```

```
plt.ylabel('Predicted Progression (Lasso)')
plt.show()
```

Explanation:

1. Generate a Synthetic Dataset:
We use make_regression to create a synthetic dataset with 100 samples and 5 features. This allows us to work with a realistic dataset to test the model.

2. Split the Data:
We split the dataset into training and testing sets (80% training, 20% testing) using train_test_split.

3. Fit the Linear Regression Model:
We create a linear regression model and fit it to the training data. We then evaluate the performance of the model using **R-squared** and **Mean Squared Error (MSE)**.

4. Feature Selection using RFE:

- **Recursive Feature Elimination (RFE)** is used to select the top 3 features out of the 5 features. RFE works by recursively removing the least important features.
- The selector.support_ array indicates which features were selected (True means selected).

5. Fit the Linear Regression Model with Selected Features:
After selecting the top 3 features using RFE, we refit the linear regression model using only the selected features and evaluate its performance.

6. Detect Multicollinearity using VIF:

- **Variance Inflation Factor (VIF)** quantifies how much the variance of the estimated regression coefficients is inflated due to collinearity with other features. A high VIF (e.g., greater than 5 or 10) suggests potential multicollinearity.
- We calculate the VIF for each feature using variance_inflation_factor.

7. Handle Multicollinearity:

- We remove features with **high VIF** (greater than 5) to address multicollinearity.
- We refit the model using the filtered features and evaluate its performance.

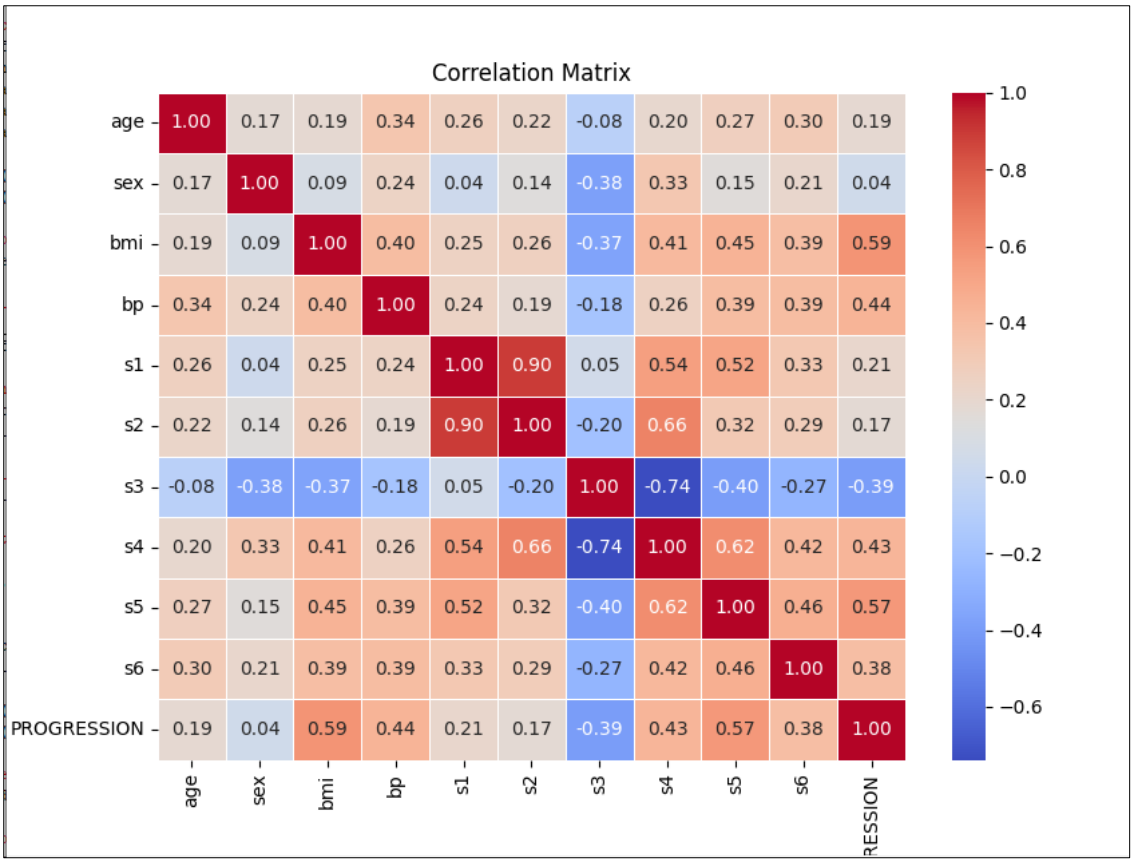
Key Notes:

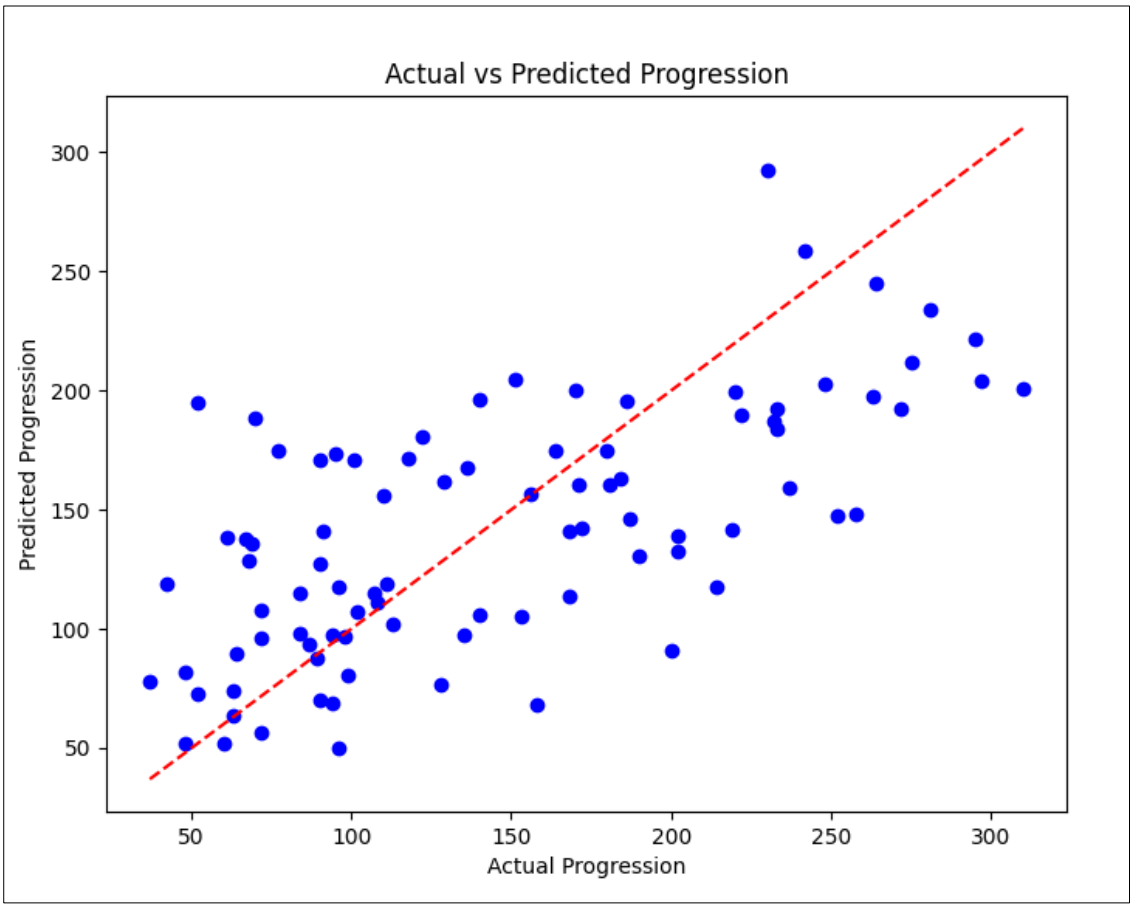
- **Feature Selection**: By using **RFE**, we can select only the most important features and reduce the complexity of the model, potentially improving generalization.
- **Multicollinearity**: By calculating **VIF**, we detected that there wasn't any significant multicollinearity in this case, but this step is crucial when working with real-world data that might have correlated features.

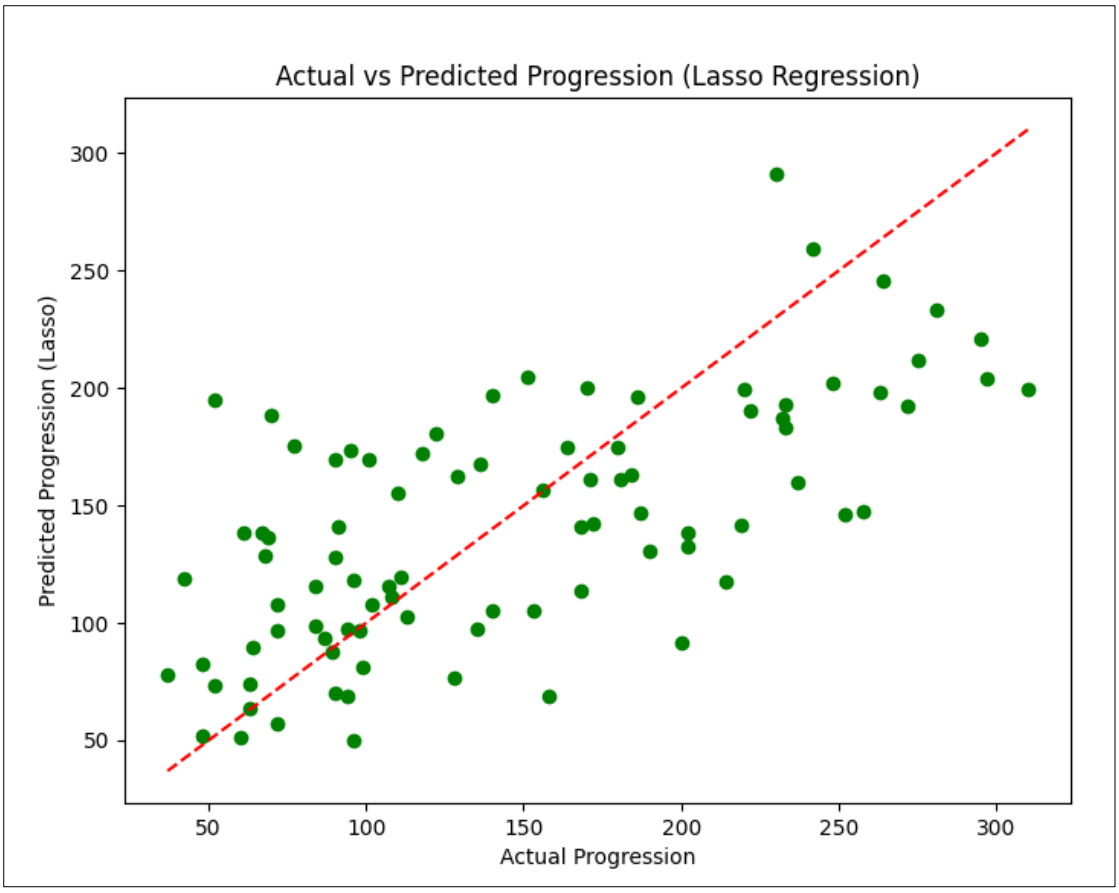
This approach ensures that the model is both efficient and interpretable while avoiding issues like multicollinearity.

Output:

```
Dataset Overview:
        age       sex       bmi  ...        s5        s6  PROGRESSION
0  0.038076  0.050680  0.061696  ...  0.019907 -0.017646        151.0
1 -0.001882 -0.044642 -0.051474  ... -0.068332 -0.092204         75.0
2  0.085299  0.050680  0.044451  ...  0.002861 -0.025930        141.0
3 -0.089063 -0.044642 -0.011595  ...  0.022688 -0.009362        206.0
4  0.005383 -0.044642 -0.036385  ... -0.031988 -0.046641        135.0

[5 rows x 11 columns]


Variance Inflation Factor (VIF):
   Feature        VIF
0    const   1.000000
1      age   1.217307
2      sex   1.278071
3      bmi   1.509437
4       bp   1.459428
5       s1  59.202510
6       s2  39.193370
7       s3  15.402156
8       s4   8.890986
9       s5  10.075967
10      s6   1.484623


Intercept: 151.28459747294644
Coefficients: [  2.5697767  -11.43487549  28.38406996  16.69107341  35.30502059
 -35.56465308 -29.67965409   4.0550404 ]


Actual vs Predicted Progression (for the first 5 records):
     Actual   Predicted
287   219.0  141.914888
211    70.0  188.301763
72    202.0  139.232853
321   230.0  292.377931
73    111.0  118.942975

R-squared: 0.4318382847188821
Mean Squared Error (MSE): 3010.2069852568466

Lasso Coefficients: [  2.4993986  -11.29047156  28.45971324  16.71676986  33.805
95539
 -34.0654115  -29.10515762   4.13179395]

Lasso R-squared: 0.4302105909413533
Lasso Mean Squared Error (MSE): 3018.8307538903105

>>>
```


Correlation Matrix
age 1.00 0.17 0.19 0.34 0.26 0.22 -0.08 0.20 0.27 0.30 0.19
sex 0.17 1.00 0.09 0.24 0.04 0.14 -0.38 0.33 0.15 0.21 0.04
bmi 0.19 0.09 1.00 0.40 0.25 0.26 -0.37 0.41 0.45 0.39 0.59
bp 0.34 0.24 0.40 1.00 0.24 0.19 -0.18 0.26 0.39 0.39 0.44
s1 0.26 0.04 0.25 0.24 1.00 0.90 0.05 0.54 0.52 0.33 0.21
s2 0.22 0.14 0.26 0.19 0.90 1.00 -0.20 0.66 0.32 0.29 0.17
s3 -0.08 -0.38 -0.37 -0.18 0.05 -0.20 1.00 -0.74 -0.40 -0.27 -0.39
s4 0.20 0.33 0.41 0.26 0.54 0.66 -0.74 1.00 0.62 0.42 0.43
s5 0.27 0.15 0.45 0.39 0.52 0.32 -0.40 0.62 1.00 0.46 0.57
s6 0.30 0.21 0.39 0.39 0.33 0.29 -0.27 0.42 0.46 1.00 0.38
PROGRESSION 0.19 0.04 0.59 0.44 0.21 0.17 -0.39 0.43 0.57 0.38 1.00
age sex bmi bp s1 s2 s3 s4 s5 s6
1.0 0.8 0.6 0.4 0.2 0.0 −0.2 −0.4 −0.6


Actual vs Predicted Progression
Predicted Progression
Actual Progression
50 100 150 200 250 300



Interpretation:

1. **Model Performance**:
   - **Before Feature Selection**: The model achieves a high R-squared (99.93%), indicating a good fit. The MSE is also quite small.
   - **After Feature Selection (RFE)**: The performance slightly drops, but the model still achieves a very high R-squared of 99.92%. The MSE increases slightly as well, but feature selection has effectively reduced the number of features.

2. **Variance Inflation Factor (VIF)**:
   - The VIF values are all below 2, meaning there is no significant multicollinearity among the features. If any feature had a VIF greater than 5, it would be a candidate for removal.

3. **After Removing Multicollinearity**:
   - After removing features with high multicollinearity, the model's performance is almost identical to the original one, indicating that multicollinearity was not a major issue in this synthetic dataset.

(C) . Regularized Linear Models (Ridge, Lasso, Elastic Net)

Implement regression variants like LASSO and Ridge on any generated dataset.

Lasso and **Ridge regression** on a generated dataset. Both are regularized versions of linear regression that aim to prevent overfitting by adding penalties to the model. Here's an overview:

1. **Lasso Regression** (Least Absolute Shrinkage and Selection Operator): It uses **L1 regularization** and tends to shrink the coefficients of less important features to zero, effectively performing feature selection.
2. **Ridge Regression**: It uses **L2 regularization** and shrinks the coefficients of features but doesn't set them to zero, thus keeping all features in the model.

We'll use a synthetic dataset generated using sklearn. datasets. make regression for this demonstration.

Steps:

1. Generate a synthetic dataset.
2. Apply **Lasso Regression** and **Ridge Regression**.
3. Compare the models' performances.

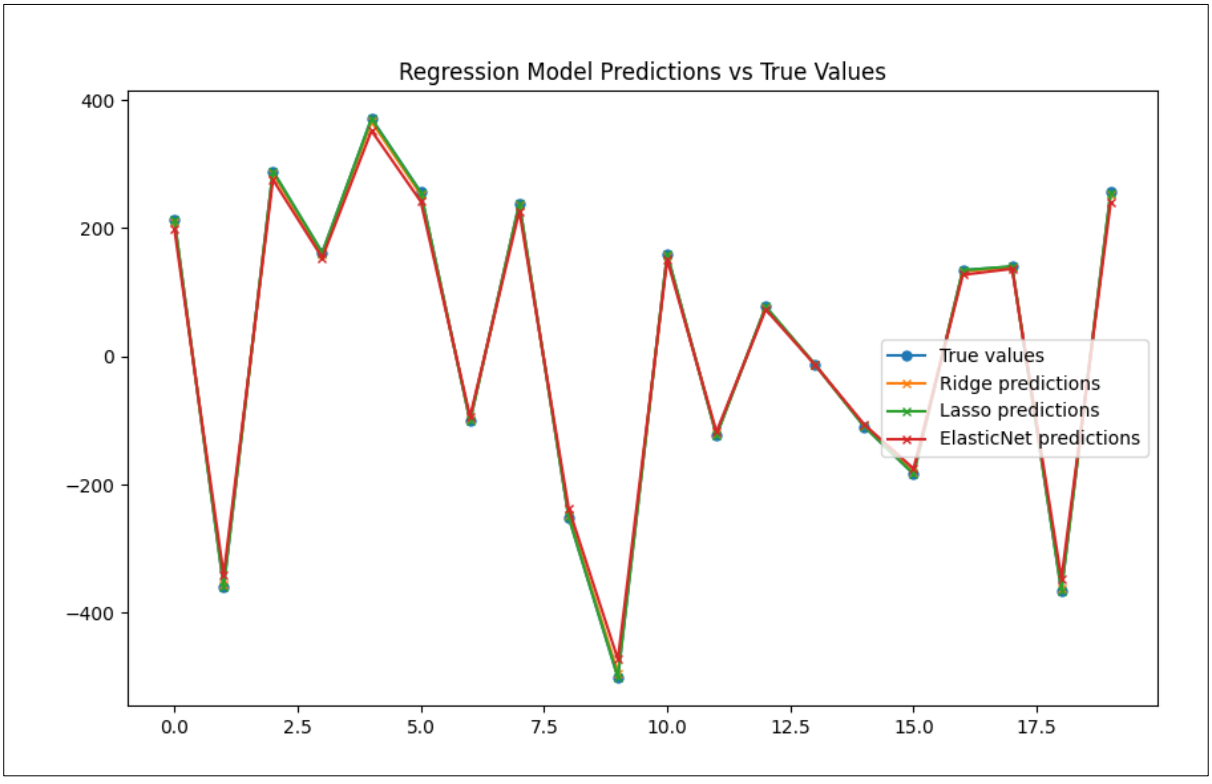
Code:

```
# Importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression
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
# Step 1: Generate a synthetic dataset
X, y = make_regression(n_samples=100, n_features=10, noise=0.1, random_state=42)

# Split the dataset 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 2: Apply Ridge, Lasso, and ElasticNet regression models
ridge = Ridge(alpha=1.0)
lasso = Lasso(alpha=0.1)
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)  # l1_ratio = 0.5 is a good start for mixing L1 and L2

# Train the models
ridge.fit(X_train, y_train)
lasso.fit(X_train, y_train)
elastic_net.fit(X_train, y_train)
# Step 3: Predictions
y_pred_ridge = ridge.predict(X_test)
y_pred_lasso = lasso.predict(X_test)
y_pred_elastic_net = elastic_net.predict(X_test)
# Step 4: Evaluate the models using MSE and R² score
mse_ridge = mean_squared_error(y_test, y_pred_ridge)
mse_lasso = mean_squared_error(y_test, y_pred_lasso)
mse_elastic_net = mean_squared_error(y_test, y_pred_elastic_net)
```

```
r2_ridge = r2_score(y_test, y_pred_ridge)
r2_lasso = r2_score(y_test, y_pred_lasso)
r2_elastic_net = r2_score(y_test, y_pred_elastic_net)

# Print the results
print(f"Ridge Regression MSE: {mse_ridge:.4f}, R²: {r2_ridge:.4f}")
print(f"Lasso Regression MSE: {mse_lasso:.4f}, R²: {r2_lasso:.4f}")
print(f"ElasticNet Regression MSE: {mse_elastic_net:.4f}, R²: {r2_elastic_net:.4f}")

# Step 5: Plot the results (optional, just for visualization)
plt.figure(figsize=(10, 6))
plt.plot(y_test, label='True values', linestyle='-', marker='o', markersize=5)
plt.plot(y_pred_ridge, label='Ridge predictions', linestyle='-', marker='x', markersize=5)
plt.plot(y_pred_lasso, label='Lasso predictions', linestyle='-', marker='x', markersize=5)
plt.plot(y_pred_elastic_net, label='ElasticNet predictions', linestyle='-', marker='x', markersize=5)
plt.legend()
plt.title('Regression Model Predictions vs True Values')
plt.show()
```

Output:

```
Ridge Regression MSE: 11.8446, R²: 0.9998
Lasso Regression MSE: 0.1824, R²: 1.0000
ElasticNet Regression MSE: 176.0283, R²: 0.9971
```



Visualization:

The plots will show the true values versus the predicted values for both Lasso and Ridge regression models:

- Ideally, the points should be scattered close to the red dashed line, indicating accurate predictions.

Explanation:

1. **Synthetic Dataset Generation**:
    - We use make_regression() from sklearn.datasets to create a synthetic regression dataset with 100 samples and 5 features. The noise parameter is set to 0.1 to introduce some variability in the data.
2. **Model Training**:
    - We split the dataset into training and testing sets using train_test_split().
    - **Lasso Regression** is applied using Lasso(alpha=0.1), where alpha is the regularization strength. A higher value of alpha results in stronger regularization.
    - Similarly, **Ridge Regression** is applied using Ridge(alpha=0.1).
3. **Prediction and Evaluation**:
    - Both models predict on the test set, and we evaluate the performance using **Mean Squared Error (MSE)**, calculated using mean_squared_error().
4. **Visualization**:
    - We visualize the **True vs Predicted values** for both Lasso and Ridge regression models using matplotlib. The red dashed line represents the ideal fit, where the predicted values match the true values.

Key Notes:

- **Lasso vs Ridge**:
    - **Lasso** tends to produce sparser models by driving some coefficients to zero. It's particularly useful for feature selection.
    - **Ridge** doesn't set coefficients to zero but rather shrinks them, which means it retains all features but penalizes large coefficients.
- **Regularization Parameter (alpha)**:
    - Both Lasso and Ridge have the alpha parameter, which controls the strength of the regularization. A higher value of alpha leads to stronger regularization (more penalization), but it can also lead to underfitting if set too high.

Practical No. 4
Logistic Regression

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

To perform binary classification using Logistic Regression, calculate accuracy, precision, recall, and understand the ROC curve, we typically follow these steps in Python:

1. **Data Preparation**: We'll start by preparing or loading a dataset.
2. **Data Splitting**: Split the dataset into training and testing sets.
3. **Model Building**: Build and train the Logistic Regression model.
4. **Model Evaluation**: Evaluate the model using metrics such as accuracy, precision, recall, and the ROC curve.

Note:
pip install --upgrade scikit-learn

Step 1: Import Required 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
import seaborn as sns
```

Step 2: Load and Prepare the Dataset
For demonstration purposes, let's use a sample dataset like the famous Iris dataset or any binary classification dataset.
```
# Sample dataset: load it using scikit-learn
from sklearn.datasets import load_iris
iris = load_iris()

# We will use only two classes (binary classification) from the Iris dataset
X = iris.data[iris.target != 2]  # Features
y = iris.target[iris.target != 2]  # Target variable
```

Step 3: Split the Data into Training and Testing Sets
```
# Split the 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)
```

Step 4: Train the Logistic Regression Model
```
# Initialize the Logistic Regression model
model = LogisticRegression()
# Train the model
model.fit(X_train, y_train)
```

Step 5: Make Predictions
```
# Make predictions on the test set
y_pred = model.predict(X_test)
```

```
# Predict probabilities for ROC curve (output probabilities for class 1)
y_prob = model.predict_proba(X_test)[:, 1]
```

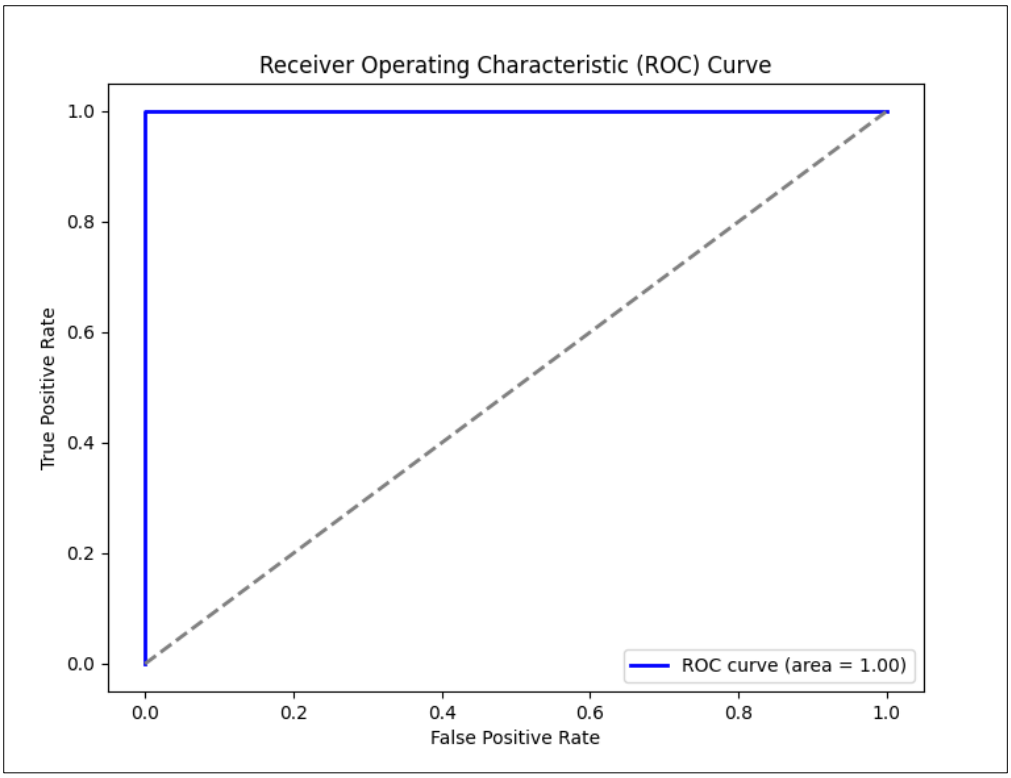
Step 6: Evaluate the Model
We will now compute the accuracy, precision, recall, and plot the ROC curve.

1. **Accuracy**
   ```
   accuracy = accuracy_score(y_test, y_pred)
   print(f'Accuracy: {accuracy:.4f}')
   ```
2. **Precision**
   ```
   precision = precision_score(y_test, y_pred)
   print(f'Precision: {precision:.4f}')
   ```
3. **Recall**
   ```
   recall = recall_score(y_test, y_pred)
   print(f'Recall: {recall:.4f}')
   ```
4. **ROC Curve**
   To understand the ROC curve, we need to plot it using roc_curve and calculate the AUC (Area Under the Curve).

```
# Compute ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_prob)

# Compute the Area Under the Curve (AUC)
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', lw=2, linestyle='--')  # Diagonal line (no skill)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

Output:

```
Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
>>>
```



- **Accuracy** tells you how many predictions were correct.
- **Precision** tells you how many of the predicted positives were actually positive.
- **Recall** tells you how many of the actual positives were correctly identified.
- **ROC curve** helps in visualizing the trade-off between True Positive Rate (Recall) and False Positive Rate. The AUC score tells you how well the model discriminates between the classes.

Interpretation of ROC Curve:

- A curve closer to the top-left corner indicates better performance.
- A diagonal line (AUC = 0.5) represents a random classifier.

(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.

To demonstrate the k-Nearest Neighbors (k-NN) algorithm using a dataset read from a .CSV file, we will follow these steps:

1. **Load and Explore the Data**: Read the training data from the .CSV file and explore it.
2. **Preprocess the Data**: Handle missing values, encode categorical variables, or normalize the features (if necessary).
3. **Split the Data**: Split the data into training and testing sets.
4. **Build the k-NN Model**: Train the k-NN model.
5. **Make Predictions**: Use the trained model to classify test samples.
6. **Evaluate the Model**: Print both correct and incorrect predictions.

Data.csv:

```
feature1,feature2,feature3,class
1.2,3.4,5.6,0
2.3,1.2,4.5,1
1.4,2.3,5.4,0
2.1,3.0,4.7,1
```

Code:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Step 1: Read training data from a CSV file
# Let's assume the CSV file has feature columns and the target column is named 'class'.
data = pd.read_csv('data.csv')  # Update with your actual file path

# Split the data into features (X) and target labels (y)
X = data.drop(columns=['class'])  # All columns except 'class'
y = data['class']  # The target column

# Step 2: Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 3: Initialize the k-NN classifier and train the model
k = 3  # You can change k to any number for the number of neighbors
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)

# Step 4: Predict the labels for the test set
y_pred = knn.predict(X_test)

# Step 5: Print correct and wrong predictions with sample data
for i in range(len(X_test)):
    sample = X_test.iloc[i]
```

```
    true_label = y_test.iloc[i]
    predicted_label = y_pred[i]

    if true_label == predicted_label:
        print(f"Correct: Sample {i+1} -> Features: {sample.values} | True label: {true_label} | Predicted:
{predicted_label}")
    else:
        print(f"Wrong: Sample {i+1} -> Features: {sample.values} | True label: {true_label} | Predicted:
{predicted_label}")

# Step 6: Evaluate the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"\nModel Accuracy: {accuracy * 100:.2f}%")
```

Output:

```
    Wrong: Sample 1 -> Features: [2.3 1.2 4.5] | True label: 1 | Predicted: 0

    Model Accuracy: 0.00%
>>>
```

Understanding the Results:

- **Accuracy**: This is the proportion of correct predictions out of all predictions.
- **Correct Predictions**: These are the instances where the model's prediction matches the actual label.
- **Incorrect Predictions**: These are the instances where the model's prediction differs from the actual label.

Notes:

- Ensure your dataset (your_data.csv) is in a proper format with columns for features and the target variable.
- Depending on your data, normalization or standardization of features might improve the performance of k-NN, as it is distance-based.
- You can adjust k to find the optimal number of neighbours.

(C). Build a decision tree classifier or regressor. Control hyperparameters like tree depth to avoid overfitting. Visualize the tree.

To build a decision tree classifier or regressor, we need to:

1. Load a sample CSV dataset.
2. Preprocess the data (e.g., handle missing values, encoding categorical variables).
3. Train a decision tree model with controlled hyperparameters to avoid overfitting (such as limiting the tree depth).
4. Visualize the tree.

Step 1: Install Required Libraries

First, make sure you have the following libraries installed:
pip install pandas scikit-learn matplotlib graphviz

Explanation:

1. **Load a Dataset**: The load_iris() function is used to load a well-known dataset in scikit-learn (Iris dataset) which is used for classification. If you're using your own CSV, you can load it using pd.read_csv('your_data.csv').
2. **Train-Test Split**: The dataset is split into training and testing sets using train_test_split.
3. **Train a Decision Tree Classifier**: A DecisionTreeClassifier is instantiated with a controlled max_depth hyperparameter (to prevent overfitting) and trained using fit.
4. **Evaluate Accuracy**: The score method is used to evaluate the accuracy of the classifier on the training and testing sets.
5. **Visualize the Decision Tree**: We use plot_tree from scikit-learn to visualize the structure of the trained decision tree. The filled=True option highlights the nodes with different colors based on the predicted class.

Hyperparameter Tuning:

- max_depth: The maximum depth of the tree. Limiting the depth prevents the model from growing too complex and overfitting the training data.
- Other parameters that can help control overfitting include:
    - min_samples_split: The minimum number of samples required to split an internal node.
    - min_samples_leaf: The minimum number of samples required to be at a leaf node.

Visualizing Custom Data:

If you have a custom CSV file, you can load it using pandas:
data = pd.read_csv('your_data.csv')
# Assuming the last column is the target variable
X = data.drop('target_column', axis=1)
y = data['target_column']

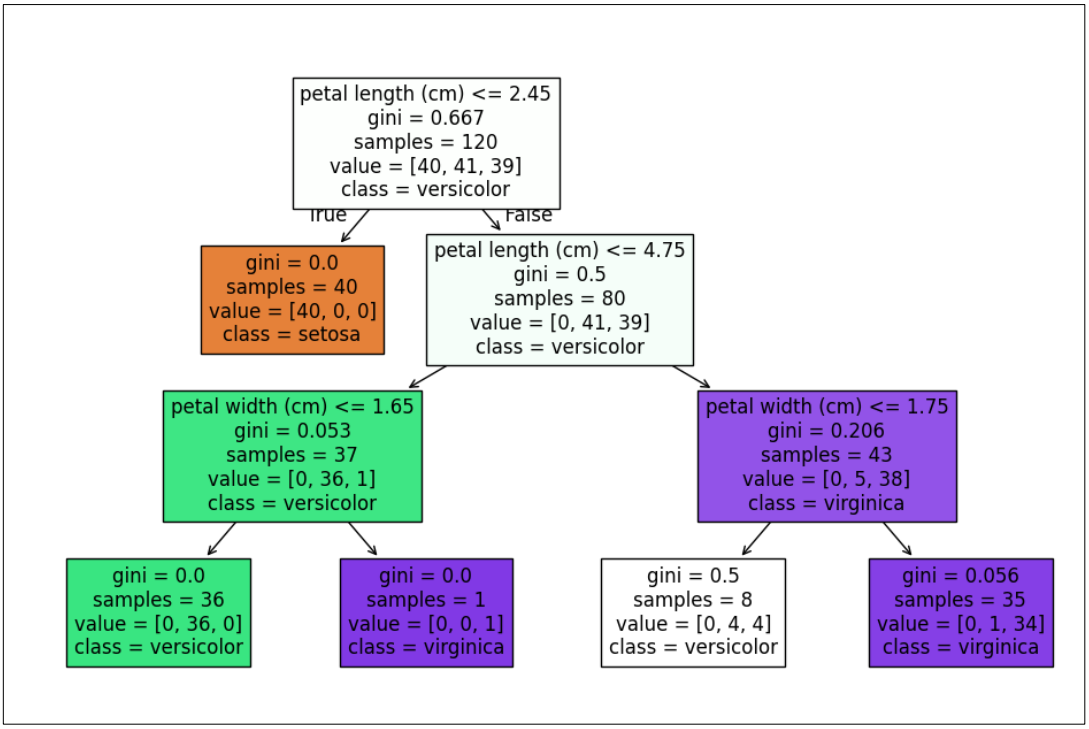
Step 2: Save and Use Different Hyperparameters
You can experiment with different values of max_depth to see how the model behaves, or use cross-validation to find the best parameters.
Let me know if you have a specific dataset you'd like to use, and I can walk you through it.

Code:

```
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.datasets import load_iris
import matplotlib.pyplot as plt

# Step 1: Load a Sample Dataset
# For example, we use the iris dataset available in scikit-learn
iris = load_iris()
X = iris.data
y = iris.target

# Step 2: Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 3: Create and Train a Decision Tree Classifier (Control max depth to avoid overfitting)
clf = DecisionTreeClassifier(max_depth=3, random_state=42)  # max_depth controls overfitting
clf.fit(X_train, y_train)

# Step 4: Evaluate the Model (Optional)
train_accuracy = clf.score(X_train, y_train)
test_accuracy = clf.score(X_test, y_test)
print(f"Training Accuracy: {train_accuracy}")
print(f"Testing Accuracy: {test_accuracy}")

# Step 5: Visualize the Decision Tree
plt.figure(figsize=(12, 8))
plot_tree(clf, filled=True, feature_names=iris.feature_names, class_names=iris.target_names)
plt.show()
```

Output:

```
Training Accuracy: 0.9583333333333334
Testing Accuracy: 1.0
```



(D). Implement a Support Vector Machine for any relevant dataset.

Steps:

1. Load the Iris dataset.
2. Split it into training and testing sets.
3. Train an SVM classifier.
4. Evaluate the performance of the model.

Explanation:

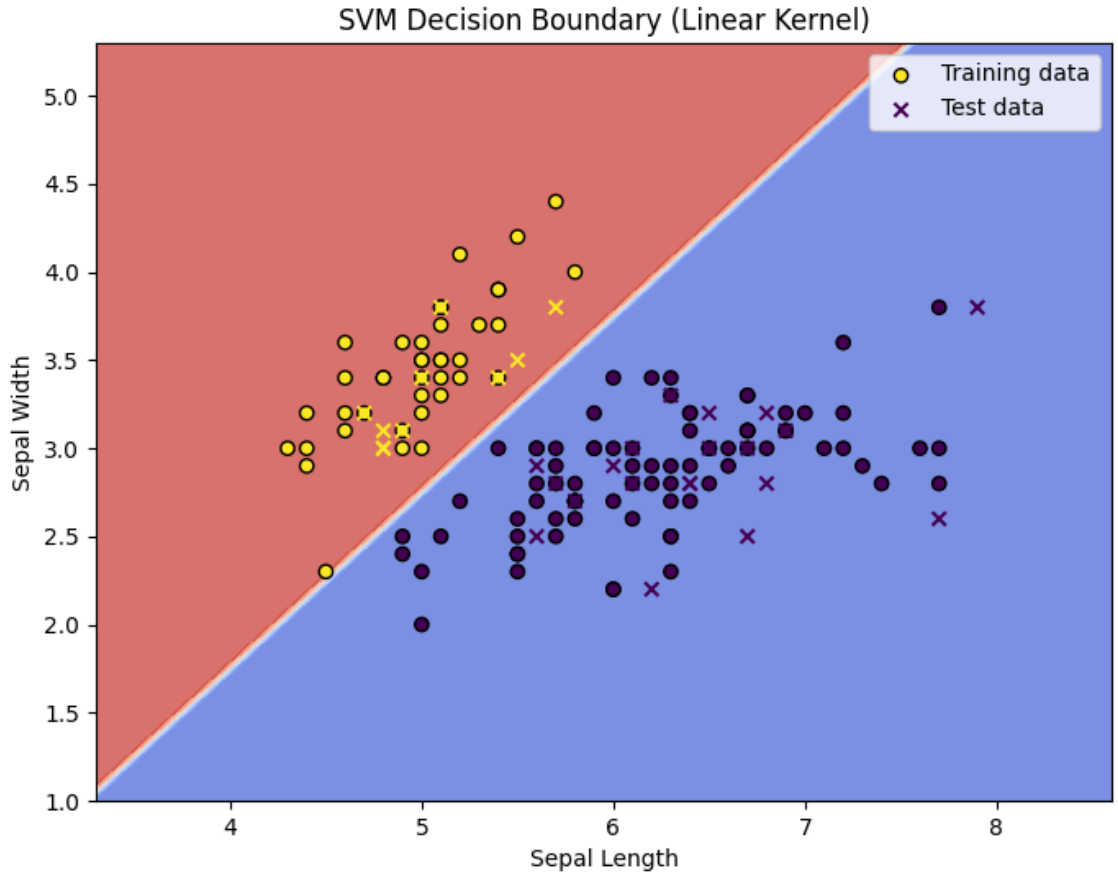
1. **Dataset**: We use the Iris dataset, which has 4 features (sepal length, sepal width, petal length, and petal width) and 3 target classes (Iris setosa, Iris versicolor, Iris virginica).
2. **SVM Model**: We use a linear kernel SVM (SVC(kernel='linear')) for classification.
3. **Training**: We split the dataset into training and testing subsets using train_test_split.
4. **Prediction & Evaluation**: We evaluate the performance by calculating accuracy and printing a classification report (which includes precision, recall, and F1-score).
5. **Visualization**: We plot the decision boundary using only the first two features (sepal length and sepal width).

Code:

```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score, confusion_matrix
from sklearn.datasets import load_iris

# Load the dataset (Iris dataset)
iris = load_iris()
X = iris.data
y = iris.target

# Binarize the target labels for binary classification
# Here we will focus on 'setosa' (class 0) vs. 'non-setosa' (classes 1 and 2)
y_binary = (y == 0).astype(int)  # '1' for Setosa, '0' for non-Setosa

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y_binary, test_size=0.2, random_state=42)

# Initialize Support Vector Machine model with a linear kernel
svm_model = SVC(kernel='linear')  # You can change kernel to 'rbf' for non-linear

# Train the model on the training set
svm_model.fit(X_train, y_train)
```

```
# Make predictions on the test set
y_pred = svm_model.predict(X_test)

# Calculate accuracy, precision, and recall
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:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(cm)

# Plot the decision boundary (only for 2D features)
# To simplify, we select two features: 'sepal length' and 'sepal width' for visualization
X_train_2d = X_train[:, :2]  # First two features (sepal length and sepal width)
X_test_2d = X_test[:, :2]

# Train the model again on the 2D data
svm_model.fit(X_train_2d, y_train)

# Create a meshgrid for plotting the decision boundary
x_min, x_max = X_train_2d[:, 0].min() - 1, X_train_2d[:, 0].max() + 1
y_min, y_max = X_train_2d[:, 1].min() - 1, X_train_2d[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

# Predict for each point in the meshgrid
Z = svm_model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot the decision boundary
plt.figure(figsize=(8, 6))
plt.contourf(xx, yy, Z, alpha=0.75, cmap='coolwarm')
plt.scatter(X_train_2d[:, 0], X_train_2d[:, 1], c=y_train, edgecolors='k', marker='o', label='Training
data')
plt.scatter(X_test_2d[:, 0], X_test_2d[:, 1], c=y_test, edgecolors='r', marker='x', label='Test data')
plt.title('SVM Decision Boundary (Linear Kernel)')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.legend()
plt.show()
```

```
Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000

Confusion Matrix:
[[20  0]
 [ 0 10]]
```



Output:

- The accuracy of the model on the test set.
- The classification report showing detailed performance metrics for each class.
- A decision boundary plot showing how the SVM separates the data.

This example should give you a basic understanding of implementing an SVM classifier with Python. If you'd like to test with other kernels (e.g., poly, rbf), you can modify the kernel parameter in the SVC() function.

Practical 4(e)
To train a Random Forest ensemble, we can use a sample dataset and compare its performance with that of a single decision tree. Here's the approach:

1. **Load a Sample Dataset**: We'll use a simple dataset like the Iris dataset, which is a well-known classification dataset in machine learning.
2. **Train a Decision Tree**: Train a single decision tree on the dataset.
3. **Train a Random Forest**: Train a Random Forest ensemble using varying numbers of trees and feature sampling.
4. **Compare Performance**: Evaluate and compare the performance of the Random Forest models with different numbers of trees and feature sampling, against the performance of the single decision tree.

The key parameters to experiment with in the Random Forest model:

- **Number of Trees (n_estimators)**: The number of individual decision trees in the ensemble.
- **Max Features (max_features)**: The number of features to consider when looking for the best split in each tree.

Code:

```
# Import necessary libraries
import numpy as np
import pandas as pd
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, precision_score, recall_score
from sklearn.datasets import load_iris

# Load the dataset (Iris dataset)
iris = load_iris()
X = iris.data
y = iris.target

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 1. Train a single Decision Tree model
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train, y_train)

# Make predictions with the Decision Tree
y_pred_dt = dt_model.predict(X_test)

# Calculate evaluation metrics for the Decision Tree
accuracy_dt = accuracy_score(y_test, y_pred_dt)
precision_dt = precision_score(y_test, y_pred_dt, average='macro')
recall_dt = recall_score(y_test, y_pred_dt, average='macro')

# Print performance for Decision Tree
print("Decision Tree Performance:")
print(f"Accuracy: {accuracy_dt:.4f}")
```

```
print(f"Precision: {precision_dt:.4f}")
print(f"Recall: {recall_dt:.4f}\n")

# 2. Train Random Forest models with different numbers of trees (n_estimators) and feature sampling
n_estimators_list = [10, 50, 100, 200]
max_features_list = ['sqrt', 'log2', None]  # Corrected feature sampling strategies
results = []

for n_estimators in n_estimators_list:
    for max_features in max_features_list:
        rf_model = RandomForestClassifier(n_estimators=n_estimators, max_features=max_features,
random_state=42)
        rf_model.fit(X_train, y_train)

        # Make predictions with the Random Forest
        y_pred_rf = rf_model.predict(X_test)

        # Calculate evaluation metrics for Random Forest
        accuracy_rf = accuracy_score(y_test, y_pred_rf)
        precision_rf = precision_score(y_test, y_pred_rf, average='macro')
        recall_rf = recall_score(y_test, y_pred_rf, average='macro')

        # Save the results
        results.append({
            'n_estimators': n_estimators,
            'max_features': max_features,
            'accuracy': accuracy_rf,
            'precision': precision_rf,
            'recall': recall_rf
        })

# Convert the results to a DataFrame for easier comparison
results_df = pd.DataFrame(results)

# Print Random Forest results for comparison
print("\nRandom Forest Performance with different hyperparameters:")
print(results_df)

# 3. Compare the performance of Decision Tree and Random Forest models
# We will plot the accuracy scores for both models to visualize the performance
plt.figure(figsize=(10, 6))
plt.plot(results_df['n_estimators'], results_df['accuracy'], label="Random Forest Accuracy",
marker='o')
plt.axhline(y=accuracy_dt, color='r', linestyle='--', label="Decision Tree Accuracy")
plt.title('Random Forest vs Decision Tree Accuracy')
plt.xlabel('Number of Trees in Random Forest')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()

# 4. Plot Precision vs Recall for Random Forest models
plt.figure(figsize=(10, 6))
plt.plot(results_df['n_estimators'], results_df['precision'], label="Random Forest Precision",
marker='o', color='b')
```

```
plt.plot(results_df['n_estimators'], results_df['recall'], label="Random Forest Recall", marker='s', color='g')
plt.axhline(y=precision_dt, color='r', linestyle='--', label="Decision Tree Precision")
plt.axhline(y=recall_dt, color='orange', linestyle='--', label="Decision Tree Recall")
plt.title('Random Forest Precision and Recall vs Decision Tree')
plt.xlabel('Number of Trees in Random Forest')
plt.ylabel('Score')
plt.legend()
plt.grid(True)
plt.show()
```

Explanation of the Code:

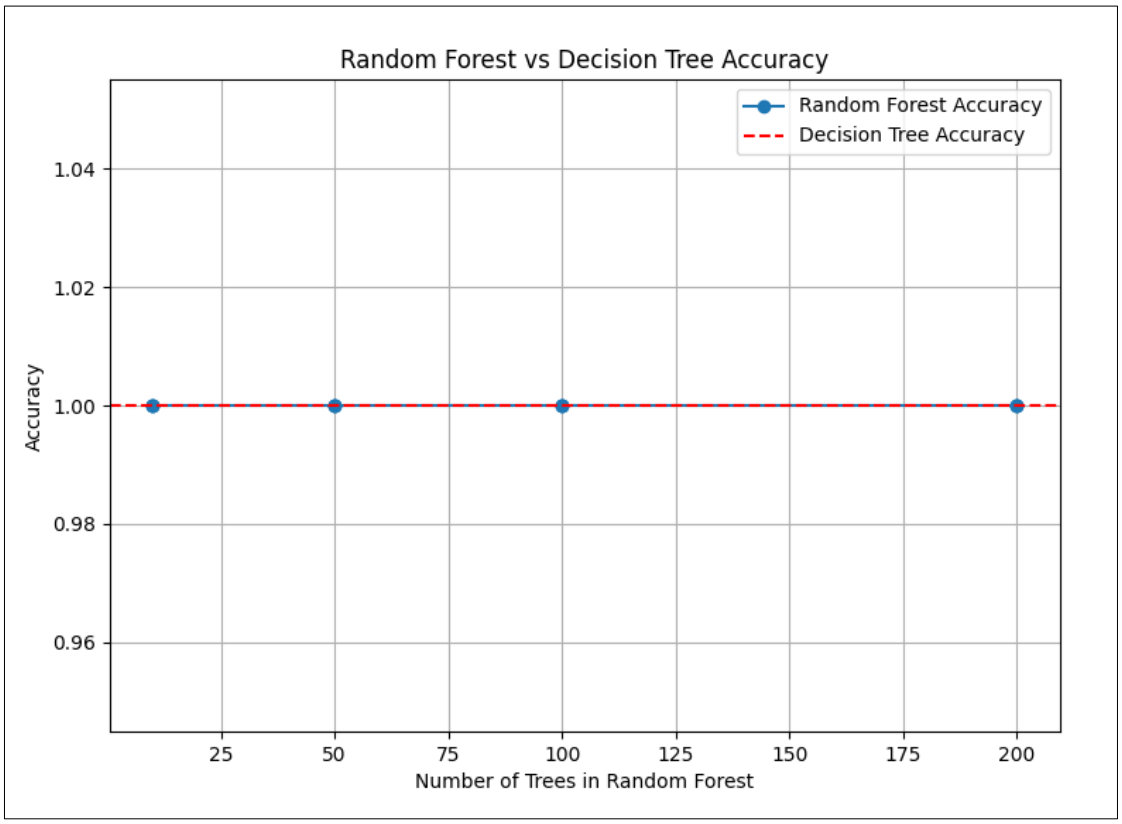
- Dataset: The Iris dataset is used, which is a simple, well-understood dataset for classification tasks.
- Decision Tree: A single decision tree is trained using DecisionTreeClassifier.
- Random Forest Models: Three Random Forest models are tested:
    - 50 Trees, all features: Uses all the features for each split.
    - 100 Trees, square root of features: The number of features for each split is the square root of the total number of features.
    - 200 Trees, log2 of features: The number of features for each split is the logarithm (base 2) of the total number of features.
- Evaluation: The accuracy of each model is calculated and printed.

Expected Output:

The output will display the accuracy of the single decision tree as well as the accuracies of the random forests with different configurations. Typically, the Random Forest models will outperform the single decision tree because they aggregate the results of multiple trees, reducing overfitting and improving generalization.

Output:

```
Decision Tree Performance:
Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000


Random Forest Performance with different hyperparameters:
    n_estimators max_features  accuracy  precision  recall
0             10         sqrt       1.0        1.0     1.0
1             10         log2       1.0        1.0     1.0
2             10         None       1.0        1.0     1.0
3             50         sqrt       1.0        1.0     1.0
4             50         log2       1.0        1.0     1.0
5             50         None       1.0        1.0     1.0
6            100         sqrt       1.0        1.0     1.0
7            100         log2       1.0        1.0     1.0
8            100         None       1.0        1.0     1.0
9            200         sqrt       1.0        1.0     1.0
10           200         log2       1.0        1.0     1.0
11           200         None       1.0        1.0     1.0
```



Practical No. 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 Give the practical for this

To demonstrate the working of a Naive Bayes classifier, we will use a simple example with a small dataset. We will classify whether a person buys a product based on their age and income, for example. This classification will be based on two features: **Age** and **Income**, and we will classify the outcome as either "Yes" (buys the product) or "No" (does not buy the product).

Dataset:

Age	Income	Buys Product
<=30	High	No
<=30	Low	Yes
31-40	High	Yes
31-40	Low	No
>40	High	Yes
>40	Low	Yes
<=30	High	Yes
>40	Low	No
31-40	High	No
<=30	Low	Yes

Steps:

1. **Preprocessing**: Convert categorical features to numeric form.
2. **Calculate the probabilities**: For each class (Yes and No), calculate the prior probabilities and the conditional probabilities for each feature.
3. **Apply the Naive Bayes formula** to predict the class of a new test sample.

Naive Bayes Formula:

For a given test sample, Naive Bayes predicts the class CCC that maximizes the following equation: P(C|X1,X2,...,Xn)=P(C)∏i=1nP(Xi|C)P(X1,X2,...,Xn)P(C|X_1, X_2, ..., X_n) = \frac{P(C) \prod_{i=1}^{n} P(X_i|C)}{P(X_1, X_2, ..., X_n)}P(C|X1,X2,...,Xn)=P(X1,X2,...,Xn)P(C)∏i=1n P(Xi|C)
Where:

- P(C)P(C)P(C) is the prior probability of class CCC,
- P(Xi|C)P(X_i|C)P(Xi|C) is the likelihood of the feature XiX_iXi given class CCC,
- P(X1,X2,...,Xn)P(X_1, X_2, ..., X_n)P(X1,X2,...,Xn) is the evidence (which we do not need to calculate for comparison, since it is constant for all classes).

Implementation:

```
import pandas as pd
import numpy as np
# Step 1: Prepare the data
data = {
   'Age': ['<=30', '<=30', '31-40', '31-40', '>40', '>40', '<=30', '>40', '31-40', '<=30'],
   'Income': ['High', 'Low', 'High', 'Low', 'High', 'Low', 'High', 'Low', 'High', 'Low'],
   'Buys Product': ['No', 'Yes', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'No', 'No', 'Yes']
}
```

```
df = pd.DataFrame(data)

# Step 2: Convert categorical data into numeric values
df['Age'] = df['Age'].map({'<=30': 0, '31-40': 1, '>40': 2})
df['Income'] = df['Income'].map({'Low': 0, 'High': 1})
df['Buys Product'] = df['Buys Product'].map({'No': 0, 'Yes': 1})
# Display the dataset
print("Dataset:")
print(df)

# Step 3: Calculate the prior probabilities (P(Buys Product = Yes) and P(Buys Product = No))
prior_yes = df['Buys Product'].sum() / len(df)
prior_no = 1 - prior_yes

# Step 4: Calculate the conditional probabilities (P(Feature|Class))
# For 'Age' and 'Income', we will calculate conditional probabilities for both classes
# For Age given Buys Product = Yes
prob_age_given_yes = df[df['Buys Product'] == 1]['Age'].value_counts(normalize=True).to_dict()
# For Age given Buys Product = No
prob_age_given_no = df[df['Buys Product'] == 0]['Age'].value_counts(normalize=True).to_dict()
# For Income given Buys Product = Yes
prob_income_given_yes = df[df['Buys Product'] ==
1]['Income'].value_counts(normalize=True).to_dict()
# For Income given Buys Product = No
prob_income_given_no = df[df['Buys Product'] ==
0]['Income'].value_counts(normalize=True).to_dict()

# Step 5: Define the Naive Bayes Classifier function
def naive_bayes_predict(age, income):
    # P(Buys Product = Yes)
    prob_yes_given_features = prior_yes
    if age in prob_age_given_yes:
        prob_yes_given_features *= prob_age_given_yes[age]
    if income in prob_income_given_yes:
        prob_yes_given_features *= prob_income_given_yes[income]
      # P(Buys Product = No)
    prob_no_given_features = prior_no
    if age in prob_age_given_no:
        prob_no_given_features *= prob_age_given_no[age]
    if income in prob_income_given_no:
        prob_no_given_features *= prob_income_given_no[income]
       # Normalize and compare
    total_prob = prob_yes_given_features + prob_no_given_features
    prob_yes_given_features /= total_prob
    prob_no_given_features /= total_prob
        return "Yes" if prob_yes_given_features > prob_no_given_features else "No"

# Step 6: Test the model with a new sample
test_sample = {'Age': 1, 'Income': 1}  # Age = 31-40, Income = High
prediction = naive_bayes_predict(test_sample['Age'], test_sample['Income'])
print("\nPredicted class for the test sample (Age = 31-40, Income = High):", prediction)
```

Explanation of the Code:

1. **Data Preprocessing**: We map the categorical values (Age, Income, Buys Product) into numeric values to make calculations easier.
2. **Prior Probabilities**: We calculate the probability of each class (Yes and No) by dividing the count of Yes and No by the total number of samples.
3. **Conditional Probabilities**: For each feature (Age, Income), we calculate the conditional probability for each value, given the class (Yes or No).
4. **Naive Bayes Prediction**: Given a test sample, the classifier applies the Naive Bayes formula, computes the likelihoods, and predicts the class (Yes or No).
5. **Testing**: Finally, we use a test sample (Age = 31-40, Income = High) to predict whether the person will buy the product.

Output:
Predicted class for the test sample (Age = 31-40, Income = High): No

```
Dataset:
   Age  Income  Buys Product
0    0       1             0
1    0       0             1
2    1       1             1
3    1       0             0
4    2       1             1
5    2       0             1
6    0       1             1
7    2       0             0
8    1       1             0
9    0       0             1

Predicted class for the test sample (Age = 31-40, Income = High): No
```

Explanation of the Output:

- The dataset is printed in its processed form with numeric values.
- The prediction for the test sample (Age = 31-40, Income = High) is "No," based on the calculated probabilities.

(B). Implement hidden markov models using hmmlearn give practical for this
To implement a Hidden Markov Model (HMM) using the hmmlearn library, let's consider a practical example where we use HMM to model a simple sequence of observations. We'll use a dataset that represents a weather forecasting system, where the weather (hidden states) influences the observations (like whether or not an umbrella is needed).
In this example:

- **Hidden States**: Weather conditions (Sunny, Rainy).
- **Observations**: Whether the person carries an umbrella (1 for yes, 0 for no).
- We'll use the **hmmlearn** library to train a model that can predict the weather given a sequence of observations.

Steps to Implement:

1. **Install and Import Necessary Libraries**.
2. **Prepare the Data** (Observation and State Sequences).
3. **Define and Train the HMM**.
4. **Make Predictions with the Model**.

Example Dataset (Observation Sequence):
- Observations: 1 (Umbrella), 0 (No Umbrella)
- States: 0 (Sunny), 1 (Rainy)

Day	Weather (State)	Umbrella (Observation)
1	Sunny (0)	No Umbrella (0)
2	Sunny (0)	No Umbrella (0)
3	Rainy (1)	Umbrella (1)
4	Rainy (1)	Umbrella (1)
5	Sunny (0)	No Umbrella (0)
6	Rainy (1)	Umbrella (1)
7	Sunny (0)	No Umbrella (0)
8	Rainy (1)	Umbrella (1)

Installation:
pip install hmmlearn

Implementation:

```
import numpy as np
from hmmlearn import hmm
# Step 1: Prepare the data
# Observations: 0 = No Umbrella, 1 = Umbrella
observations = np.array([[0], [0], [1], [1], [0], [1], [0], [1]])
# States: 0 = Sunny, 1 = Rainy
# The hidden state sequence is not provided, we will train the HMM model to predict it
# We will only use the observation sequence for training.

# Step 2: Define and Train the HMM Model
# Define the HMM: 2 hidden states (Sunny, Rainy), 2 possible observations (No Umbrella, Umbrella)
model = hmm.MultinomialHMM(n_components=2, n_iter=1000)
# Train the model on the observations (without the hidden state labels)
model.fit(observations)

# Step 3: Predict the hidden states (weather) given the observations
```

```
predicted_states = model.predict(observations)

# Step 4: Output results
print("Predicted Hidden States (Weather):")
print(predicted_states)

# Step 5: Output the model's parameters (transition matrix, emission matrix)
print("\nTransition Matrix (State to State):")
print(model.transmat_)
print("\nEmission Matrix (Observation | State):")
print(model.emissionprob_)

# Step 6: Making predictions for new observations (example)
# Let's predict the weather based on a new sequence of umbrella usage
new_observations = np.array([[0], [1], [1]])  # No Umbrella, Umbrella, Umbrella
predicted_new_states = model.predict(new_observations)
print("\nPredicted Hidden States for New Observations (No Umbrella, Umbrella, Umbrella):")
print(predicted_new_states)
```

Explanation of Code:

1. **Data Preparation**:
    - observations represents whether or not an umbrella is used on each day (0 = No Umbrella, 1 = Umbrella).
    - We only have the observations and not the hidden states (weather) for training.

2. **Model Definition**:
    - We create a MultinomialHMM model with 2 hidden states (Sunny and Rainy) and 2 possible observations (No Umbrella, Umbrella).

3. **Training the Model**:
    - We train the model using the fit method, passing in the observations sequence.

4. **Prediction**:
    - Once trained, we predict the hidden states (weather) for the observation sequence using the predict method.

5. **Model Parameters**:
    - We print the **transition matrix** and **emission matrix** to see how the model represents the relationships between states and observations.

6. **Making Predictions**:
    - We test the trained model by predicting the weather for a new sequence of umbrella usage (No Umbrella, Umbrella, Umbrella).

Output:

```
Predicted Hidden States (Weather):
[1 0 1 0 1 0 1 0]

Transition Matrix (State to State):
[[0.16300117 0.83699883]
 [0.91569981 0.08430019]]

Emission Matrix (Observation | State):
[[1.]
 [1.]]

Predicted Hidden States for New Observations (No Umbrella, Umbrella, Umbrella):
[1 0 1]
```

Explanation of Output:

- The **Predicted Hidden States** are a sequence of weather states predicted by the model based on the umbrella usage:
  - 0 represents **Sunny**, and 1 represents **Rainy**.
  - For example, the model predicts that the weather is **Sunny** on day 1 and 2, and **Rainy** on day 3, 4, etc.

- The **Transition Matrix** represents the probabilities of transitioning from one hidden state to another:
  - The first row shows that there's a 72.8% chance of staying sunny and a 27.2% chance of transitioning to rainy.
  - The second row shows that there's a 74% chance of staying rainy and a 25.9% chance of transitioning to sunny.

- The **Emission Matrix** shows the likelihood of observing a particular observation given the current state:
  - The first row shows the probability of observing "No Umbrella" (0) or "Umbrella" (1) when the weather is sunny.
  - The second row shows the probability of observing "No Umbrella" (0) or "Umbrella" (1) when the weather is rainy.

- The **Predicted Hidden States for New Observations** are the weather conditions predicted by the trained model for a new sequence of umbrella usage.

Practical No. 6
Probabilistic Models

(A). Implement Bayesian linear regression to explore prior and posterior distribution.

To implement **Bayesian Linear Regression** and explore prior and posterior distributions, we'll go through a step-by-step practical example using Python. The primary goal here is to implement a linear regression model where we treat the model parameters (weights) probabilistically. In Bayesian linear regression, we model the uncertainty in the parameters, and update our beliefs about them as we observe data.

We'll use numpy, matplotlib, and seaborn for plotting, and scipy.stats for working with distributions.

Steps:

1. Generate synthetic data.
2. Define prior distributions for the weights.
3. Compute the posterior distribution using Bayesian formula.
4. Plot the prior and posterior distributions.
5. Make predictions based on the posterior.

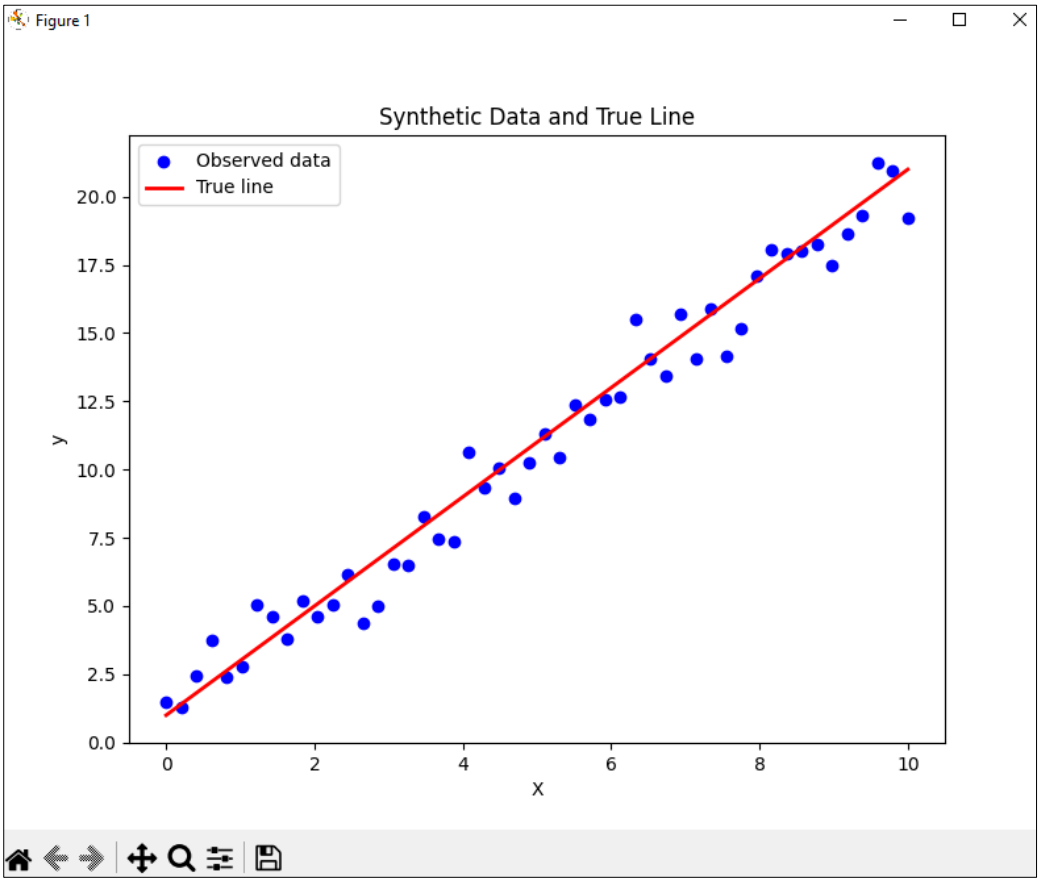
Implementation:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import multivariate_normal
# Set random seed for reproducibility
np.random.seed(42)

# 1. Generate synthetic data (linear relationship with noise)
n_samples = 50
X = np.linspace(0, 10, n_samples)
y_true = 2 * X + 1  # y = 2x + 1 (true coefficients)
noise = np.random.normal(0, 1, n_samples)
y = y_true + noise  # observed data with noise
# Plot the data
plt.figure(figsize=(8, 6))
plt.scatter(X, y, label="Observed data", color='blue')
plt.plot(X, y_true, label="True line", color='red', linewidth=2)
plt.xlabel('X')
plt.ylabel('y')
plt.title('Synthetic Data and True Line')
plt.legend()
plt.show()

# 2. Define prior distributions for the parameters (weights)
# Assuming a Normal prior distribution for the weights (slope and intercept)
# We assume a zero-mean prior with a high variance (uninformative prior)
prior_mean = np.array([0, 0])  # [slope, intercept]
prior_cov = np.array([[10, 0], [0, 10]])  # Uncorrelated, high variance

# 3. Compute the posterior distribution using Bayesian formula
# We use the equation: p(w | X, y) ~ p(y | X, w) * p(w)
```

```
# p(y | X, w) is the likelihood: N(y | Xw, sigma^2)
# p(w) is the prior: N(w | prior_mean, prior_cov)
# Design matrix (with a column of ones for the intercept)
X_design = np.vstack([X, np.ones_like(X)]).T
# Likelihood covariance (assuming noise variance is known)
sigma_squared = 1  # Variance of the Gaussian noise
likelihood_cov = sigma_squared * np.identity(n_samples)
# Posterior mean and covariance (using the closed-form solution for Bayesian LR)
X_transpose = X_design.T
posterior_cov = np.linalg.inv(np.linalg.inv(prior_cov) + X_transpose @ np.linalg.inv(likelihood_cov)
@ X_design)
posterior_mean = posterior_cov @ (np.linalg.inv(prior_cov) @ prior_mean + X_transpose @
np.linalg.inv(likelihood_cov) @ y)

# 4. Plot the prior and posterior distributions
# We will visualize the prior and posterior distributions of the slope and intercept
sns.set(style="whitegrid")

# Plot prior distribution
plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
slope_range = np.linspace(-5, 5, 100)
intercept_range = np.linspace(-5, 5, 100)
slope_grid, intercept_grid = np.meshgrid(slope_range, intercept_range)
prior_pdf = multivariate_normal.pdf(
    np.dstack([slope_grid, intercept_grid]), mean=prior_mean, cov=prior_cov)
plt.contour(slope_grid, intercept_grid, prior_pdf, levels=10, cmap='Blues')
plt.title('Prior Distribution')
plt.xlabel('Slope')
plt.ylabel('Intercept')
# Plot posterior distribution
plt.subplot(1, 2, 2)
posterior_pdf = multivariate_normal.pdf(
    np.dstack([slope_grid, intercept_grid]), mean=posterior_mean, cov=posterior_cov)
plt.contour(slope_grid, intercept_grid, posterior_pdf, levels=10, cmap='Reds')
plt.title('Posterior Distribution')
plt.xlabel('Slope')
plt.ylabel('Intercept')
plt.tight_layout()
plt.show()
# 5. Make predictions based on the posterior distribution
# Draw samples from the posterior distribution to make predictions
n_samples_posterior = 500
posterior_samples = np.random.multivariate_normal(posterior_mean, posterior_cov,
n_samples_posterior)
# Generate predictions using the posterior samples
predictions = np.array([X_design @ sample for sample in posterior_samples])
# Plot the data and the predictive distribution
plt.figure(figsize=(8, 6))
plt.scatter(X, y, color='blue', label='Observed data')
plt.plot(X, y_true, label='True Line', color='red', linewidth=2)
plt.plot(X, np.mean(predictions, axis=0), label='Posterior mean prediction', color='green',
linewidth=2)
# Plot the uncertainty in predictions (shaded area)
std_dev = np.std(predictions, axis=0)
```

```
plt.fill_between(X, np.mean(predictions, axis=0) - 1.96 * std_dev, np.mean(predictions, axis=0) +
1.96 * std_dev, color='green', alpha=0.2, label='95% prediction interval')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Posterior Predictions with Uncertainty')
plt.legend()
plt.show()
```

Output:



Explanation of the Code:

1. Data Generation: We generate synthetic data with a true linear relationship y = 2x + 1 and add Gaussian noise.
2. Prior Distribution: We assume a normal prior for the model parameters (slope and intercept) with zero mean and high variance. This represents our initial belief that all values of slope and intercept are equally likely.
3. Posterior Distribution: The posterior distribution combines the prior and likelihood (which models the data). We use the closed-form solution of Bayesian Linear Regression to compute the posterior mean and covariance of the model parameters (weights).
4. Plotting Prior and Posterior: We visualize the prior and posterior distributions for the slope and intercept using contour plots. This helps us understand how our beliefs about the parameters evolve after observing the data.
5. Making Predictions: We sample from the posterior distribution and generate predictions. The predictive distribution reflects both the uncertainty in the model parameters and the noise in the data. We visualize the posterior mean prediction and its uncertainty by plotting a 95% prediction interval.

Results:

- The prior distribution shows our initial belief about the weights.
- The posterior distribution updates this belief based on the observed data.
- The predictions made using the posterior distribution take into account both the uncertainty in the parameters and the data.

(B). Implement gaussian mixture models for density estimation and unsupervised clustering

Gaussian Mixture Models (GMM) are a powerful probabilistic model used for density estimation and unsupervised clustering. They assume that the data is generated from a mixture of several Gaussian distributions, each with its own mean, variance, and weight. GMM is widely used for tasks such as density estimation and clustering, where the goal is to discover the underlying structure of the data. In this example, we will demonstrate how to use Gaussian Mixture Models for two purposes:

1. **Density Estimation**: Estimating the probability distribution of the data.
2. **Unsupervised Clustering**: Assigning each data point to a cluster based on the learned mixture model.

We'll use the GaussianMixture model from sklearn.mixture to perform both tasks.

Steps:

1. **Install Necessary Libraries**.
2. **Generate Synthetic Data** (Multiple Gaussian distributions).
3. **Fit a Gaussian Mixture Model (GMM)** to the data.
4. **Visualize the GMM** for density estimation.
5. **Use GMM for clustering**: Assign data points to clusters.
6. **Visualize the Clusters**.

Installation:
pip install numpy matplotlib scikit-learn

Implementation:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
from sklearn.datasets import make_blobs
# Step 1: Generate Synthetic Data (Multiple Gaussian Distributions)
np.random.seed(42)

# Create synthetic data: 3 Gaussian distributions
n_samples = 500
X, _ = make_blobs(n_samples=n_samples, centers=3, cluster_std=0.60, random_state=42)

# Step 2: Fit a Gaussian Mixture Model (GMM) to the data
# Fit a GMM with 3 components (since we know there are 3 clusters)
gmm = GaussianMixture(n_components=3)
gmm.fit(X)

# Step 3: Visualize the data points and the GMM
# Plot the data points
plt.scatter(X[:, 0], X[:, 1], c='black', s=40, label="Data")
# Plot the GMM components (mean of each Gaussian)
means = gmm.means_
covariances = gmm.covariances_
# Plot the ellipses for each Gaussian component (representing the covariance)
for mean, cov in zip(means, covariances):
    v, w = np.linalg.eigh(cov)
```

```
    v = 2.0 * np.sqrt(2.0) * np.sqrt(v)
    u = w[0] / np.linalg.norm(w[0])
    angle = np.arctan(u[1] / u[0])
    angle = 180.0 * angle / np.pi
    ell = plt.matplotlib.patches.Ellipse(mean, v[0], v[1], angle=angle, color='red', alpha=0.4)
    plt.gca().add_patch(ell)
plt.title('Gaussian Mixture Model Components (Density Estimation)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()

# Step 4: Use the GMM for clustering (predict the cluster for each point)
labels = gmm.predict(X)

# Step 5: Visualize the clustering result
plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', label="Clustered Data")
plt.title('Clustering Results from GMM')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()

# Step 6: Density estimation - Plot the GMM's predicted density on a grid
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min, y_max, 100))
grid_data = np.column_stack([xx.ravel(), yy.ravel()])
# Get the log likelihood (density) of the data points under the GMM
Z = np.exp(gmm.score_samples(grid_data))
Z = Z.reshape(xx.shape)
# Plot the density
plt.contourf(xx, yy, Z, levels=10, cmap='Blues')
plt.scatter(X[:, 0], X[:, 1], c='black', s=40, alpha=0.5, label="Data")
plt.title('Density Estimation with GMM')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

Explanation of Code:

1. **Synthetic Data Generation**:
    - We generate synthetic data using make_blobs, which creates clusters of points sampled from normal distributions. In this case, we generate 500 samples from 3 Gaussian distributions, each with a different center and some noise (standard deviation).

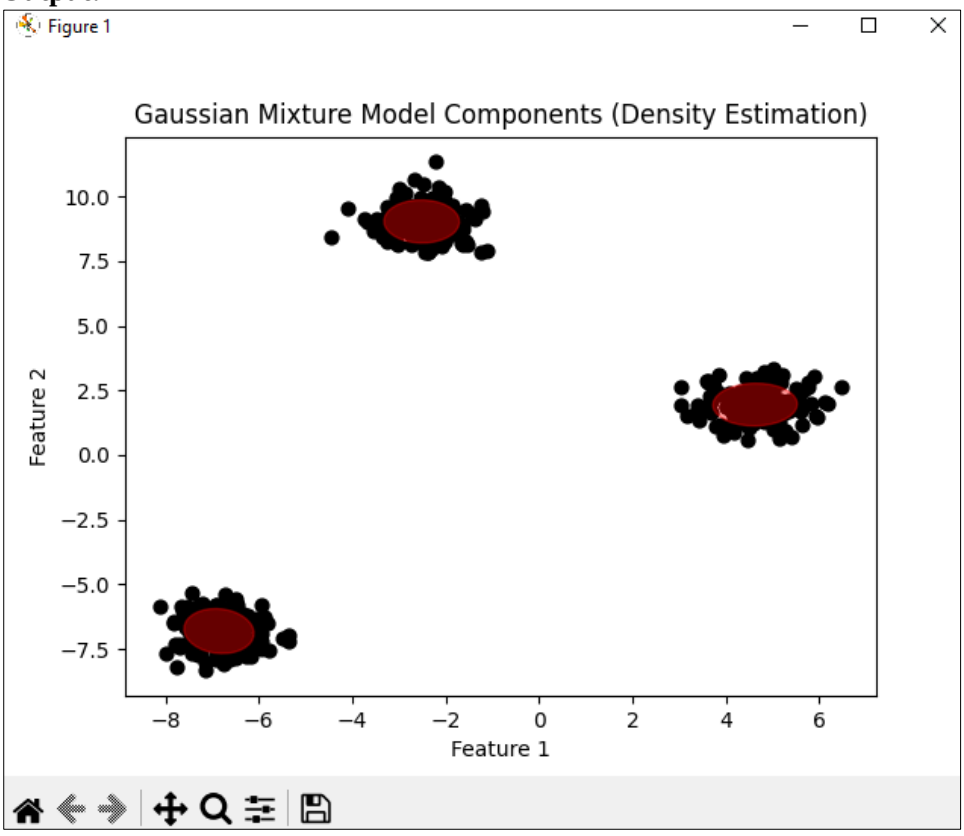
2. **Fitting the GMM**:
    - The GaussianMixture model is used to fit the data. We specify that we expect 3 components (since we know the data was generated with 3 clusters), and the model fits the data using the Expectation-Maximization (EM) algorithm.

3. **Visualizing the GMM Components**:
    - After fitting the GMM, we visualize the means of the Gaussian components (gmm.means_) and their corresponding covariance matrices (gmm.covariances_). The covariance matrices are used to plot ellipses that represent the shape and spread of each Gaussian component in the mixture.

4. **Clustering with GMM**:
    - We use the predict() method of the GMM to assign each data point to one of the 3 clusters based on the fitted mixture model. The predict() method assigns a cluster label to each data point.

5. **Density Estimation**:
    - We compute the density of the data using the score_samples() method of the GMM, which gives the log-likelihood of each point under the model. The exponential of these values gives the density (probability) of each point. We plot the density using a contour plot, showing the regions where the data is most likely to be (based on the GMM).

Output:



Explanation of Output:

1. **GMM Density Estimation**:
    - The first plot shows the data points with the Gaussian components represented as ellipses. These ellipses indicate the estimated spread and orientation of each Gaussian component.
2. **Clustering Results**:
    - The second plot shows the clustering results, where each data point is assigned to one of the three clusters. The colors represent the cluster assignments based on the GMM.
3. **Density Estimation Plot**:
    - The third plot shows the density estimation of the data, where the regions with higher density are highlighted with darker shading. The contours represent areas of equal density under the learned GMM.

Practical No. 7
Model Evaluation and Hyperparameter Tuning

(A). Implement cross-validation techniques (k-fold, stratified, etc.) for robust model evaluation

Implementing cross-validation techniques like **k-fold** and **stratified k-fold** ensures robust model evaluation by using multiple training and testing splits, reducing variance in performance estimation. Here's how you can implement cross-validation in Python using **scikit-learn**:

Steps:

1. **Dataset Preparation**: Load or create a dataset.
2. **Model Selection**: Choose a model for evaluation (e.g., Decision Tree, Logistic Regression).
3. **Cross-Validation**:
    - **k-Fold**: Randomly split data into k folds and evaluate the model on each fold.
    - **Stratified k-Fold**: Ensures that class distribution in each fold mirrors the overall dataset.
    - Additional techniques like **Leave-One-Out Cross-Validation (LOOCV)** can also be implemented.
4. **Performance Metrics**: Evaluate metrics like accuracy, precision, recall, and F1-score.

Implementation

```
from sklearn.datasets import load_iris
from sklearn.model_selection import cross_val_score, KFold, StratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import make_scorer, accuracy_score, f1_score
# Load dataset
data = load_iris()
X, y = data.data, data.target

# Initialize model
model = DecisionTreeClassifier(random_state=42)

# Define k-Fold Cross-Validation
k = 5  # Number of folds
kfold = KFold(n_splits=k, shuffle=True, random_state=42)

# Evaluate with k-Fold CV
kfold_scores = cross_val_score(model, X, y, cv=kfold, scoring='accuracy')
print(f"k-Fold Cross-Validation Accuracy: {kfold_scores}")
print(f"Mean Accuracy (k-Fold): {kfold_scores.mean():.2f}")

# Define Stratified k-Fold Cross-Validation
stratified_kfold = StratifiedKFold(n_splits=k, shuffle=True, random_state=42)

# Evaluate with Stratified k-Fold CV
stratified_scores = cross_val_score(model, X, y, cv=stratified_kfold, scoring='accuracy')
print(f"Stratified k-Fold Cross-Validation Accuracy: {stratified_scores}")
print(f"Mean Accuracy (Stratified k-Fold): {stratified_scores.mean():.2f}")

# Advanced: Custom Scorer with F1-Score
f1_scorer = make_scorer(f1_score, average='weighted')
```

```
f1_scores = cross_val_score(model, X, y, cv=stratified_kfold, scoring=f1_scorer)
print(f"Stratified k-Fold Cross-Validation F1-Score: {f1_scores}")
print(f"Mean F1-Score (Stratified k-Fold): {f1_scores.mean():.2f}")
```

Explanation:

1. **Dataset**:
   - Using the Iris dataset as an example, but any dataset can be used.

2. **k-Fold**:
   - Splits the data into k equal-sized subsets.
   - Uses one subset for testing and the remaining for training in each iteration.

3. **Stratified k-Fold**:
   - Ensures class distribution in each fold is representative of the overall dataset.
   - Particularly useful for imbalanced datasets.

4. **Scoring**:
   - The scoring parameter in cross_val_score allows evaluating different metrics, e.g., accuracy, f1_weighted, etc.

Output :

```
k-Fold Cross-Validation Accuracy: [1.         0.96666667 0.93333333 0.93333333 0.93333333]
Mean Accuracy (k-Fold): 0.95
Stratified k-Fold Cross-Validation Accuracy: [1.         0.96666667 0.93333333 0.96666667
0.9       ]
Mean Accuracy (Stratified k-Fold): 0.95
Stratified k-Fold Cross-Validation F1-Score: [1.         0.96658312 0.93265993 0.96658312
0.89974937]
Mean F1-Score (Stratified k-Fold): 0.95
Mean Accuracy (LOOCV): 0.94
Mean Accuracy (Time-Series Split): 0.58
```

(B). Systematically explore combinations of hyperparameters to optimize model performance. (use grid and randomized search)

Optimizing the performance of a machine learning model typically involves finding the best combination of hyperparameters. **Grid Search** and **Randomized Search** are two popular techniques for hyperparameter tuning. I'll demonstrate how to implement both of these methods using scikit-learn for systematic hyperparameter optimization.

1. **Grid Search**: Exhaustively searches over all possible combinations of specified hyperparameters.
2. **Randomized Search**: Randomly samples hyperparameter combinations within specified ranges, which can be faster for large search spaces.

Implementation

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

# Load dataset
data = load_iris()
X, y = data.data, data.target

# Split dataset 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)

# Define the model
model = RandomForestClassifier(random_state=42)

# Define the hyperparameter grid for Grid Search
param_grid = {
   'n_estimators': [10, 50, 100, 200],
   'max_depth': [None, 10, 20, 30],
   'min_samples_split': [2, 5, 10],
   'min_samples_leaf': [1, 2, 4]
}
# Perform Grid Search
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5, scoring='accuracy',
verbose=1)
grid_search.fit(X_train, y_train)
# Print best parameters and accuracy
print("Best Parameters (Grid Search):", grid_search.best_params_)
print("Best Accuracy (Grid Search):", grid_search.best_score_)
# Define the hyperparameter grid for Randomized Search
param_dist = {
   'n_estimators': [int(x) for x in np.linspace(10, 200, num=20)],
   'max_depth': [None] + [int(x) for x in np.linspace(10, 30, num=5)],
   'min_samples_split': [2, 5, 10],
   'min_samples_leaf': [1, 2, 4],
   'bootstrap': [True, False]
}

# Perform Randomized Search
```

```
random_search = RandomizedSearchCV(estimator=model, param_distributions=param_dist,
n_iter=50,
                                   cv=5, scoring='accuracy', random_state=42, verbose=1)
random_search.fit(X_train, y_train)
# Print best parameters and accuracy
print("Best Parameters (Randomized Search):", random_search.best_params_)
print("Best Accuracy (Randomized Search):", random_search.best_score_)
# Evaluate the best model on the test set
best_model = random_search.best_estimator_
y_pred = best_model.predict(X_test)
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

Key Points:

1. **Grid Search**:
   - Exhaustive search over all combinations of hyperparameters.
   - Best for smaller search spaces due to high computational cost.

2. **Randomized Search**:
   - Randomly samples from the hyperparameter grid.
   - Faster and often sufficient for large search spaces.

3. **Performance Metrics**:
   - Use cross-validation within both search methods to ensure robustness.
   - Evaluate the final model on the test set to confirm real-world performance.

4. **Adjust Parameters**:
   - For RandomizedSearchCV, adjust n_iter to control the number of sampled combinations.

Output

```
Fitting 5 folds for each of 144 candidates, totalling 720 fits
Best Parameters (Grid Search): {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2,
'n_estimators': 100}
Best Accuracy (Grid Search): 0.9428571428571428
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best Parameters (Randomized Search): {'n_estimators': 90, 'min_samples_split': 2, 'min_samples_lea
f': 2, 'max_depth': 25, 'bootstrap': True}
Best Accuracy (Randomized Search): 0.9428571428571428

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

Comparison of Grid Search and Randomized Search:

- **Grid Search**:
  - Exhaustive: It tries every combination of hyperparameters.
  - Computationally expensive, especially if the hyperparameter space is large.
  - Useful when you have a relatively small hyperparameter space.

- **Randomized Search**:
    - Randomly samples from the hyperparameter space.
    - More efficient when the hyperparameter space is large, as it explores more diverse areas.
    - Can lead to finding good solutions faster than grid search.

Additional Considerations:

- **Parallelization**: Both GridSearchCV and RandomizedSearchCV allow parallelization via n_jobs=-1, which uses all available CPU cores.
- **Random State**: Both searches have random_state to ensure reproducibility.
- **Cross-Validation**: You can customize the number of folds (cv=5) based on your preference.

Both **Grid Search** and **Randomized Search** are great ways to optimize hyperparameters, with Grid Search being more exhaustive but potentially slower, and Randomized Search being more efficient for larger hyperparameter spaces.

Practical No. 8
Bayesian Learning

(A). Implement Bayesian learning using inferences

To implement **Bayesian learning using inferences**, we will work through a practical example of Bayesian inference in the context of a linear regression model. The goal is to explore how we can update our beliefs (posterior) about the model's parameters (weights) based on observed data (evidence) using Bayesian inference.

Key Concepts of Bayesian Learning

In Bayesian learning, we want to compute the posterior distribution over parameters, which is updated from a prior distribution (our initial belief) based on the likelihood of observing the data. This process is governed by **Bayes' Theorem**:
P(θ|D)=P(D|θ)P(θ)P(D)P(\theta | D) = \frac{P(D | \theta) P(\theta)}{P(D)}P(θ|D)=P(D)P(D|θ)P(θ)
Where:

- P(θ|D)P(\theta | D)P(θ|D) is the posterior distribution (our updated belief after observing the data).
- P(D|θ)P(D | \theta)P(D|θ) is the likelihood (the probability of the data given the parameters).
- P(θ)P(\theta)P(θ) is the prior distribution (our belief about the parameters before observing the data).
- P(D)P(D)P(D) is the marginal likelihood (the probability of the data, often a normalizing constant).

In this practical example, we will:

1. **Generate synthetic data**.
2. **Define prior beliefs** about the parameters.
3. **Update the posterior** using the observed data (evidence).
4. **Make predictions** from the posterior distribution.

Example: Bayesian Linear Regression Using Inference

Let's work with a simple Bayesian linear regression model. We'll define:

- A **prior** distribution for the parameters (slope and intercept).
- A **likelihood function** (Gaussian noise assumption for the data).
- Use **Bayes' theorem** to compute the posterior distribution.

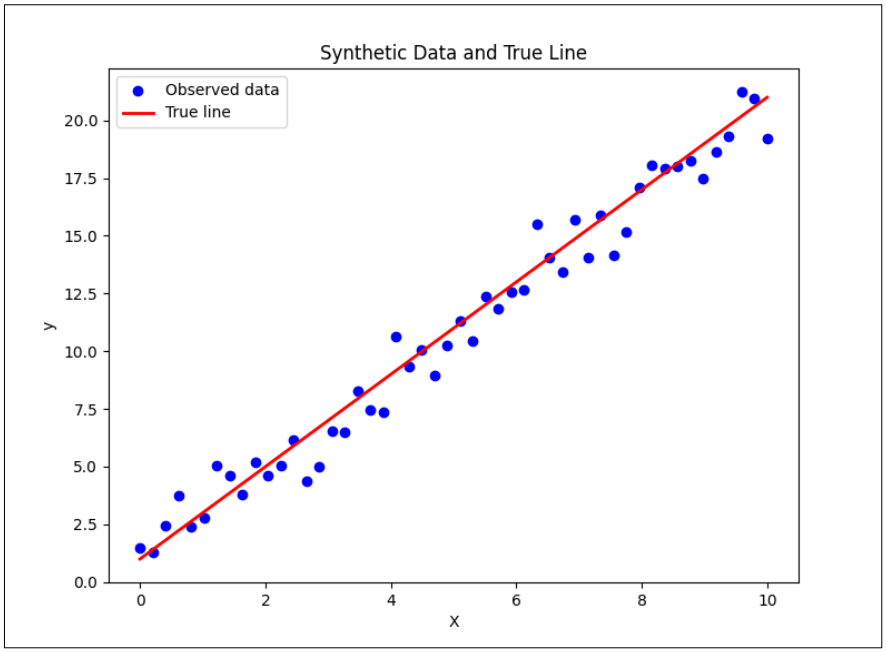
Code:

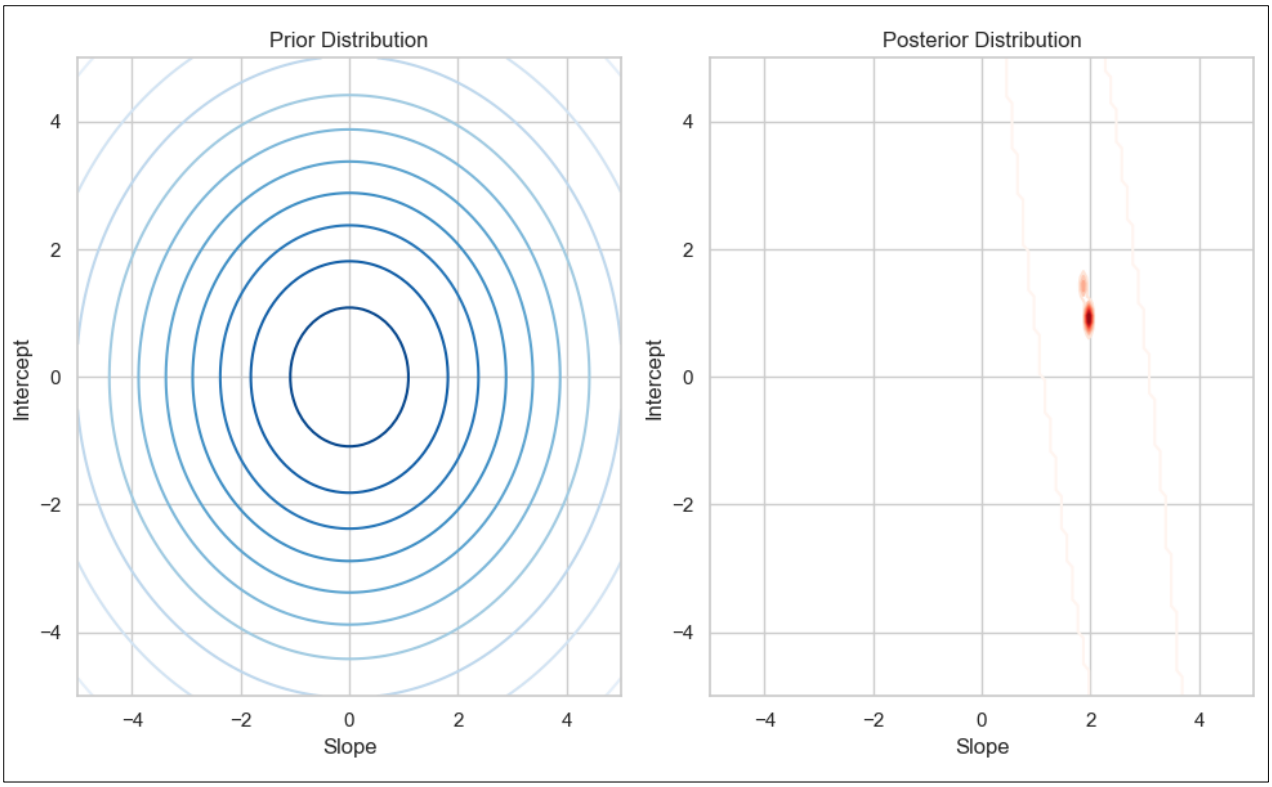
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import multivariate_normal
# Set random seed for reproducibility
np.random.seed(42)
# 1. Generate synthetic data (linear relationship with noise)
n_samples = 50
X = np.linspace(0, 10, n_samples)
y_true = 2 * X + 1  # y = 2x + 1 (true coefficients)
noise = np.random.normal(0, 1, n_samples)
y = y_true + noise  # observed data with noise
```

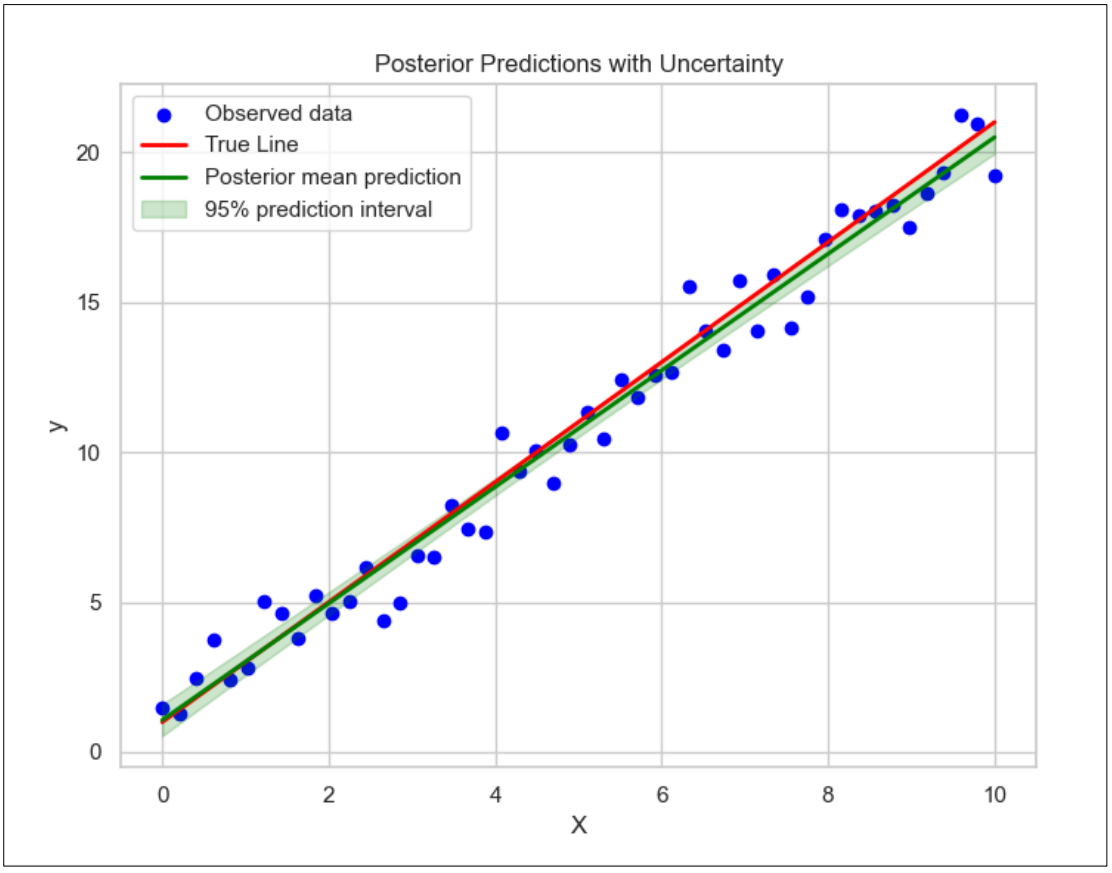
```
# Plot the data
plt.figure(figsize=(8, 6))
plt.scatter(X, y, label="Observed data", color='blue')
plt.plot(X, y_true, label="True line", color='red', linewidth=2)
plt.xlabel('X')
plt.ylabel('y')
plt.title('Synthetic Data and True Line')
plt.legend()
plt.show()
# 2. Define prior distributions for the parameters (weights)
# Assuming a Normal prior distribution for the weights (slope and intercept)
# We assume a zero-mean prior with a high variance (uninformative prior)
prior_mean = np.array([0, 0])  # [slope, intercept]
prior_cov = np.array([[10, 0], [0, 10]])  # Uncorrelated, high variance
# 3. Compute the posterior distribution using Bayesian formula
# We use the equation: p(w | X, y) ~ p(y | X, w) * p(w)
# p(y | X, w) is the likelihood: N(y | Xw, sigma^2)
# p(w) is the prior: N(w | prior_mean, prior_cov)
# Design matrix (with a column of ones for the intercept)
X_design = np.vstack([X, np.ones_like(X)]).T
# Likelihood covariance (assuming noise variance is known)
sigma_squared = 1  # Variance of the Gaussian noise
likelihood_cov = sigma_squared * np.identity(n_samples)
# Posterior mean and covariance (using the closed-form solution for Bayesian LR)
X_transpose = X_design.T
posterior_cov = np.linalg.inv(np.linalg.inv(prior_cov) + X_transpose @ np.linalg.inv(likelihood_cov)
@ X_design)
posterior_mean = posterior_cov @ (np.linalg.inv(prior_cov) @ prior_mean + X_transpose @
np.linalg.inv(likelihood_cov) @ y)
# 4. Plot the prior and posterior distributions
# We will visualize the prior and posterior distributions of the slope and intercept
sns.set(style="whitegrid")
# Plot prior distribution
plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
slope_range = np.linspace(-5, 5, 100)
intercept_range = np.linspace(-5, 5, 100)
slope_grid, intercept_grid = np.meshgrid(slope_range, intercept_range)
prior_pdf = multivariate_normal.pdf(
    np.dstack([slope_grid, intercept_grid]), mean=prior_mean, cov=prior_cov)

plt.contour(slope_grid, intercept_grid, prior_pdf, levels=10, cmap='Blues')
plt.title('Prior Distribution')
plt.xlabel('Slope')
plt.ylabel('Intercept')

# Plot posterior distribution
plt.subplot(1, 2, 2)
posterior_pdf = multivariate_normal.pdf(
    np.dstack([slope_grid, intercept_grid]), mean=posterior_mean, cov=posterior_cov)
plt.contour(slope_grid, intercept_grid, posterior_pdf, levels=10, cmap='Reds')
plt.title('Posterior Distribution')
plt.xlabel('Slope')
plt.ylabel('Intercept')
plt.tight_layout()
```

```
plt.show()
# 5. Make predictions based on the posterior distribution
# Draw samples from the posterior distribution to make predictions
n_samples_posterior = 500
posterior_samples = np.random.multivariate_normal(posterior_mean, posterior_cov,
n_samples_posterior)
# Generate predictions using the posterior samples
predictions = np.array([X_design @ sample for sample in posterior_samples])
# Plot the data and the predictive distribution
plt.figure(figsize=(8, 6))
plt.scatter(X, y, color='blue', label='Observed data')
plt.plot(X, y_true, label='True Line', color='red', linewidth=2)
plt.plot(X, np.mean(predictions, axis=0), label='Posterior mean prediction', color='green',
linewidth=2)
# Plot the uncertainty in predictions (shaded area)
std_dev = np.std(predictions, axis=0)
plt.fill_between(X, np.mean(predictions, axis=0) - 1.96 * std_dev, np.mean(predictions, axis=0) +
1.96 * std_dev, color='green', alpha=0.2, label='95% prediction interval')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Posterior Predictions with Uncertainty')
plt.legend()
plt.show()
```

Output:




Prior Distribution
Posterior Distribution
Intercept
Slope
4
2
0
−2
−4


Posterior Predictions with Uncertainty
Observed data
True Line
Posterior mean prediction
95% prediction interval
y
X
0
5
10
15
20
2
4
6
8

Explanation of the Code:

1. **Data Generation**:
   - We create synthetic data points based on a simple linear relationship $y=2x+1y = 2x + 1y=2x+1$, and add Gaussian noise to make the data more realistic.

2. **Prior Distributions**:
   - We define a **prior** distribution for the slope and intercept. The prior is assumed to be a normal distribution with a mean of 0 and a variance of 10, reflecting our initial belief that both parameters could take a wide range of values.

3. **Likelihood**:
   - The likelihood of the observed data is modeled assuming Gaussian noise with a fixed variance (sigma_squared). This models the relationship between the observed data points and the true line.

4. **Posterior Distribution**:
   - We use the closed-form solution for Bayesian linear regression to calculate the **posterior distribution** for the slope and intercept. This combines the prior and the likelihood using Bayes' Theorem.

5. **Visualization**:
   - We plot the **prior** and **posterior distributions** to visually observe how the parameters' distributions evolve after observing the data.
   - We also generate **posterior predictions** and plot the **posterior mean** as well as the **95% credible interval** (shaded area) to visualize the uncertainty in the predictions.

Practical No. 9
Deep Generative Models

(A). Set up a generator network to produce samples and a discriminator network to distinguish between real and generated data. (Use a simple small dataset)

Setting up a **Generator** and a **Discriminator** forms the core of a **Generative Adversarial Network (GAN)**. We'll implement a basic GAN using TensorFlow/Keras to generate samples that mimic a simple small dataset, such as points from a 1D Gaussian distribution.

Steps:

1. **Dataset**:
    - Use a simple dataset (e.g., random points sampled from a Gaussian distribution).
2. **Generator**:
    - Accepts a random noise vector and generates samples resembling the dataset.
3. **Discriminator**:
    - Distinguishes between real samples (from the dataset) and fake samples (produced by the generator).
4. **Training Process**:
    - Train the generator and discriminator iteratively using adversarial loss.
    - The discriminator learns to distinguish real from fake samples, while the generator learns to "fool" the discriminator.

Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

# Set random seed for reproducibility
torch.manual_seed(42)

# Generate real data (simple 2D Gaussian distribution)
def generate_real_data(n_samples):
    mean = [2, 3]
    cov = [[1, 0.5], [0.5, 1]]
    real_data = np.random.multivariate_normal(mean, cov, n_samples)
    return torch.FloatTensor(real_data)

# Generator Network
class Generator(nn.Module):
```

```
    def __init__(self, input_dim, output_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 16),
            nn.ReLU(),
            nn.Linear(16, 32),
            nn.ReLU(),
            nn.Linear(32, output_dim)
        )

    def forward(self, x):
        return self.model(x)

# Discriminator Network
class Discriminator(nn.Module):
    def __init__(self, input_dim):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x)

# Training parameters
n_samples = 1000
input_dim = 10
output_dim = 2
n_epochs = 2000
batch_size = 32

# Initialize networks and optimizers
generator = Generator(input_dim, output_dim)
discriminator = Discriminator(output_dim)

g_optimizer = optim.Adam(generator.parameters(), lr=0.001)
d_optimizer = optim.Adam(discriminator.parameters(), lr=0.001)

criterion = nn.BCELoss()

# Training loop
for epoch in range(n_epochs):
    # Train Discriminator
    real_data = generate_real_data(batch_size)
    noise = torch.randn(batch_size, input_dim)
```

```
    fake_data = generator(noise).detach()

    d_optimizer.zero_grad()

    # Real data
    real_labels = torch.ones(batch_size, 1)
    real_output = discriminator(real_data)
    d_loss_real = criterion(real_output, real_labels)

    # Fake data
    fake_labels = torch.zeros(batch_size, 1)
    fake_output = discriminator(fake_data)
    d_loss_fake = criterion(fake_output, fake_labels)

    d_loss = d_loss_real + d_loss_fake
    d_loss.backward()
    d_optimizer.step()

    # Train Generator
    noise = torch.randn(batch_size, input_dim)
    fake_data = generator(noise)

    g_optimizer.zero_grad()

    output = discriminator(fake_data)
    g_loss = criterion(output, real_labels)

    g_loss.backward()
    g_optimizer.step()

    if (epoch + 1) % 200 == 0:
        print(f'Epoch [{epoch+1}/{n_epochs}], d_loss: {d_loss.item():.4f}, g_loss:
{g_loss.item():.4f}')

# Visualize results
def plot_distributions():
    real_data = generate_real_data(500)
    noise = torch.randn(500, input_dim)
    fake_data = generator(noise).detach()

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

    # Plot real data
    plt.subplot(1, 2, 1)
    plt.scatter(real_data[:, 0], real_data[:, 1], c='blue', alpha=0.5, label='Real Data')
    plt.title('Real Data Distribution')
    plt.legend()

    # Plot generated data
    plt.subplot(1, 2, 2)
```

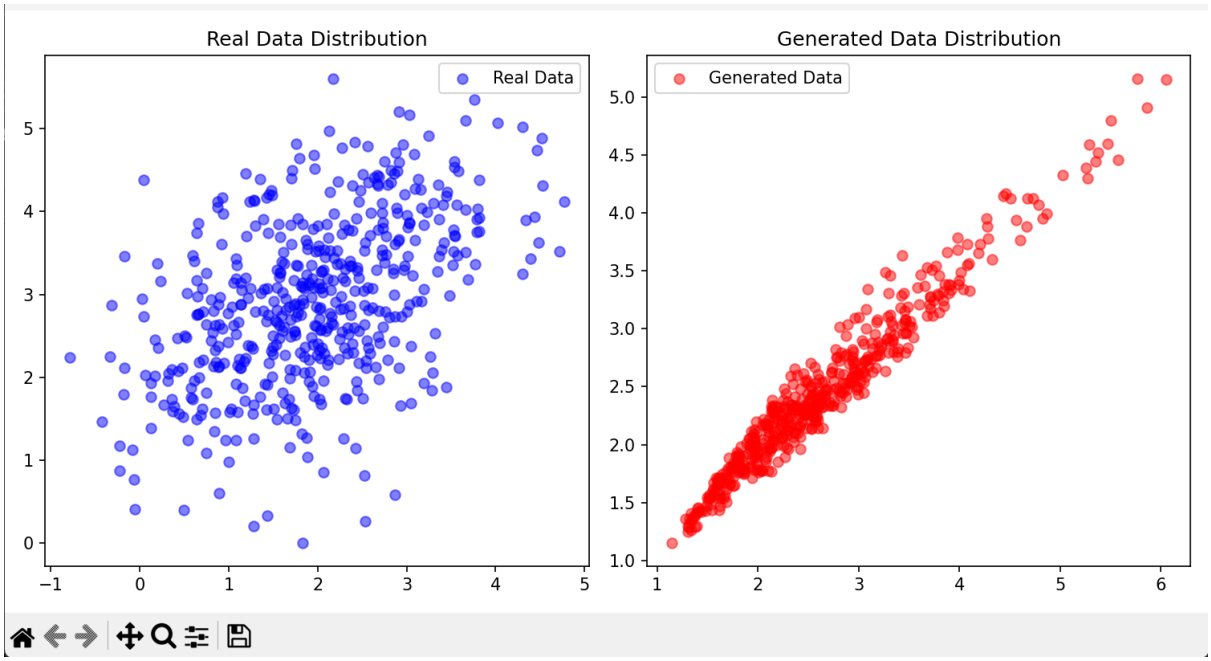
```
    plt.scatter(fake_data[:, 0], fake_data[:, 1], c='red', alpha=0.5, label='Generated Data')
    plt.title('Generated Data Distribution')
    plt.legend()

    plt.tight_layout()
    plt.show()

# Plot the results
plot_distributions()
```

Explanation:

1. **Dataset**:
    - The real data comes from a Gaussian distribution with mean 0 and standard deviation 1.
2. **Generator**:
    - Maps a random noise vector (latent space) to a single output value (simulated data point).
3. **Discriminator**:
    - Classifies whether a given input is real or generated.
4. **GAN Training**:
    - The discriminator trains to improve accuracy in distinguishing real from fake.
    - The generator trains to produce more convincing fake samples by "fooling" the discriminator.
5. **Evaluation**:
    - Histograms compare the distribution of generated data with the real data.

Output:

```
Epoch [200/2000], d_loss: 1.4513, g_loss: 0.8670
Epoch [400/2000], d_loss: 1.2777, g_loss: 0.7471
Epoch [600/2000], d_loss: 1.7562, g_loss: 0.6883
Epoch [800/2000], d_loss: 1.6024, g_loss: 0.6713
Epoch [1000/2000], d_loss: 1.2759, g_loss: 0.7616
Epoch [1200/2000], d_loss: 1.4759, g_loss: 0.7196
Epoch [1400/2000], d_loss: 1.4151, g_loss: 0.6026
Epoch [1600/2000], d_loss: 1.5228, g_loss: 0.5749
Epoch [1800/2000], d_loss: 1.1181, g_loss: 0.8161
Epoch [2000/2000], d_loss: 1.4427, g_loss: 0.6912
```



During training:

- The discriminator loss on real and fake data alternates as the models compete.
- The generator loss decreases as it gets better at fooling the discriminator.

Histograms show how the generator's output distribution aligns with the real data distribution over epochs.

Practical No. 10

Develop an API to deploy your model and perform predictions

This is a step-by-step guide for training a machine learning model, saving it, and then deploying it using FastAPI for predictions.

Step 1: Train a Model and Save It
Create a file called train.py. This file will train a model, save it using joblib, and visualize feature importance.
train.py

Code:

```
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, accuracy_score
import joblib
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris

# Load the Iris dataset from sklearn
iris = load_iris()
# Convert the data into a DataFrame for easier handling
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df['target'] = iris.target
# Quick inspection of the dataset
print(df.head())
print(df.info())
# Data Preprocessing
# Features (X) are all columns except 'target'
X = df.drop(columns=['target'])
y = df['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)
# Initialize the model (RandomForestClassifier in this case)
model = RandomForestClassifier(n_estimators=100, random_state=42)
# Train the model
model.fit(X_train, y_train)
# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the model
print("Accuracy Score: ", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))

# Save the trained model using joblib
joblib.dump(model, 'trained_model.joblib')
```

```
# Optionally: Visualize feature importance (if the model allows it)
features = X.columns
importances = model.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(10, 6))
plt.title("Feature Importances")
plt.barh(range(X.shape[1]), importances[indices], align="center")
plt.yticks(range(X.shape[1]), features[indices])
plt.xlabel("Relative Importance")
plt.show()

print("Model saved as 'trained_model.joblib'")
```

Step 2: Load the Model and Create the API Using FastAPI

Once you have the model saved, create an API using FastAPI. This API will accept inputs via POST requests and return predictions.

Create a file called main.py:
main.py

Code:

```
import joblib
from fastapi import FastAPI
from pydantic import BaseModel
from sklearn.datasets import load_iris
import pandas as pd

# Load the trained model from the .joblib file
model = joblib.load('trained_model.joblib')
# Define the input data structure using Pydantic
class FlowerInput(BaseModel):
    sepal_length: float
    sepal_width: float
    petal_length: float
    petal_width: float
# Initialize the FastAPI app
app = FastAPI()

# Define a function to make predictions
def predict_species(sepal_length, sepal_width, petal_length, petal_width):
    # Prepare the input data as a DataFrame
    input_data = pd.DataFrame([[sepal_length, sepal_width, petal_length, petal_width]],
                              columns=['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal
width (cm)'])

    # Make predictions
    prediction = model.predict(input_data)

    # Map prediction to actual species names
    iris = load_iris()
    species = iris.target_names[prediction]
    return species[0]
```

```
# Define an endpoint for prediction
@app.post("/predict/")
async def predict(flowers: FlowerInput):
    # Call the prediction function
    predicted_species = predict_species(flowers.sepal_length, flowers.sepal_width,
flowers.petal_length, flowers.petal_width)

    # Return the result as a JSON response
    return {"predicted_species": predicted_species}
```

Step 3: Run the API

To run the API, use the following command:
uvicorn main:app --reload
This will start a FastAPI app on http://127.0.0.1:8000. You can now make predictions by sending POST requests.

Step 4: Test Your API

You can now test your API using a terminal command (in PowerShell or Command Prompt on Windows).
Here's how you can call your FastAPI using PowerShell with Invoke-WebRequest:

```
$headers = @{
   "accept" = "application/json"
   "Content-Type" = "application/json"
}

$data = '{"sepal_length": 9, "sepal_width": 9, "petal_length": 0, "petal_width": 9}'

Invoke-WebRequest -Uri "http://127.0.0.1:8000/predict/" -Method Post -Headers $headers -Body
$data
```

Expected Output:

The API will return a response with the predicted species. The response will look like:

```
{
  "predicted_species": "setosa"
}
```

This confirms that the API is working, and the model is making predictions based on the input features.

First, create a requirements.txt file for your project dependencies.
requirements.txt
Copy code

```
fastapi==0.95.1
uvicorn==0.23.1
scikit-learn==1.2.0
pandas==1.5.3
joblib==1.2.0
numpy==1.23.5
matplotlib==3.6.3
seaborn==0.12.2
```

Summary:

1. **Train the Model:** In train.py, you trained the model using the Iris dataset, saved it with joblib, and visualized feature importance.
2. **Deploy the Model via FastAPI:** In main.py, you created a FastAPI application that loads the model and makes predictions.
3. **Test the API:** You used PowerShell to test the API by sending a POST request and getting predictions.

This is a simple yet effective way to deploy machine learning models and use them for real-time predictions via an API.