PRML_(CSL2050) Programming Assignment 2_3

Report

Name-Soumen Kumar
Roll No.-B22ES006

Question1-Decision Tree Implementation

Implementing a classification decision tree from scratch to classify whether someone will survive in the Titanic wreck.

Task_1-

Features of the given dataset-

Categorical variables represent distinct categories without inherent order, while ordinal variables have categories with a meaningful order or ranking.

Feature	Type
PassengerId	— Categorical
Survived	— Categorical
Pclass	— Ordinal
Name	— Categorical
Sex	— Categorical
Age	— Continuous Numerical(neither ordinal nor nominal)
SibSp	— Continuous Numerical(neither ordinal nor nominal)
Parch	— Continuous Numerical(neither ordinal nor nominal)
Ticket	— Categorical
Fare	— Continuous Numerical(neither ordinal nor nominal)
Cabin	— Categorical
Embarked	— Categorical

Data Exploration and Visualization–

For this, I have used the following to display the loaded data frame, its details and its statistics some histogram plots displaying relations between target variable with some other data fields

```
# Displaying the first five rows of the dataset
display(df.head())

# Displaying basic information about the dataset
print("Dataset Information:")
display(df.info())

# Summary statistics for numerical features
display(df.describe())

# Histograms for numerical features
df.hist(bins=15, figsize=(15, 6), layout=(2, 4),color='red')
plt.suptitle('Histograms of Numerical Features')
plt.show()

# Survival based on different features
for column in ['Pclass', 'Sex', 'Embarked', 'SibSp', 'Parch']:
    plt.figure(figsize=(8, 5))
    sns.countplot(x='Survived', hue=column, data=df, palette="Set1")  #
Change x='Survived' for counts on the x-axis
    plt.title(f'Survival Rate based on {column}')
    plt.xlabel('Survived')
    plt.ylabel('Count')
    plt.show()
```

Pre-processing data is a crucial machine learning step involving cleaning and transforming raw data into a format suitable for training a model.

Handling Missing Values:

- Identify missing values in each column.
- Decide on a strategy to handle missing values (e.g., imputation, removal).
- Common imputation methods include mean, median, or mode imputation for numerical data, and using the most frequent category for categorical data.

```
# Check for missing values and printing it.
display(df.isnull().sum())

# Handling missing values
df['Age'].fillna(df['Age'].mean(), inplace=True)
df['Fare'].fillna(df['Fare'].mean(), inplace=True)
```

```
df['Embarked'].fillna(df['Embarked'].mode()[0], inplace=True)
```

In this problem, I have done the following to fill in missing data in some of the data fields

The data is missing from three Columns:
1. PassengerId (177/891) => 20%
2. Cabin (687/891) => 77%
3. Embarked (2/891) => 0.2%

The cabin column is being removed as it has an amount of data missing and it's not important also

df['Age'].fillna(df['Age'].mean(), inplace=True): This line fills missing values in the 'Age' column with the mean (average) value of the non-missing values in that column. This is a common strategy for imputing missing numerical data when the missing values are assumed to be missing at random and unrelated to any specific pattern.

df['Fare'].fillna(df['Fare'].mean(), inplace=True): Similar to the first line, this line fills missing values in the 'Fare' column with the mean fare value. Again, this assumes that missing fare values are missing at random.

df['Embarked'].fillna(df['Embarked'].mode()[0], inplace=True): This line fills missing values in the 'Embarked' column with the mode (most frequently occurring value) of the non-missing values in that column. This is often done when dealing with categorical data, and the mode is used to impute missing values, assuming that the most common category is a reasonable estimate for missing values.

```
# Droping irrelevant columns
df = df.drop(['Cabin'], axis=1)
df.drop(columns = ["PassengerId", "Name", "Ticket"], inplace = True)
```

Categorical Encoding:

- Convert categorical variables into numerical representations, as many machine learning algorithms work with numerical data.
- Common encoding methods include one-hot encoding and label encoding.

```
# Performing Categorical Encoding
#Sex Column
df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})
```

```
#embarked column
def encode_embarked(column):
  for data in column:
    if data == 'S':
      column[column.index(data)] = 0
    elif data == 'C':
      column[column.index(data)] = 1
    else:
      column[column.index(data)] = 2
  return column
df["Embarked"] = encode_embarked(list(df["Embarked"]))
```

Encoding 'Sex' Column:

- The 'Sex' column contains categorical values ('male' and 'female').
- The code uses the map function to replace 'male' with 0 and 'female' with 1.
- This transformation converts the 'Sex' column into numerical values (0 for male, 1 for female), making it suitable for machine learning algorithms that require numerical input.

Encoding 'Embarked' Column:

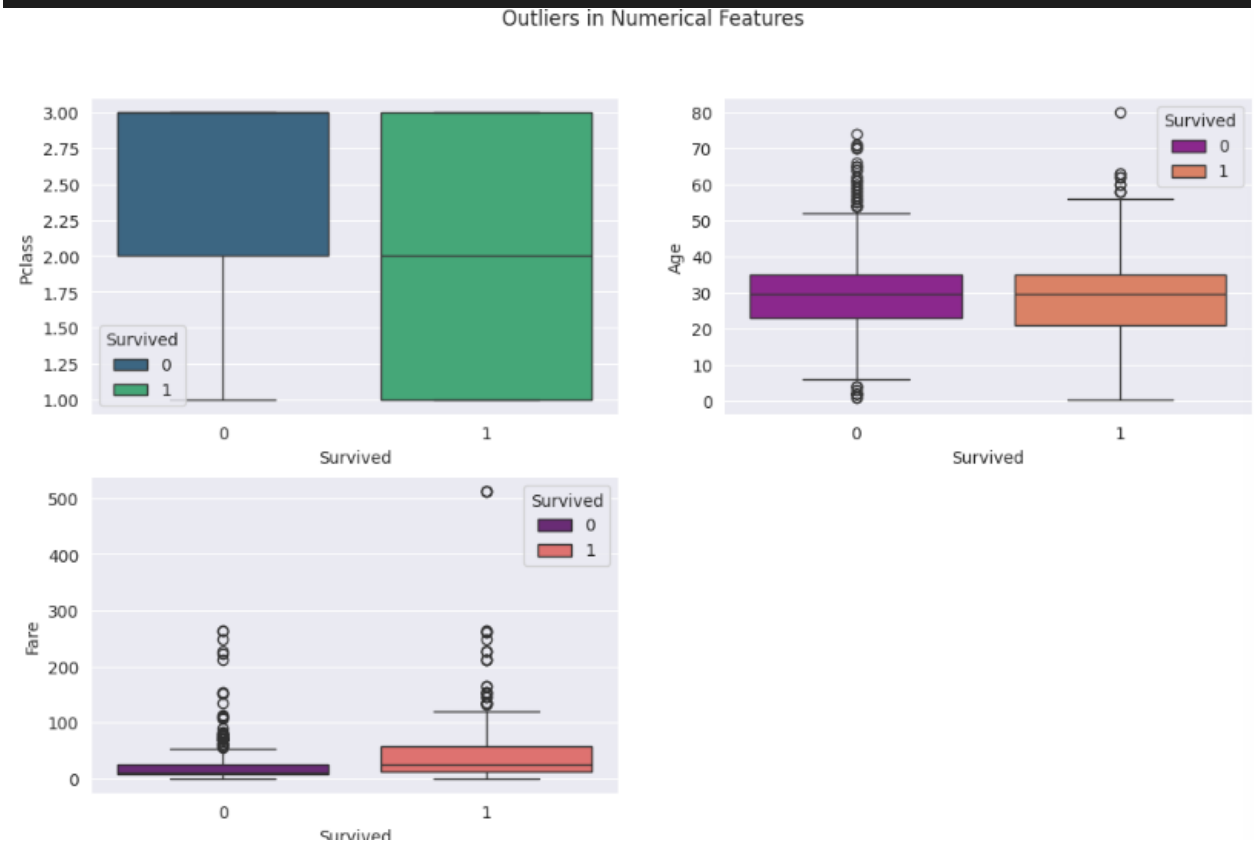
- The 'Embarked' column contains categorical values ('S', 'C', and others).
- The code defines a custom function encode_embarked to map each category to a numerical value.
- It iterates through the 'Embarked' column and replaces 'S' with 0, 'C' with 1, and other values with 2.
- The function returns the modified 'Embarked' column.

Handling Outliers:

- Identify and handle outliers in the data.
  Common techniques include removing outliers, transforming data, or applying robust methods.
- Visualization tools like box plots can help identify outliers.

```
# Checking for Outliers using Box Plots
figure, axes = plt.subplots(2, 2, figsize=(12.5, 7.5))
figure.suptitle("Outliers in Numerical Features")
figure.delaxes(axes[1][1])  # Removing the unused subplot

# Create box plots
sns.boxplot(ax=axes[0, 0], data=df, x="Survived", y="Pclass",
hue="Survived", palette="viridis")
```

```
sns.boxplot(ax=axes[0, 1], data=df, x="Survived", y="Age", hue="Survived",
palette="plasma")
sns.boxplot(ax=axes[1, 0], data=df, x="Survived", y="Fare",
hue="Survived", palette="magma")

# Show the plot
plt.show()
```



This Block of the code visually explores the presence of outliers in 'Pclass', 'Age', and 'Fare' concerning the 'Survived' column, utilizing different colours for the 'Survived' and 'Not Survived' categories. Box plots are effective tools for understanding the central tendency, spread, and potential outliers in a dataset.

• Pclass: Being a categorical variable with three distinct levels (1, 2, and 3), the concept of outliers does not apply to Pclass. The distribution across these categories did not reveal any anomalies warranting concern.

• Age: The distribution of Age showed data points extending into the higher age range, above 60 years. These points are considered valid representations of the age diversity among passengers rather than outliers.

• Fare: While the Fare feature displayed some high-value data points, such variations in fare prices are plausible, reflecting the range of ticket prices aboard the Titanic. These high-value points are deemed reasonable within the context of the dataset.

Given these observations, we concluded that the dataset does not contain significant outliers in the analyzed features.

Scaling Numerical Features:

- Scale numerical features to a similar range to prevent some features from dominating others.
- Common scaling methods include Min-Max scaling or standardization (z-score normalisation).
- Scaling is often important for distance-based algorithms or those sensitive to the scale of features

Handling Categorical Features:

- For algorithms that can handle categorical data directly, one-hot encoding may be preferred over label encoding.
- One-hot encoding creates binary columns for each category.

Feature Engineering:

- Create new features from existing ones if it improves the model's performance.
- For example, combining 'SibSp' and 'Parch' to create a 'FamilySize' feature

```
# Feature Engineering
df['FamilySize'] = df['SibSp'] + df['Parch'] + 1

# Removing 'SibSp' and 'Parch'
df = df.drop(['SibSp', 'Parch'], axis=
```

Splitting the Data:

- Split the dataset into training, validation, and test sets.
- Common split ratios are 70-20-10 or 80-20.

```
# Define the target variable and features
X = df.drop('Survived', axis=1)
y = df['Survived']

print("Features:", "\n", X.head(10))
print("Target Feature:", "\n", y.head(10))
```

```
# Split the data into train (70%), validation (20%), and test (10%) sets
using train_test_split
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,
random_state=21)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp,
test_size=0.33, random_state=21)

# Checking the shape of the splits
print(f"Train shape: {X_train.shape}, Validation shape: {X_val.shape},
Test shape: {X_test.shape}")
print(f"Train shape: {y_train.shape}, Validation shape: {y_val.shape},
Test shape: {y_test.shape}")
```

Task_2 Implement Entropy as a cost function–

```
def calculate_entropy(y):
    class_labels = np.unique(y)
    entropy = 0
    for cls in class_labels:
        p = len(y[y == cls]) / len(y)
        entropy -= p * np.log2(p)
    return entropy

#Entropy as the cost function to calculate the split.
survived_column = df['Survived'].values  # Extract the 'Survived' column
titanic_entropy = calculate_entropy(survived_column)  # Calculate the
entropy

print(f"Entropy of the 'Survived' column in the Titanic dataset:
{titanic_entropy:.3f}")
```

a function **calculate_entropy(y)** to compute the entropy of a target variable, and then it calculates and prints the entropy of the 'Survived' column in the Titanic dataset. Entropy is a measure of impurity or disorder in a set of labels. Lower entropy indicates a more homogeneous set.

This code computes the entropy for the 'Survived' column in the Titanic dataset using the calculate_entropy function. It avoids the issue of taking the logarithm of 0 by checking if p is greater than 0 before performing the calculation. The result is then printed with three decimal places.

Task_3 - Implementing conTocat() function–decision function that makes the split.

```
def conTocat(data, continuous_features, target):
    for feature in continuous_features:
        best_split = None
        best_info_gain = 0
        unique_values = data[feature].unique()

        for split_value in unique_values:
            # Create temporary binary feature based on the split
            data['temp_split'] = (data[feature] >=split_value).astype(int)

            # Calculate information gain
            new_entropy = entropy_of_split(data, 'temp_split', target)
            base_entropy = calculate_entropy(data[target])
            info_gain = base_entropy - new_entropy

            if info_gain > best_info_gain:
                best_info_gain = info_gain
                best_split = split_value

        # Apply the best split to the feature
        data[feature] = (data[feature] >= best_split).astype(int)

    # Drop the temporary split column
    if 'temp_split' in data.columns:
        data.drop(columns=['temp_split'], inplace=True)

    return data

# Example usage
continuous_features = ['Age', 'Fare'] # List of continuous features
df = conTocat(df, continuous_features, 'Survived') # Assuming 'Survived'
is the target column
print(df)
```

Loop over Continuous Features:
It iterates through each continuous feature provided in the continuous_features list.

Find Best Split:
For each feature, it finds the best split point by iterating through unique values of that feature. It calculates the information gain for each split.

Create Temporary Binary Feature:
It creates a temporary binary feature called 'temp_split' based on the identified split point. If the original feature value is greater than or equal to the split point, the temporary feature is set to 1; otherwise, it is set to 0.

Calculate Information Gain:
It calculates the information gain based on the temporary split feature and the target variable.

Apply Best Split:
It applies the best split point to the original feature by converting it into a binary feature. Values greater than or equal to the best split point become 1, and others become 0.

Drop Temporary Column:
It drops the temporary split column created during the process.

Return Modified DataFrame:
The function modifies the original DataFrame by applying the best splits to the specified continuous features and returns the modified DataFrame.

Task_4- Generating Decision Tree
Decision Tree-
Several efficient algorithms have been developed to construct a decision tree for a given dataset in a reasonable amount of time. These algorithms usually employ a greedy strategy: which means that the tree grows by making a series of locally optimum decisions about which attribute to use for partitioning the data creating new split condition nodes. Examples of greedy algorithms: are Hunt's algorithm, ID3, C4.5, and CART.
I will use ID3 as the basic recursive partitioning algorithm, a foundational method for constructing decision trees.
The ID3 algorithm, developed by Ross Quinlan, is one of the foundational algorithms for constructing decision trees. It uses the concept of information gain to decide how to split the dataset at each node. Information gain measures the effectiveness of a split in reducing uncertainty about the target variable.

Calculating Information Gain

Information gain measures the reduction in entropy or disorder when a dataset is split on a specific feature. The goal is to maximize information gain, which leads to the most informative splits and ultimately a more accurate model.

Information Gain Calculation Information gain is defined as the difference between the initial entropy and the weighted sum of entropies of the two subsets. The function is outlined as follows:

```
def information_gain(total, left_split, right_split):
    total_len = len(total)
    if total_len == 0 or len(left_split) == 0 or len(right_split) == 0:
        return 0

    total_entropy = calculate_entropy(total)
    left_entropy = calculate_entropy(left_split)
    right_entropy = calculate_entropy(right_split)

    left_weight = len(left_split) / total_len
    right_weight = len(right_split) / total_len

    weighted_entropy = left_weight * left_entropy + right_weight * right_entropy
    return total_entropy - weighted_entropy

print(df[['Age', 'Fare', 'Survived']].head())
```

Note: A higher information gain indicates a more effective split, reducing un certainty about the target variable more significantly

Selecting the Best Attribute for Split

This process involves evaluating each attribute (except the target attribute) to determine which one results in the highest information gain. The attribute that maximizes this criterion is chosen for the split

Best Attribute Selection Algorithm

he algorithm iterates over all possible attributes, calculates the information gain for splitting the dataset based on each attribute, and selects the attribute that offers the highest information gain. The code for this process is as follows:

```
def best_attribute_for_split(data, attributes, target_attribute):
    best_attribute = None
    max_info_gain = -1

    for attribute in attributes:
        if attribute != target_attribute:
            info_gain = information_gain(data[target_attribute],
                                         data[data[attribute] == 0][target_attribute],
                                         data[data[attribute] == 1][target_attribute])
            if info_gain > max_info_gain:
```

```
                max_info_gain = info_gain
                best_attribute = attribute

    return best_attribute, max_info_gain
```

Explanation

The function best attribute for split takes three parameters: the dataset, a list of attributes, and the target attribute. It initializes variables to store the best attribute found so far and the maximum information gain associated with that attribute. It then iterates over each attribute, excluding the target attribute. For each attribute, it calculates the information gain obtained by splitting the dataset into two subsets: one where the attribute's value is 0 and another where it's 1. This calculation is done using the previously defined information gain function. If the information gain from splitting on the current attribute exceeds the highest information gain found so far, the function updates the best attribute and the maximum information gain accordingly. After evaluating all attributes, the function returns the attribute that results in the highest information gain and the value of that information gain.

Dataset Splitting Based on Attribute

```
def split_dataset(data, attribute):
    return data[data[attribute] == 0], data[data[attribute] == 1]
```

The function split dataset is designed to partition the dataset into two subsets based on the value of a specified attribute. It takes two parameters: the dataset and the attribute on which to split. The attribute is expected to have binary values (0 or 1)

Building the Decision Tree

Tree Construction Algorithm:-

```
def build_tree(data, attributes, target_attribute, depth=0, max_depth=5):
    # Check if stopping criteria are met

    if depth == max_depth or data[target_attribute].nunique() == 1:
        return data[target_attribute].mode()[0]

    # Find the best attribute to split
    best_attr, info_gain = best_attribute_for_split(data, attributes, target_attribute)

    # Check if there's no information gain
    if info_gain <= 0:
        return data[target_attribute].mode()[0]

    # Split the dataset and build subtrees
    tree = {}
    left_data, right_data = split_dataset(data, best_attr)
    tree[best_attr] = {
        0: build_tree(left_data, attributes, target_attribute, depth + 1, max_depth),
        1: build_tree(right_data, attributes, target_attribute, depth + 1, max_depth)
```

```
    }

    return tree

print(df.columns)

def train_decision_tree(data, target_attribute, max_depth=5):
    attributes = list(data.columns)
    attributes.remove(target_attribute)
    return build_tree(data, attributes, target_attribute, max_depth=max_depth)

print(df.columns)
decision_tree = train_decision_tree(df, 'Survived', max_depth=2)
```

- **build_tree is a recursive function that constructs the decision tree. It checks stopping criteria, finds the best attribute to split, and recursively builds subtrees for the left and right splits.**
- **train_decision_tree is a wrapper function that initiates the decision tree construction by calling build_tree.**
- **The decision tree is trained on the df DataFrame with the target attribute 'Survived' and a maximum depth of 2**

The function build tree constructs the decision tree in a recursive manner. It takes the dataset, the list of attributes, the target attribute, the current depth of the tree, and the maximum depth of the tree as parameters.

1. Stopping Criteria: The recursion stops if any of these conditions are met: the tree reaches the maximum specified depth, or the dataset at the current node has the same value for the target attribute (indicating a pure subset).
2. Best Attribute Selection: At each step, the algorithm selects the attribute that provides the highest information gain for splitting the dataset.
3. Dataset Splitting: Once the best attribute is identified, the dataset is split into two subsets based on the attribute's values, and the process is recursively applied to build subtrees for each subset
4. Tree Structure: The decision tree is represented as a nested dictionary, with keys being the attributes used for splitting and the values being either the result of the prediction (at the leaf nodes) or further subtrees.

The train decision tree function initiates the training process by preparing the attributes (excluding the target attribute) and invoking the build tree function with the dataset, the list of attributes, the target attribute, and the maximum depth as parameters

The decision tree model is trained using the train decision tree function, previously defined. The function takes the dataset, the name of the target attribute ('Survived'), and the maximum depth of the tree as arguments. For this demonstration, the maximum depth is set to 2, allowing the tree to make predictions based on a limited number of splits. This depth is chosen to ensure the model's simplicity and interpretability.

Task_5- The infer Function

The infer function takes a sample and the decision tree as inputs. It recursively traverses the tree based on the sample's attribute values until it reaches a leaf 12 node. The value of the leaf node represents the classification of the sample

```
def infer(sample, tree):
    if not isinstance(tree, dict):
        return tree

    attribute = next(iter(tree))

    if attribute in sample:
        attribute_value = sample[attribute]
        if attribute_value in tree[attribute]:
            return infer(sample, tree[attribute][attribute_value])
        else:
            # Return the most common classification among the branches
            branch_counts = {}
            for branch in tree[attribute].values():
                if isinstance(branch, dict):
                    prediction = infer(sample, branch)
                    branch_counts[prediction] = branch_counts.get(prediction, 0) + 1
                else:
                    branch_counts[branch] = branch_counts.get(branch, 0) + 1
            return max(branch_counts, key=branch_counts.get)
    else:
        # If attribute is not present in the sample, handle it as a missing case
        return max(tree[attribute].values(), key=lambda x: (isinstance(x, dict), x))

# Example usage
sample = df.iloc[0]  # Replace with your sample data
classification = infer(sample, decision_tree)
print("The classification is:", classification)
```

The infer function illustrates the flexibility of decision trees in dealing with various types of data and scenarios, including missing attributes and unforeseen attribute values. By considering the most common classification among the branches when encountering an unexpected attribute value, the function ensures a reasoned prediction even in the absence of direct matches in the tree

Task_6-Evaluating Decision Tree Performance

The calculate The calculate accuracy Function accuracy function assesses the decision tree's performance by comparing the predicted labels against the actual labels for a dataset.

```
def calculate_accuracy(data, labels, tree):
    correct_predictions = 0
    tot_0 = 0
    correct_0 = 0
    tot_1 = 0
    correct_1 = 0

    for index, sample in data.iterrows():
        prediction = infer(sample, tree)
        if prediction == labels[index]:
            correct_predictions += 1
        if(labels[index] == 0):
          tot_0 += 1
          if(labels[index] == prediction):
            correct_0 += 1

        if(labels[index] == 1):
          tot_1 += 1
          if(labels[index] == prediction):
            correct_1 += 1

    class0acc = correct_0 / tot_0
    class1acc = correct_1 / tot_1

    accuracy = correct_predictions / len(data)
    return accuracy,class0acc,class1acc
```

Accuracy Results
The model achieved an overall training accuracy of 78.65% and a test accuracy of 80.89%, indicating its effectiveness in predicting survival in the Titanic Wreck.

Task 7 : Confusion Matrix Analysis

To further evaluate the predictive performance of the decision tree model, a con fusion matrix was generated for the test dataset. The confusion matrix provides a visual and quantitative representation of the model's predictions in compari son to the actual outcomes, offering insights into the accuracy, specificity, and sensitivity of the model.

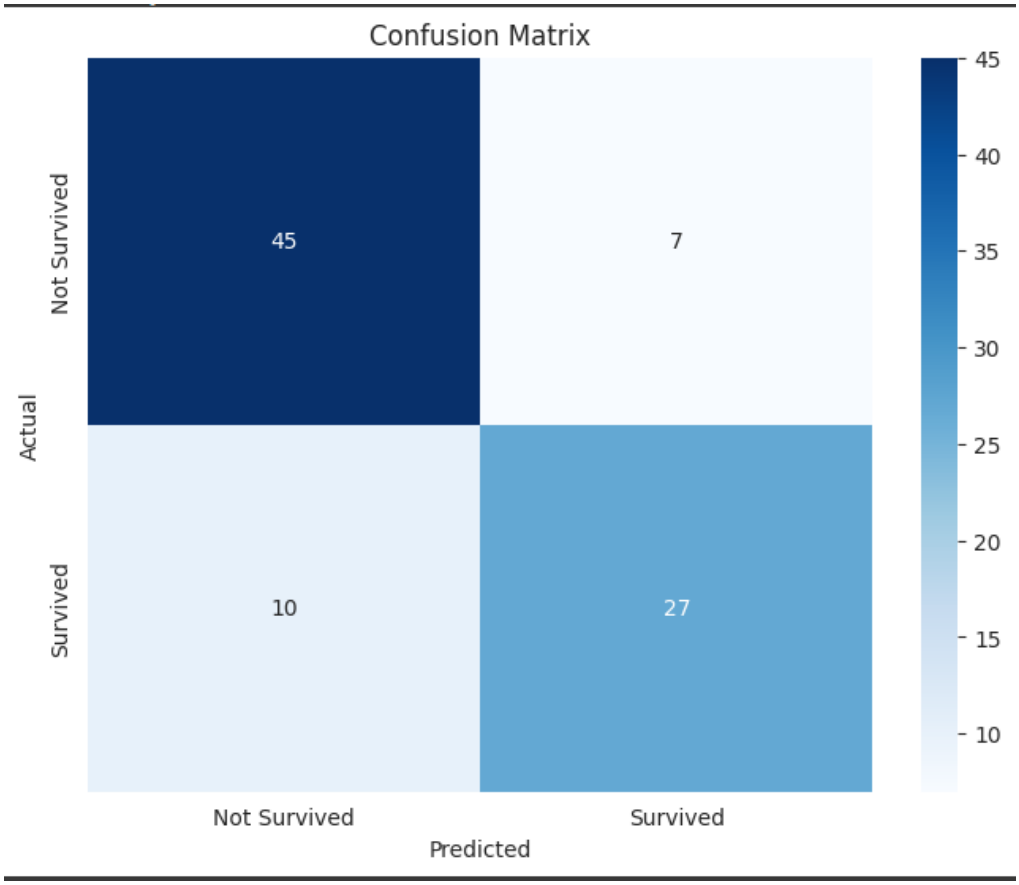
```
# Generate predictions for the test set
y_test_pred = []
for _, row in X_test.iterrows():
    prediction = infer(row.to_dict(), decision_tree)
    y_test_pred.append(prediction)

# Compute the confusion matrix
cm = confusion_matrix(y_test, y_test_pred)

# Plot the confusion matrix
```

```
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues', xticklabels=['Not Survived',
'Survived'], yticklabels=['Not Survived', 'Survived'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()

precision, recall, f1_score, _ = precision_recall_fscore_support(y_test, y_test_pred)
```

This process involves traversing the decision tree based on the attributes of each sample until a prediction is made at a leaf node

The confusion matrix was computed using the confusion matrix function from the scikit-learn library, comparing the predicted labels with the actual labels of the test set.

The confusion matrix provides valuable insights into the decision tree model's classification accuracy. The number of true positives and true negatives indi cates the model's ability to correctly predict survivors and non-survivors, respec tively. Meanwhile, the presence of false positives and false negatives highlights instances where the model's predictions deviated from the actual outcomes.



Task_8-Performance Evaluation Metrics-

Class-wise Performance Metrics
The calculated class-wise metrics are as follows:
• Precision for Class 0 (Not Survived) is `0.8181818181818182`, and for Class 1 (Survived) it is `0.7941176470588235`.
• Recall for Class 0 is `0.8653846153846154`, and for Class 1 it is `0.8653846153846154`
• F1-ScoreforClass 0is `0.8653846153846154`, and for Class 1 it is `0.8653846153846154`.

Average Performance Metrics
• Average Precision: `0.8653846153846154`
• Average Recall: `0.7975571725571726`
• Average F1-Score: `0.8008424378043965`

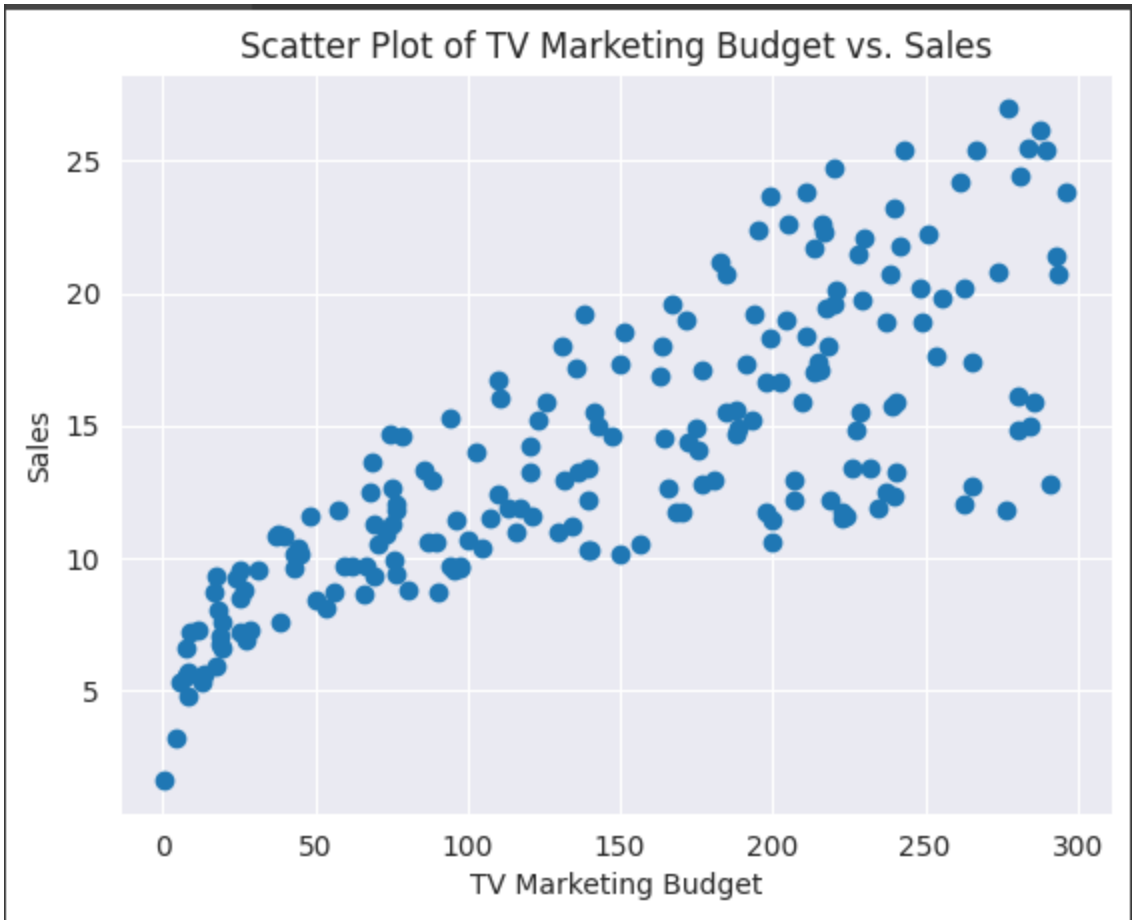
Qno2-Linear Regression Implementation

Task-1: Dataset Exploration
(a) Loading the dataset and displaying the first few rows.

The dataset was loaded from the provided URL, and the first few rows were displayed.

```
url =
"https://raw.githubusercontent.com/devzohaib/Simple-Linear-Regression/master/tvmarketing.csv"
df = pd.read_csv(url)
print("First few rows of the dataset:")
display(df.head())
```

(b) Scatter Plot Analysis
A scatter plot was created to visualize the relationship between the TV marketing budget and sales.



Comment on the trend observed in the scatter plot:
The scatter plot shows a positive linear trend, indicating that as the TV marketing budget increases, sales also tend to increase.

```
plt.scatter(df['TV'], df['Sales'])
plt.xlabel('TV Marketing Budget')
plt.ylabel('Sales')
plt.title('Scatter Plot of TV Marketing Budget vs. Sales')
plt.show()
```

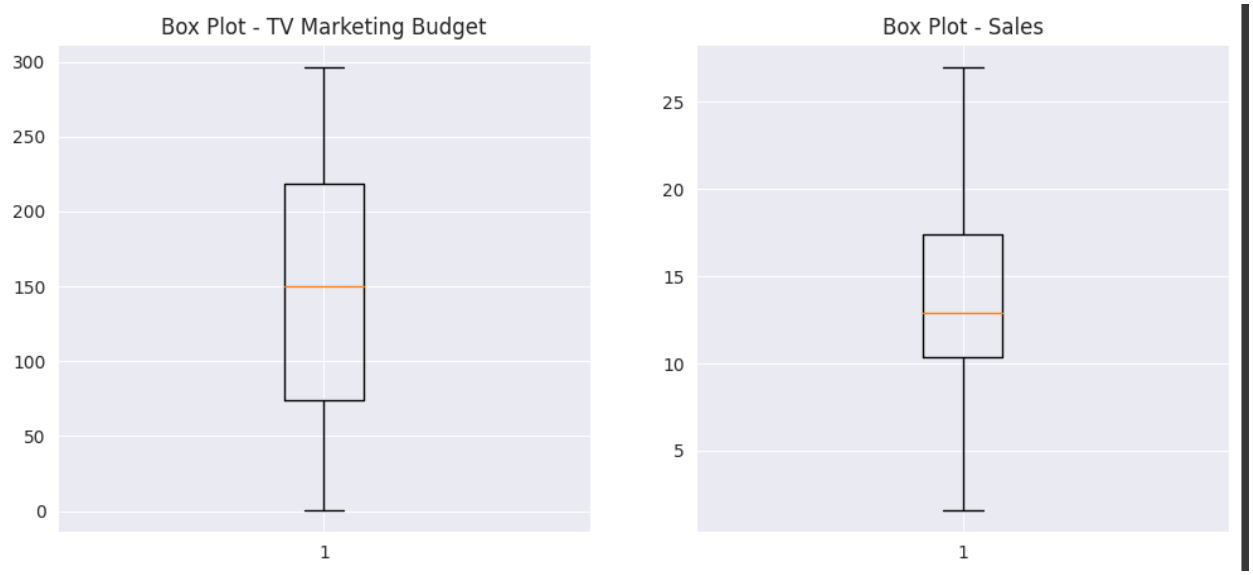
(c) Statistical Measures
Basic statistical measures (mean and standard deviation) for both TV marketing budget and sales were calculated and displayed.

```
tv_mean = df['TV'].mean()
tv_std = df['TV'].std()
sales_mean = df['Sales'].mean()
sales_std = df['Sales'].std()

# Graphical representation of statistical measures
```

```
fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# Bar graph for TV Marketing Budget mean and standard deviation
axs[0].bar(['Mean', 'Std Dev'], [tv_mean, tv_std], color=['blue', 'orange'])
axs[0].set_title('TV Marketing Budget')
axs[0].set_ylabel('Values')

# Bar graph for Sales mean and standard deviation
axs[1].bar(['Mean', 'Std Dev'], [sales_mean, sales_std], color=['blue', 'orange'])
axs[1].set_title('Sales')
axs[1].set_ylabel('Values')

plt.tight_layout()
plt.show()
```

Checking for Outliers-

```
# Visualize outliers using box plots
fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# Box plot for 'TV'
axs[0].boxplot(df['TV'])
axs[0].set_title('Box Plot - TV Marketing Budget')

# Box plot for 'Sales'
axs[1].boxplot(df['Sales'])
axs[1].set_title('Box Plot - Sales')
```



No Outliers Present

Task-2: Data Preprocessing

(a) Checking for Missing Values

Missing values in the dataset were checked and displayed.

```
missing_values = df.isnull().sum()
print("Missing values in the dataset:")
display(missing_values)
```

(b) Normalizing Data

The TV marketing budget and sales columns were normalized using Min-Max scaling.

```
scaler = MinMaxScaler()  # Use MinMaxScaler
df[['TV', 'Sales']] = scaler.fit_transform(df[['TV', 'Sales']])
```

Histograms were plotted for the normalized data to visualize the distribution.

(c) Splitting the Dataset

The dataset was split into training and testing sets using an 80-20 split.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=21)

# Print the shapes of the training and testing sets
print("\nShapes of the training and testing sets:")
print(f"X_train shape: {X_train.shape}, y_train shape: {y_train.shape}")
print(f"X_test shape: {X_test.shape}, y_test shape: {y_test.shape}")
```

Task-3: Linear Regression Implementation

(a) Hypothesis Function and Gradient Descent

The hypothesis function for linear regression (y = w1x + w0) was implemented using Gradient Descent.

The mean squared error (MSE) cost function was used for optimization.

```
def hypothesis(X, w0, w1):
    """Hypothesis function for linear regression."""
    return w0 + w1 * X

def mean_squared_error(y_true, y_pred):
    """Calculate mean squared error."""
```

```
    return np.mean((y_true - y_pred) ** 2)

def gradient_descent(X, y, w0, w1, learning_rate, epochs):
    """Gradient Descent for linear regression."""
    m = len(X)
    for epoch in range(epochs):
        # Calculate predictions and errors
        predictions = hypothesis(X, w0, w1)
        errors = predictions - y

        # Update weights
        w0 -= learning_rate * (1/m) * np.sum(errors)
        w1 -= learning_rate * (1/m) * np.sum(errors * X)

        # Calculate and print mean squared error for every 100 epochs
        if epoch % 100 == 0:
            mse = mean_squared_error(y, predictions)
            print(f"Epoch {epoch}, MSE: {mse}")

    return w0, w1
```

Optimal value of Learning rate and Epoch-(0.1,700)

```
Epoch 0, MSE: 0.2811771498542997
Epoch 100, MSE: 0.018933554131361333
Epoch 200, MSE: 0.01699025652610584
Epoch 300, MSE: 0.01647800668889496
Epoch 400, MSE: 0.016342978531047013
Epoch 500, MSE: 0.016307385345676603
Epoch 600, MSE: 0.016298003044495846
```

As after 600 there is a constancy in the mse value

(b) Regression Line Plot

The regression line was plotted on the scatter plot from Task-1.

```
# Plot the scatter plot with the regression line
plt.scatter(X_original, y_original, label='Actual Data')
plt.plot(X_original, predictions_optimal, color='red', label='Regression
Line')
plt.xlabel('TV Marketing Budget')
plt.ylabel('Sales')
```

```
plt.title('Linear Regression with Gradient Descent')
plt.legend()
plt.show()
```

Explanation:

- The hypothesis function represents the predicted values (y_pred) based on weights w0 and w1.
- Gradient Descent was employed to minimize the mean squared error by adjusting weights (w0 and w1).
- The regression line was plotted using the original TV marketing budget (X_original) and the predicted values.

Error Parameters

- Mean Squared Error (MSE): It measures the average of the squared differences between predicted and actual values. Lower MSE indicates a better fit.
- Mean Absolute Error (MAE): It calculates the average absolute differences between predicted and actual values. It provides a measure of the model's accuracy.

Task-4: Evaluation

The trained model was evaluated on the test set, and MSE and MAE were calculated.

```
mse_test = mean_squared_error(y_test, predictions_test)
mae_test = mean_absolute_error(y_test, predictions_test)
```

Evaluation Metrics:

```
Regression Line: y = 0.22 + 0.55 * x
Mean Squared Error on Test Set: 0.0159348233927403
Mean Absolute Error on Test Set: 0.09577726198653644
```

Conclusion

- The scatter plot and statistical measures provided insights into the dataset.
- Min-Max scaling was applied for normalization to ensure uniformity in the range of features.
- The linear regression model was implemented using Gradient Descent, and the regression line was visualized on the scatter plot.
- Evaluation metrics (MSE and MAE) were used to assess the model's performance on the test set.

Qno_3-Linear Regression Model Application on Boston Housing Data

Data Exploration And Visualization-

(a)Loading and Displaying the First Few Rows and some other data of the Boston Housing Dataset

```
url='https://raw.githubusercontent.com/SK-190803/PRML_Assignment-data/main/DATA/bostonHousingData.csv'
data_boston=pd.read_csv(url,encoding='unicode_escape')


# Task-1: Dataset Exploration
# (a) Display the first few rows of the Boston Housing dataset
print("First few rows of the Boston Housing dataset:")
print(data_boston.head())
display(data_boston)
display(data_boston.info())
display(data_boston.describe())
```

(b) Scatter Plots to Visualize Relationships-
Scatter plots are created to visually inspect the relationships between selected features (RM, NOX, CRIM, DIS, PTRATIO) and the target variable (MEDV).

The selected features represent various aspects that are known to impact housing prices, including the size of the property (RM), environmental factors (NOX), safety (CRIM), location (DIS), and education-related factors (PTRATIO). Analyzing these relationships through scatter plots helps uncover patterns and trends that contribute to a better understanding of the dataset

Further,information related to basic statistical parameters about the above mentioned features is also there in the output of the code along with its pictorial representation(bar graphs)

Task-2: Data Preprocessing

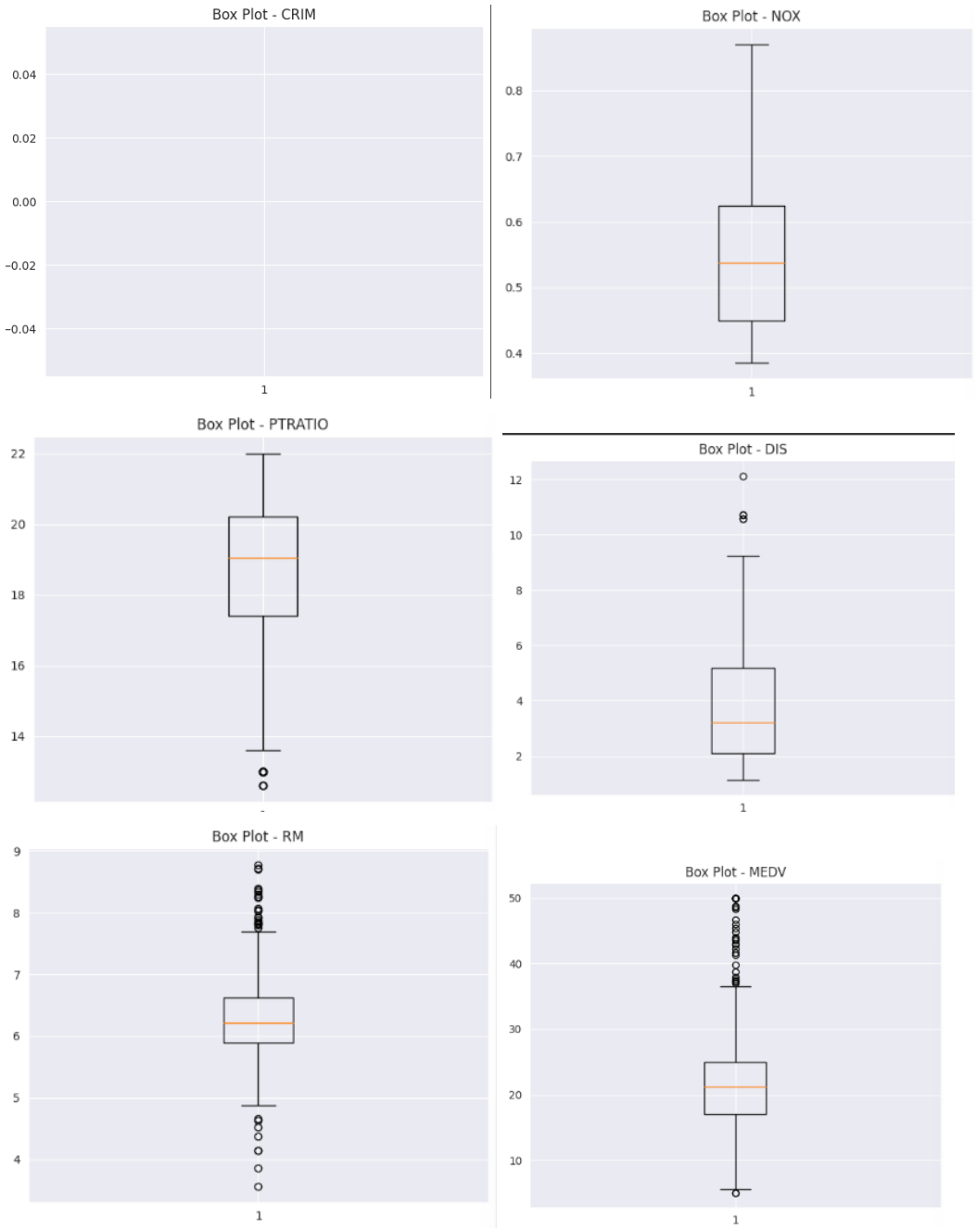
(a) Checking for Missing Values and Handling Appropriately

```
missing_values_boston = data_boston.isnull().sum()
print("Missing values in the Boston Housing dataset:")
print(missing_values_boston)

columns_with_missing_values = ['CRIM', 'ZN', 'INDUS', 'CHAS','AGE', 'LSTAT']
# Impute missing values with mean
for column in columns_with_missing_values:
    data_boston[column].fillna(data_boston[column].mean(), inplace=True)
```

Missing values are checked for and handled by imputing with the mean for the specified columns ('CRIM', 'ZN', 'INDUS', 'CHAS', 'AGE', 'LSTAT').

(b) Checking for Outliers

```
for feature in features:
    # Box plot for the feature
    axs[1].boxplot(data_boston[feature])
    axs[1].set_title(f'Box Plot - {feature}')

    plt.tight_layout()
    plt.show()
```



Reason for CRIM Box Plot to be Empty-
'CRIM' values are on a different scale compared to other features, the default settings for the box plot might not identify any points as outliers. Adjusting the scale or using a different outlier detection method could reveal potential outliers.

(c) Normalizing Columns

```
min_max_scaler = lambda x: (x - np.min(x)) / (np.max(x) - np.min(x))
data_boston[['RM', 'MEDV']] = data_boston[['RM', 'MEDV']].apply(min_max_scaler)
# Min-max normalization for 'CRIM', 'NOX', 'PTRATIO', 'DIS'
columns_to_normalize = ['CRIM', 'NOX', 'PTRATIO', 'DIS']
data_boston[columns_to_normalize] = data_boston[columns_to_normalize].apply(min_max_scaler)
# Display the normalized data
display(data_boston)
display(data_boston.isnull().sum())
```

The 'RM' and 'MEDV' columns are normalized using min-max normalization. Additionally, 'CRIM', 'NOX', 'PTRATIO', and 'DIS' columns are normalized.
Reason to use Min-Max-Presence of Outliers in some of the features

(d) Histograms for Normalized DataFields-
The resulting set of histograms gives a visual representation of the distribution of normalized values for each selected feature in the Boston Housing datase

(e) Splitting Data into train-test:-
The dataset is split into training and testing sets using an 80-20 split

Task_3-Implementing Multivariate Linear Regression-

```
# Define the hypothesis function for multivariate linear regression
def hypothesis(X, theta):
    return np.dot(X, theta)

# Define the mean squared error (MSE) cost function
def mean_squared_error(X, y, theta):
    m = len(y)
    predictions = hypothesis(X, theta)
    return np.sum((predictions - y) ** 2) / (2 * m)

# Define the gradient descent function for multivariate linear regression
def gradient_descent(X, y, theta, alpha, num_iterations):
    m = len(y)
    cost_history = []

    for _ in range(num_iterations):
        error = hypothesis(X, theta) - y
        gradient = np.dot(X.T, error) / m
        theta -= alpha * gradient
        cost = mean_squared_error(X, y, theta)
        cost_history.append(cost)
```

```
    return theta, cost_history

# Select features and target variable
X_multivariate = data_boston[['RM', 'NOX', 'CRIM', 'DIS', 'PTRATIO']].values
y_multivariate = data_boston['MEDV'].values

# Add a column of ones for the bias term
X_multivariate = np.c_[np.ones(X_multivariate.shape[0]), X_multivariate]

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_multivariate, y_multivariate,
test_size=0.2, random_state=42)

# Initialize parameters
theta_initial = np.zeros(X_train.shape[1])

# Set hyperparameters
alpha = 0.01
num_iterations = 1000

# Run gradient descent
theta_final, cost_history = gradient_descent(X_train, y_train, theta_initial, alpha,
num_iterations)

# Plot scatter plot with regression lines for each feature
features_to_plot = ['RM', 'NOX', 'CRIM', 'DIS', 'PTRATIO']

for feature in features_to_plot:
    # Select the current feature and target variable
    X_feature = data_boston[[feature]].values
    y_feature = data_boston['MEDV'].values

    # Add a column of ones for the bias term
    X_feature = np.c_[np.ones(X_feature.shape[0]), X_feature]

    # Split the dataset into training and testing sets
    X_train_feature, X_test_feature, y_train_feature, y_test_feature =
train_test_split(X_feature, y_feature, test_size=0.2, random_state=42)

    # Initialize parameters
    theta_initial_feature = np.zeros(X_train_feature.shape[1])
# Set hyperparameters
    alpha_feature = 0.01
    num_iterations_feature = 1000
# Run gradient descent
    theta_final_feature, _ = gradient_descent(X_train_feature, y_train_feature,
theta_initial_feature, alpha_feature, num_iterations_feature)
```

The provided code implements multivariate linear regression using gradient descent on the Boston Housing dataset. Here's a breakdown of the code:

Hypothesis Function:

- The hypothesis function takes input features X and parameters theta and calculates the dot product, representing the predicted values.

Mean Squared Error (MSE) Cost Function:

- The mean_squared_error function computes the mean squared error between the predicted values and the actual target variable y.

Gradient Descent Function:

- The gradient_descent function performs gradient descent optimization to update the parameters theta iteratively, minimizing the mean squared error.

Selecting Features and Target Variable:

- Features (X_multivariate) include 'RM', 'NOX', 'CRIM', 'DIS', and 'PTRATIO'.
- Target variable (y_multivariate) is 'MEDV' (House Price).

Adding Bias Term:

- A column of ones is added to the features (X_multivariate) to account for the bias term in the linear regression model.

Splitting the Dataset:

- The dataset is split into training and testing sets using an 80-20 split.

Initializing Parameters and Hyperparameters:

- Initial parameters (theta_initial) are set to zeros.
- Hyperparameters include the learning rate (alpha) and the number of iterations for gradient descent.

Running Gradient Descent:

- Gradient descent is executed on the training set (X_train, y_train) to obtain the final parameters (theta_final) and track the cost history.

Plotting Regression Lines:

- For each feature in features_to_plot ('RM', 'NOX', 'CRIM', 'DIS', 'PTRATIO'), a scatter plot with regression lines is created.
- The regression lines are plotted for both training and testing data using the parameters obtained from gradient descent.

Task_4-Evaluation-

This section of the code provides a comprehensive view of the trained multivariate linear regression model's performance, including the regression equation and evaluation metrics on the test set.

```
# Plot the scatter plot with regression line
    plt.scatter(X_train_feature[:, 1], y_train_feature, label=f'Training Data ({feature})',
alpha=0.7)
    plt.scatter(X_test_feature[:, 1], y_test_feature, label=f'Testing Data ({feature})',
marker='x', alpha=0.7)
    plt.xlabel(feature)
    plt.ylabel('House Price (MEDV)')
    plt.title(f'Scatter Plot with Regression Line ({feature} vs. MEDV)')

    # Plot the regression line for training data in different colors
    plt.plot(X_train_feature[:, 1], hypothesis(X_train_feature, theta_final_feature),
label=f'Training Regression Line ({feature})')

    # Plot the regression line for testing data in different colors
    plt.plot(X_test_feature[:, 1], hypothesis(X_test_feature, theta_final_feature),
label=f'Testing Regression Line ({feature})', linestyle='--')

    plt.legend()
    plt.show()


# Extract coefficients
theta_intercept = theta_final[0]
theta_RM = theta_final[1]
theta_NOX = theta_final[2]
theta_CRIM = theta_final[3]
theta_DIS = theta_final[4]
theta_PTRATIO = theta_final[5]

# Print the regression equation
print(f'Regression Line: MEDV = {theta_intercept:.2f} + {theta_RM:.2f} * RM +
{theta_NOX:.2f} * NOX + {theta_CRIM:.2f} * CRIM + {theta_DIS:.2f} * DIS +
{theta_PTRATIO:.2f} * PTRATIO')


# Predict on the test set using the trained model
y_pred_test = np.dot(X_test, theta_final)

# Calculate Mean Squared Error (MSE)
mse = np.mean((y_pred_test - y_test)**2)

# Calculate Absolute Error (AE)
ae = np.mean(np.abs(y_pred_test - y_test))

print(f'Mean Squared Error (MSE) on the test set: {mse}')
print(f'Absolute Error (AE) on the test set: {ae}')
```

Scatter Plot with Regression Lines:

- For each feature in features_to_plot, scatter plots are created for both training and testing data with regression lines.
- The training data is represented by circles, and testing data is represented by 'x' markers.
- Regression lines are plotted for both training and testing data, each in a different color.

Extracting Coefficients:

- Coefficients (parameters) are extracted from the final parameters obtained after gradient descent.
- theta_intercept represents the intercept, and other theta variables represent coefficients for each feature.

Printing Regression Equation:

- The regression equation is printed, showing the relationship between the target variable ('MEDV') and each feature.

Predicting on the Test Set:

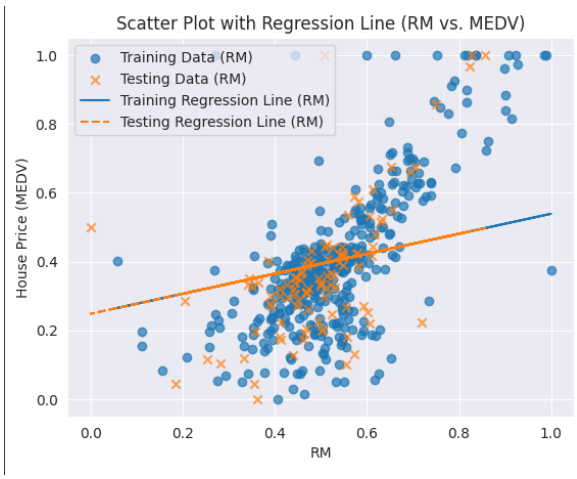
- The trained model is used to make predictions on the test set (X_test), and the predicted values are stored in y_pred_test.

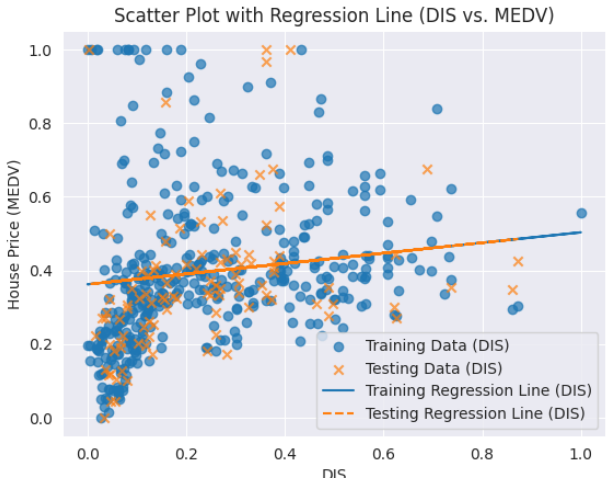
Calculating Evaluation Metrics:

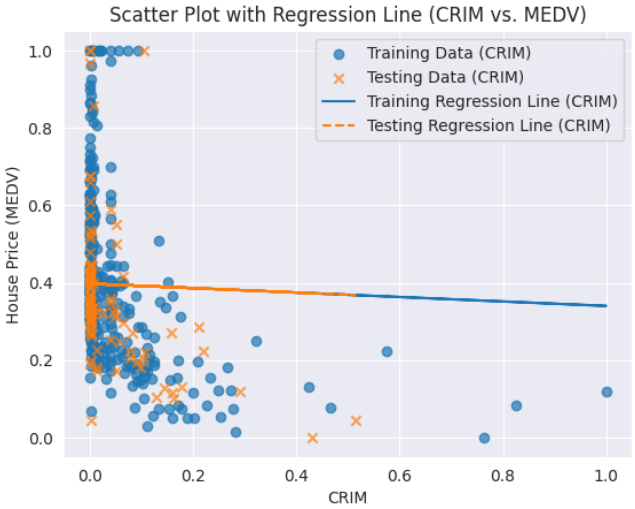
- Mean Squared Error (MSE) and Absolute Error (AE) are calculated to evaluate the performance of the model on the test set.

```
Regression Line:
MEDV=0.26+ 0.30*RM + (-0.06*NOX) + -0.05*CRIM + 0.11*DIS + (-0.06*PTRATIO)

Mean Squared Error (MSE) on the test set: 0.02352350450802887
Absolute Error (AE) on the test set: 0.10672335085503948
```

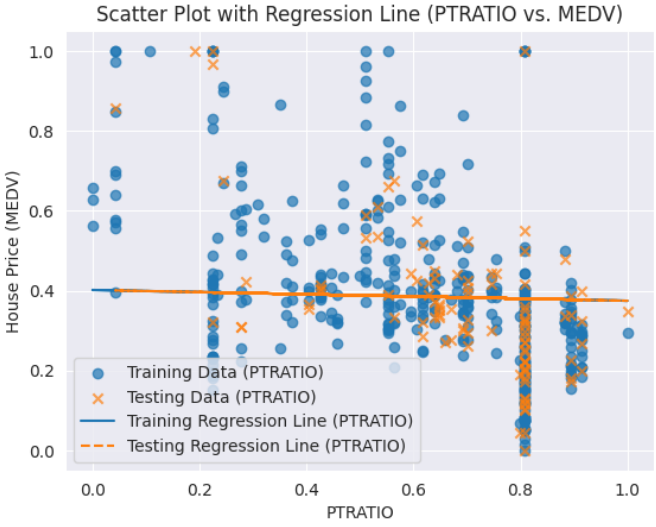
Regression Line Plots-


Scatter Plot with Regression Line (RM vs. MEDV)
Training Data (RM)
Testing Data (RM)
Training Regression Line (RM)
Testing Regression Line (RM)
House Price (MEDV)
RM


Scatter Plot with Regression Line (DIS vs. MEDV)
Training Data (DIS)
Testing Data (DIS)
Training Regression Line (DIS)
Testing Regression Line (DIS)
House Price (MEDV)
DIS


Scatter Plot with Regression Line (CRIM vs. MEDV)
Training Data (CRIM)
Testing Data (CRIM)
Training Regression Line (CRIM)
Testing Regression Line (CRIM)
House Price (MEDV)
CRIM


Scatter Plot with Regression Line (NOX vs. MEDV)
Training Data (NOX)
Testing Data (NOX)
Training Regression Line (NOX)
Testing Regression Line (NOX)
House Price (MEDV)
NOX


Scatter Plot with Regression Line (PTRATIO vs. MEDV)
Training Data (PTRATIO)
Testing Data (PTRATIO)
Training Regression Line (PTRATIO)
Testing Regression Line (PTRATIO)
House Price (MEDV)
PTRATIO