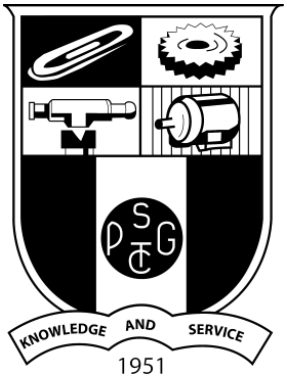


PSG COLLEGE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PSG COLLEGE OF TECHNOLOGY

(Autonomous Institution)

COIMBATORE – 641 004

19Z610 – MACHINE LEARNING LABORATORY

Bonafide work done by
s
R K RANJITH (21Z335)

CERTIFICATE

Certified that this report titled **“19Z610 - Machine Learning Laboratory Report ”** for the 19Z610 Machine Learning Laboratory is a bonafide work of **R K Ranjith(21Z335)** who has carried out the work for the partial fulfillment of the requirements for the award of the degree of Bachelor of Engineering in Computer Science and Engineering.

Place: Coimbatore
Date :

Dr. Lovelyn Rose

CONTENT

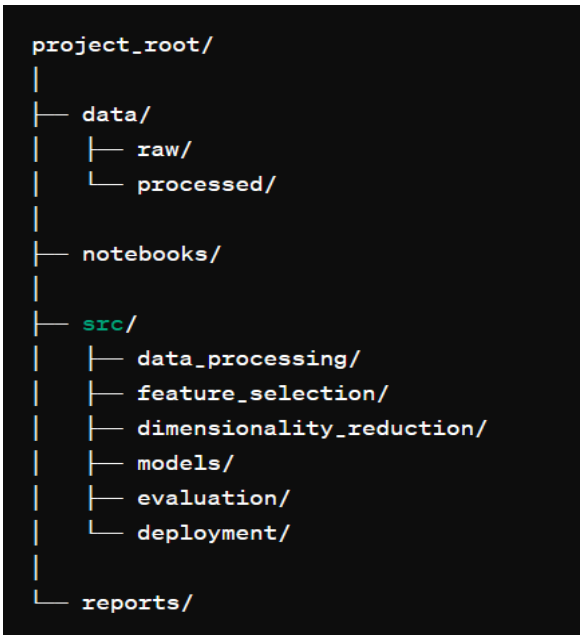
Disclaimer: This report is not copied from anyone and is genuine

CRISP - DM Regression

Objective

The purpose of this report is to detail the deployment of a machine learning pipeline designed for regression tasks. This pipeline encompasses several key stages such as data comprehension, data preprocessing, model building, performance evaluation, hyperparameter optimization, unit and integration testing, as well as deployment. Our goal is to leverage a selection of widely used regression algorithms along with suitable methodologies for handling missing data, selecting relevant features, reducing dimensionality, and assessing model performance.

Directory Structure



Design Patterns

To maintain a scalable and maintainable codebase, the pipeline utilizes design patterns such as:

1. Singleton Pattern: For managing configurations and global resources.
2. Factory Pattern: For creating instances of regressors and other objects dynamically.
3. Strategy Pattern: For interchangeable feature selection and dimensionality reduction techniques.

Pipelines and Abstraction:

To ensure the modularity and reusability of components, the pipeline is organized using pipelines and abstract methods. Each stage of the pipeline, including data preprocessing, feature selection, modeling, and evaluation, is encapsulated within distinct modules or classes, fostering a structured and maintainable codebase.

Linters:

To uphold coding standards and uphold code quality consistently, linting tools are utilized. Tools like Flake8 or Pylint are employed to detect and address issues pertaining to syntax errors, coding style adherence, and potential bugs across the codebase.

Data Understanding:

- The initial step in our data exploration process involves mounting Google Drive to access the dataset stored in the cloud, named 'raw_mpg.csv'. This dataset contains information pertinent to automotive fuel efficiency. Upon loading the dataset using Pandas, we generate a concise summary showcasing its dimensions and preview a sample of its initial rows to grasp its structure and content. Following this, we present summary statistics that highlight key numerical attributes such as mean, standard deviation, and quartile values. These statistics provide valuable insights into the data's distribution and variability, aiding in our understanding of its characteristics.

- Subsequently, we employ data visualization techniques to delve deeper into the dataset. Specifically, we leverage the Matplotlib and Seaborn libraries to create a histogram focusing on the 'mpg' (miles per gallon) variable. This histogram visually represents the distribution of MPG values across the dataset. By customizing parameters such as bin count and enabling kernel density estimation (KDE), we obtain a smoothed approximation of the underlying probability density function. This visualization not only facilitates a rapid assessment of central tendencies and dispersion but also assists in identifying potential data patterns or anomalies.

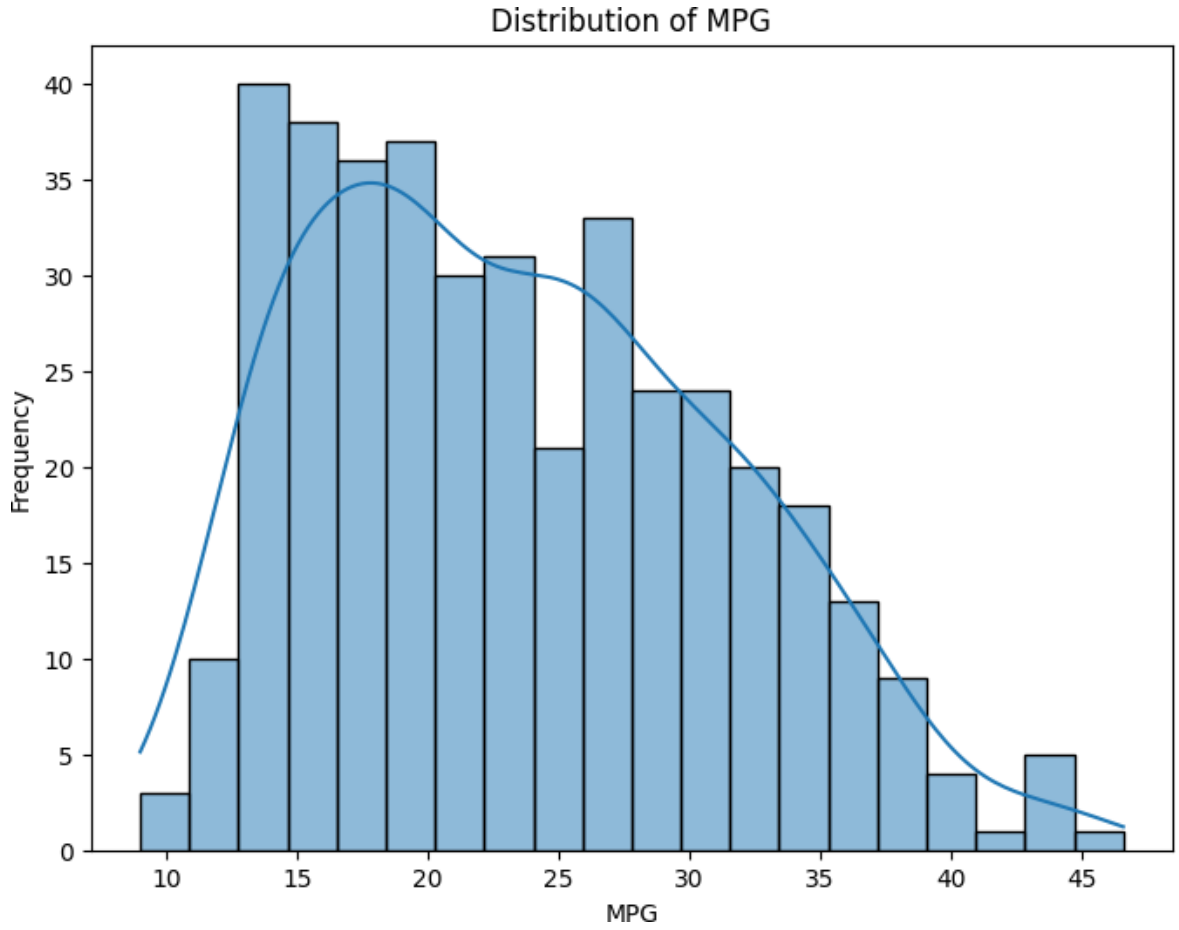
Code:

```
from google.colab import drive
import pandas as pd

# Mount Google Drive
drive.mount('/content/drive')

# Define dataset path
data_path = '/content/drive/My Drive/Dataset/raw_mpg.csv'

# Load the dataset
data = pd.read_csv(data_path)

# Display dataset summary
print("Dataset shape:", data.shape)
print(data.head())

# Display summary statistics
print("Summary Statistics:")
print(data.describe())

# Visualizations (example: Histogram of MPG)
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(8, 6))
sns.histplot(data['mpg'], bins=20, kde=True)
plt.title('Distribution of MPG')
plt.xlabel('MPG')
plt.ylabel('Frequency')
plt.show()
```

Output:

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
Dataset shape: (398, 9)
    mpg  cylinders  displacement horsepower  weight  acceleration  model  year  \
0  18.0          8         307.0        130    3504          12.0         70
1  15.0          8         350.0        165    3693          11.5         70
2  18.0          8         318.0        150    3436          11.0         70
3  16.0          8         304.0        150    3433          12.0         70
4  17.0          8         302.0        140    3449          10.5         70

   origin                   name
0       1  chevrolet chevelle malibu
1       1          buick skylark 320
2       1         plymouth satellite
3       1              amc rebel sst
4       1                ford torino
Summary Statistics:
              mpg   cylinders  displacement      weight  acceleration  \
count  398.000000  398.000000    398.000000  398.000000    398.000000
mean    23.514573    5.454774    193.425879 2970.424623     15.568090
std      7.815984    1.701004    104.269838  846.841774      2.757689
min      9.000000    3.000000     68.000000 1613.000000      8.000000
25%     17.500000    4.000000    104.250000 2223.750000     13.825000
50%     23.000000    4.000000    148.500000 2803.500000     15.500000
75%     29.000000    8.000000    262.000000 3608.000000     17.175000
max     46.600000    8.000000    455.000000 5140.000000     24.800000

       model_year      origin
count  398.000000  398.000000
mean    76.010050    1.572864
std      3.697627    0.802055
min     70.000000    1.000000
25%     73.000000    1.000000
50%     76.000000    1.000000
75%     79.000000    2.000000
max     82.000000    3.000000
```

Graph Visualization:



Data Preparation:

- The initial step in data preparation involves importing essential libraries such as pandas, numpy, and seaborn to facilitate data manipulation and visualization tasks.

- A custom function named outlier_z_score is defined to detect outliers in numerical columns utilizing the Z-Score method, aiding in identifying data points significantly deviating from the mean.

- The outlier_z_score function is then applied to target numerical columns within the dataset, flagging potential outlier rows based on their Z-Score values.

- Following outlier detection, a list containing rows identified as outliers is generated, highlighting the data points requiring further scrutiny or action.

- Subsequently, the identified outlier rows are removed from the dataset, ensuring data integrity and mitigating the potential influence of outliers on subsequent analyses or modeling efforts.

- Key statistical information, such as the lengths of the original dataset and the outlier-free dataset, along with the difference in lengths, is computed and displayed. This provides valuable insights into the impact of outlier removal on the dataset size and distribution.

- Visual representations in the form of boxplots are generated to visualize the distribution of the target variable 'mpg' before and after outlier removal, aiding in understanding the effects of data preprocessing on key variables.

- Handling missing values is addressed by first identifying and replacing placeholder values (e.g., '?' denoting missing data) with NaN (Not a Number) values in relevant columns such as 'horsepower.'

- A utility function named missing_values_table is defined to systematically analyze and summarize missing values across the dataset, facilitating informed decision-making regarding data imputation strategies.

- The KNNImputer is employed to impute missing values specifically in the 'horsepower' column, leveraging the K-Nearest Neighbors algorithm to estimate and fill in missing values based on neighboring data points.

- Additionally, data type conversions and categorical variable preprocessing tasks such as one-hot encoding (if required) are prepared within the codebase, ensuring data readiness for subsequent modeling phases while maintaining flexibility and scalability in data handling processes.

Code:

```
import pandas as pd
import numpy as np
import seaborn as sns

# Assuming df_train is your dataframe
# Read data
# df_train = pd.read_csv('data.csv')

# Outlier function with threshold 2
def outliers_z_score(df):
    threshold = 2

    mean = np.mean(df)
    std = np.std(df)
    z_scores = [(y - mean) / std for y in df] #Used a Z-Score to remove the
outliers
    return np.where(np.abs(z_scores) > threshold)

# Selecting only the numerical columns in data set
my_list = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
num_columns = list(df_train.select_dtypes(include=my_list).columns)
numerical_columns = df_train[num_columns]

# Calling the outlier function and Calculating the outlier of dataset
outlier_list = numerical_columns.apply(outliers_z_score)

# Making outlier list ot dataframe
df_of_outlier = outlier_list.iloc[0]
df_of_outlier = pd.DataFrame(df_of_outlier)
```

```
df_of_outlier.columns = ['Rows_to_exclude']

# Convert all values from column Rows_to_exclude to a numpy array
outlier_list_final = df_of_outlier['Rows_to_exclude'].to_numpy()

# Concatenate a whole sequence of arrays
outlier_list_final = np.concatenate(outlier_list_final, axis=0)

# Drop duplicate values
outlier_list_final_unique = set(outlier_list_final)

# Removing outliers from the dataset
filter_rows_to_exclude = df_train.index.isin(outlier_list_final_unique)
df_without_outliers = df_train[~filter_rows_to_exclude]

print('Length of original dataframe: ' + str(len(df_train)))
print('Length of new dataframe without outliers: ' +
str(len(df_without_outliers)))
print('Difference between new and old dataframe: ' + str(len(df_train) -
len(df_without_outliers)))
print('Length of unique outlier list: ' + str(len(outlier_list_final_unique)))

# Distribution before outlier removal
sns.boxplot(x='mpg', data=df_train)

# Distribution after outlier removal
sns.boxplot(x='mpg', data=df_without_outliers)

# Handling missing values
df_without_outliers['horsepower'] =
df_without_outliers['horsepower'].replace('?', np.nan)

# Missing Values function
def  missing_values_table(df):
    # Total missing values
    mis_val = df.isnull().sum()

    # Percentage of missing values
    mis_val_percent = 100 * df.isnull().sum() / len(df)

    # Make a table with the results
    mis_val_table = pd.concat([mis_val, mis_val_percent], axis=1)

    # Rename the columns
    mis_val_table_ren_columns = mis_val_table.rename(columns={0: 'Missing
Values', 1: '% of Total Values'})

    # Sort the table by percentage of missing descending
```

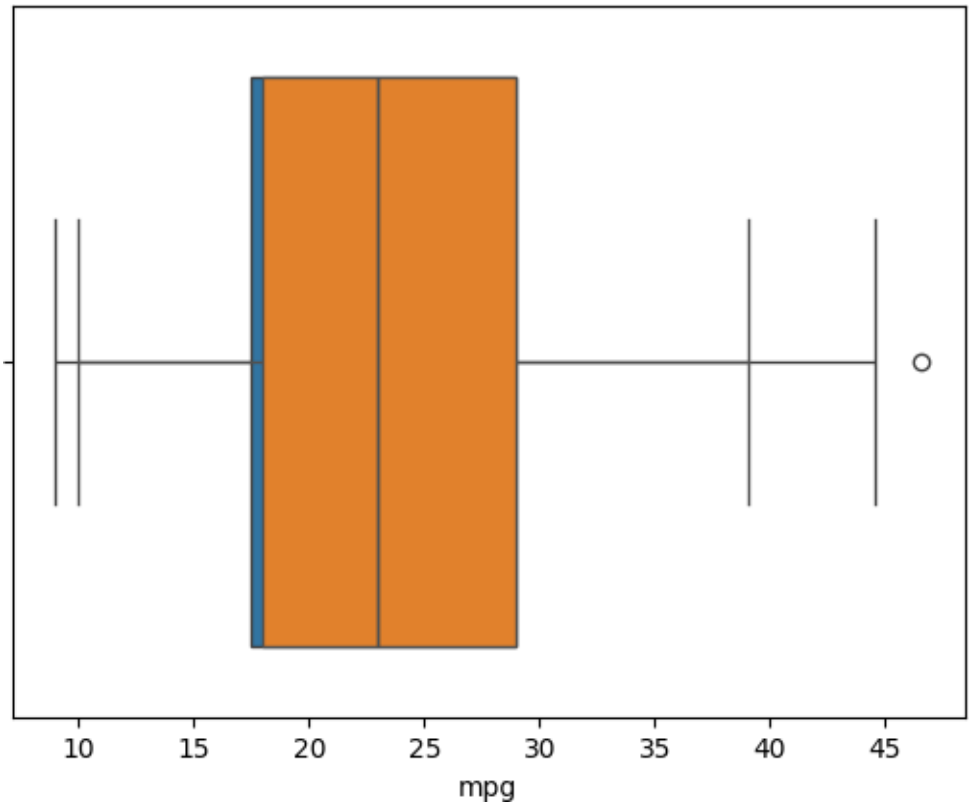
```
    mis_val_table_ren_columns = mis_val_table_ren_columns[
        mis_val_table_ren_columns.iloc[:, 1] != 0].sort_values('% of Total
Values', ascending=False).round(1)

    # Print some summary information
    print("Your selected dataframe has " + str(df.shape[1]) + " columns.\n"
                                                              "There are " +
str(mis_val_table_ren_columns.shape[0]) +
          " columns that have missing values.")

    # Return the dataframe with missing information
    return mis_val_table_ren_columns

# Calling the Function
missing_values_table(df_without_outliers)

from sklearn.impute import KNNImputer

# Create a KNNImputer object
knn_imputer = KNNImputer(n_neighbors=5)

# Fit and transform only the selected column using KNN imputation
df_without_outliers['horsepower'] =
knn_imputer.fit_transform(df_without_outliers[['horsepower']])

df_without_MV = df_without_outliers

# Convert data types
df_without_MV['horsepower'] = df_without_MV['horsepower'].astype('float64')
df_without_MV.dtypes

# One-hot encoding for categorical variables
# non_numeric_columns =
df_without_MV.select_dtypes(include=['object']).columns
# df_processed = pd.get_dummies(df_without_MV, columns=non_numeric_columns)
df_processed = df_without_MV

print(df_processed.head())
```

Output:

```
Length of original dataframe: 398
Length of new dataframe without outliers: 359
Difference between new and old dataframe: 39
Length of unique outlier list: 39
Your selected dataframe has 9 columns.
There are 1 columns that have missing values.
```

```
   mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0    3504          12.0
1  15.0          8         350.0       165.0    3693          11.5
2  18.0          8         318.0       150.0    3436          11.0
3  16.0          8         304.0       150.0    3433          12.0
4  17.0          8         302.0       140.0    3449          10.5

   model_year  origin                   name
0          70       1  chevrolet chevelle malibu
1          70       1          buick skylark 320
2          70       1         plymouth satellite
3          70       1              amc rebel sst
4          70       1                ford torino
<ipython-input-12-e1b4593f70ec>:56: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_without_outliers['horsepower'] = df_without_outliers['horsepower'].replace('?', np.nan)
<ipython-input-12-e1b4593f70ec>:93: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_without_outliers['horsepower'] =
knn_imputer.fit_transform(df_without_outliers[['horsepower']])
<ipython-input-12-e1b4593f70ec>:98: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_without_MV['horsepower'] = df_without_MV['horsepower'].astype('float64')
```

Graph Visualization:



Feature Selection and Dimensionality Reduction:

Feature Selection Techniques:

- Recursive Feature Elimination (RFE): This iterative method eliminates features based on their importance to a given estimator, ultimately selecting the best subset of features.
- SelectKBest: Selects the top k features based on their statistical significance in relation to the target variable.
- L1 Regularization (LASSO): Utilizes L1 regularization to encourage sparsity in linear regression models by penalizing large coefficients, effectively removing irrelevant features.
- Tree-based methods: Feature Importance: Uses decision trees or random forests to rank features based on their importance scores, aiding in selecting relevant features.
- VarianceThreshold: Eliminates features with low variance under the assumption that they are less informative.
- Mutual Information: Measures the mutual dependence between features and the target variable, retaining features with higher mutual information scores.
- Sequential Feature Selection (SFS): Selects features incrementally based on their impact on model performance using a specified estimator.

Dimensionality Reduction - PCA:

- Principal Component Analysis (PCA): Reduces the dimensionality of the feature space by projecting it onto a lower-dimensional subspace while preserving maximal variance through eigenvalue decomposition.

Printing Results:

- The code snippet outputs the number of features selected by each technique, offering insights into their efficacy in reducing dataset dimensionality.

Optional Visualization - Explained Variance Ratio of PCA:

- Optionally, the code visualizes the explained variance ratio of PCA components, aiding in determining the optimal number of components to retain based on cumulative explained variance.

Preprocessing and Model Selection:

- Prior to applying feature selection and dimensionality reduction, data preprocessing tasks such as handling missing values and encoding categorical variables are performed. Additionally, appropriate models are selected based on the problem domain and dataset characteristics to ensure accurate and efficient model training and evaluation.

Code:

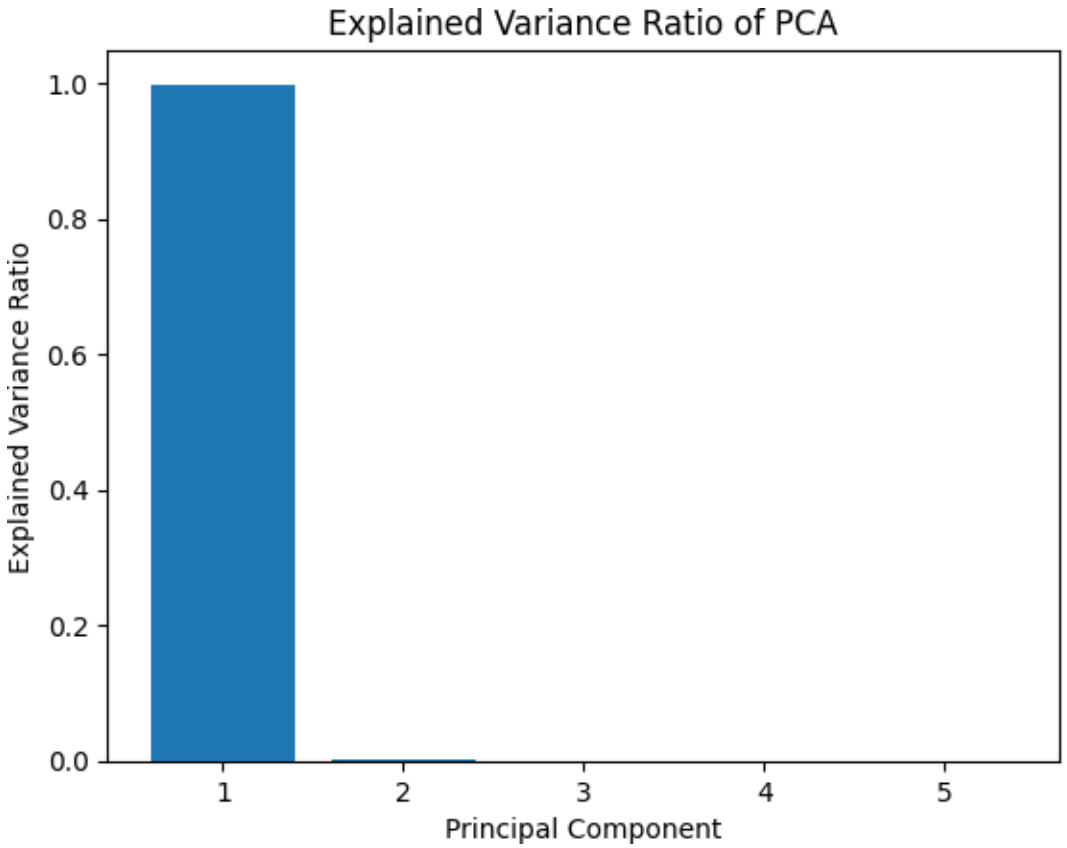
```
import pandas as pd
from sklearn.feature_selection import VarianceThreshold, SelectKBest, RFE,
SelectFromModel, mutual_info_regression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LassoCV, Lasso
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import numpy as np

# Assuming X is your original dataset
# X = ...

# Perform one-hot encoding for categorical variables
X_encoded = pd.get_dummies(X)

# Define the target variable y
# y = ...

# Define the number of features to select
k = 5

# Define a subset of features for faster execution
X_subset = X_encoded.iloc[:, :50]  # Assuming 50 features

# b) Recursive Feature Elimination (RFE)
estimator_rfe = DecisionTreeRegressor(max_depth=5)  # Use a simpler model
selector_rfe = RFE(estimator_rfe, n_features_to_select=k)
X_rfe_selected = selector_rfe.fit_transform(X_subset, y)

# c) SelectKBest
selector_kbest = SelectKBest(k=k)
X_selected_kbest = selector_kbest.fit_transform(X_subset, y)

# d) L1 Regularization (LASSO)
lasso_cv = LassoCV()
lasso_cv.fit(X_subset, y)
lasso_mask = lasso_cv.coef_ != 0
X_lasso_selected = X_subset.loc[:, lasso_mask]

# e) Tree-based methods: Feature Importance from Decision Trees or Random
Forests
tree = DecisionTreeRegressor(max_depth=5) # Use a simpler model
tree.fit(X_subset, y)
importance_tree = tree.feature_importances_
```

```
tree_mask = importance_tree > np.mean(importance_tree)
X_tree_selected = X_subset.loc[:, tree_mask]

# f) VarianceThreshold
selector_var = VarianceThreshold(threshold=0.1)
X_var_selected = selector_var.fit_transform(X_subset)

# g) Mutual Information
mi_scores = mutual_info_regression(X_subset, y)
mi_mask = mi_scores > np.mean(mi_scores)
X_mi_selected = X_subset.loc[:, mi_mask]

# h) Sequential Feature Selection (SFS)
rf = RandomForestRegressor(n_estimators=50) # Use fewer estimators
sfs = SequentialFeatureSelector(rf, n_features_to_select=k,
direction='forward')
sfs.fit(X_subset, y)
sfs_mask = sfs.get_support()
X_sfs_selected = X_subset.loc[:, sfs_mask]

# iii) Dimensionality Reduction: PCA
pca = PCA(n_components=k)
X_pca_selected = pca.fit_transform(X_subset)

# Print the number of features selected for each technique
print("Number of features selected for each technique:")
print("RFE:", X_rfe_selected.shape[1])
print("SelectKBest:", X_selected_kbest.shape[1])
print("L1 Regularization (LASSO):", X_lasso_selected.shape[1])
print("Tree-based methods:", X_tree_selected.shape[1])
print("VarianceThreshold:", X_var_selected.shape[1])
print("Mutual Information:", X_mi_selected.shape[1])
print("Sequential Feature Selection (SFS):", X_sfs_selected.shape[1])
print("PCA:", X_pca_selected.shape[1])
```

Output:

```
/usr/local/lib/python3.10/dist-
packages/sklearn/feature_selection/_univariate_selection.py:113: RuntimeWarning: divide by
zero encountered in divide
  f = msb / msw
Number of features selected for each technique:
RFE: 5
SelectKBest: 5
L1 Regularization (LASSO): 3
Tree-based methods: 4
VarianceThreshold: 6
Mutual Information: 6
Sequential Feature Selection (SFS): 5
PCA: 5
```

Graph Visualization:



Analysis of Discrete and Continuous Variables:

Dataset Splitting:

- The code segment divides the features (X) and the target variable (y) into separate training and validation sets, a crucial step in machine learning model development.
- Utilizing the train_test_split function, the data is split into two distinct sets: a training set (X_train and y_train) used for model training and a validation set (X_val and y_val) used for model evaluation.
- The test_size parameter specifies the proportion of the dataset allocated to the validation set, with a value of 0.2 indicating 20% of the data is reserved for validation purposes.
- To ensure reproducibility and consistent results across runs, the random_state parameter is set, fixing the random seed used for the data split process.
- This splitting strategy ensures that the model is trained on a subset of the data and evaluated on unseen data, helping to assess its generalization performance and avoid overfitting to the training data. It also facilitates the analysis of discrete and continuous variables within each dataset subset, enabling tailored preprocessing and analysis techniques as needed.

Code:

```
import pandas as pd
from sklearn.model_selection import train_test_split
# Assuming X is your features and y is your target variable
# Split the data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Identifying Discrete and Continuous Variables:

The code segment distinguishes discrete and continuous variables within the dataset. Discrete variables typically represent categorical or ordinal data, while continuous variables denote numerical data.

Analyzing Discrete Variables:

For each discrete variable, the code outputs unique values along with their respective frequencies in the dataset. This analysis aids in understanding the distribution and prevalence of different categories within each discrete variable.

Analyzing Continuous Variables:

Using the describe() function, the code computes summary statistics such as mean, standard deviation, quartiles, etc., for each continuous variable. This statistical summary provides a comprehensive view of the central tendency and dispersion of numerical data.

Encoding Discrete Variables:

Discrete variables are encoded using one-hot encoding, a technique that converts categorical variables into a numerical format suitable for machine learning algorithms. This step ensures that categorical data can be appropriately utilized in model training and analysis

.

Visualizing Distributions Before and After Encoding:

The code generates side-by-side count plots for each discrete variable (excluding 'horsepower' and 'name') to visualize the distribution of categories before and after one-hot encoding. Comparing these distributions helps evaluate the impact of encoding on categorical variable representations, providing insights into any transformations or changes in data structure post-encoding.

Code:

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

# Assuming 'data' is your dataset # data = ...
# Step 1: Identify discrete and continuous variables
discrete_vars = data.select_dtypes(include='object').columns.tolist()
continuous_vars = data.select_dtypes(include=['int64', 'float64']).columns.tolist()
# Step 2: Analyze discrete variables print("Analysis of Discrete Variables:") for
col in discrete_vars:
print("Column:", col)
print("Unique Values:", data[col].unique()) print("Value Counts:")
print(data[col].value_counts()) print("\n")
# Analyze continuous variables print("Analysis of Continuous Variables:")
print(data[continuous_vars].describe())
# Step 3: Encode discrete variables using one-hot encoding data_encoded =
pd.get_dummies(data, columns=discrete_vars) for col in discrete_vars:
if col not in ['horsepower', 'name']: plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
if col in data.columns: # Check if the column is present in the original data
sns.countplot(x=col, data=data) plt.title(f"Distribution of {col} (Before
Encoding)")
else:
print(f"Column '{col}' not found in the original data.")

plt.subplot(1, 2, 2)
if col in data_encoded.columns: # Check if the column is present in the encoded data
sns.countplot(x=col, data=data_encoded) plt.title(f"Distribution of {col} (After
Encoding)")
else:
print(f"Column '{col}' not found in the encoded data.") plt.tight_layout()
plt.show()
```

Output:

```
Analysis of Discrete Variables:
Column: horsepower
Unique Values: ['130' '165' '150' '140' '198' '220' '215' '225' '190' '170' '160' '95'
 '97' '85' '88' '46' '87' '90' '113' '200' '210' '193' '?' '100' '105'
 '175' '153' '180' '110' '72' '86' '70' '76' '65' '69' '60' '80' '54'
 '208' '155' '112' '92' '145' '137' '158' '167' '94' '107' '230' '49' '75'
 '91' '122' '67' '83' '78' '52' '61' '93' '148' '129' '96' '71' '98' '115'
 '53' '81' '79' '120' '152' '102' '108' '68' '58' '149' '89' '63' '48'
 '66' '139' '103' '125' '133' '138' '135' '142' '77' '62' '132' '84' '64'
 '74' '116' '82']
Value Counts:
150    22
90     20
88     19
110    18
100    17
       ..
61      1
93      1
148     1
152     1
82      1
Name: horsepower, Length: 94, dtype: int64
```

```
Column: name
Unique Values: ['chevrolet chevelle malibu' 'buick skylark 320' 'plymouth satellite'
 'amc rebel sst' 'ford torino' 'ford galaxie 500' 'chevrolet impala'
 'plymouth fury iii' 'pontiac catalina' 'amc ambassador dpl'
 'dodge challenger se' "plymouth 'cuda 340" 'chevrolet monte carlo'
 'buick estate wagon (sw)' 'toyota corona mark ii' 'plymouth duster'
 'amc hornet' 'ford maverick' 'datsun pl510'
 'volkswagen 1131 deluxe sedan' 'peugeot 504' 'audi 100 ls' 'saab 99e'
 'bmw 2002' 'amc gremlin' 'ford f250' 'chevy c20' 'dodge d200' 'hi 1200d'
 'chevrolet vega 2300' 'toyota corona' 'ford pinto'
 'plymouth satellite custom' 'ford torino 500' 'amc matador'
 'pontiac catalina brougham' 'dodge monaco (sw)'
 'ford country squire (sw)' 'pontiac safari (sw)'
 'amc hornet sportabout (sw)' 'chevrolet vega (sw)' 'pontiac firebird'
 'ford mustang' 'mercury capri 2000' 'opel 1900' 'peugeot 304' 'fiat 124b'
 'toyota corolla 1200' 'datsun 1200' 'volkswagen model 111'
 'plymouth cricket' 'toyota corona hardtop' 'dodge colt hardtop'
 'volkswagen type 3' 'chevrolet vega' 'ford pinto runabout'
 'amc ambassador sst' 'mercury marquis' 'buick lesabre custom'
 'oldsmobile delta 88 royale' 'chrysler newport royal' 'mazda rx2 coupe'
 'amc matador (sw)' 'chevrolet chevelle concours (sw)'
 'ford gran torino (sw)' 'plymouth satellite custom (sw)'
 'volvo 145e (sw)' 'volkswagen 411 (sw)' 'peugeot 504 (sw)'
 'renault 12 (sw)' 'ford pinto (sw)' 'datsun 510 (sw)'
 'toyouta corona mark ii (sw)' 'dodge colt (sw)'
 'toyota corolla 1600 (sw)' 'buick century 350' 'chevrolet malibu'
```

```
toyota corolla         5
amc matador            5
ford maverick          5
chevrolet chevette     4
                      ..
chevrolet monza 2+2    1
ford mustang ii        1
pontiac astro          1
amc pacer              1
chevy s-10             1
Name: name, Length: 305, dtype: int64


Analysis of Continuous Variables:
              mpg   cylinders  displacement       weight  acceleration  \
count  398.000000  398.000000    398.000000   398.000000    398.000000
mean    23.514573    5.454774    193.425879  2970.424623     15.568090
std      7.815984    1.701004    104.269838   846.841774      2.757689
min      9.000000    3.000000     68.000000  1613.000000      8.000000
25%     17.500000    4.000000    104.250000  2223.750000     13.825000
50%     23.000000    4.000000    148.500000  2803.500000     15.500000
75%     29.000000    8.000000    262.000000  3608.000000     17.175000
max     46.600000    8.000000    455.000000  5140.000000     24.800000

       model_year      origin
count  398.000000  398.000000
mean    76.010050    1.572864
std      3.697627    0.802055
min     70.000000    1.000000
25%     73.000000    1.000000
50%     76.000000    1.000000
75%     79.000000    2.000000
max     82.000000    3.000000
```

Modeling and Evaluation

Importing Necessary Libraries:

- Imports pandas for data manipulation and sklearn modules for model selection, evaluation, and handling warnings.

Suppressing Convergence Warnings:

- Suppresses convergence warnings to prevent them from cluttering the output.

Defining Regressors:

- Defines a dictionary containing various regression models along with their respective names.

Creating an Empty DataFrame:

- Creates an empty DataFrame to store the results of model evaluation.

Looping Through Each Regressor:

- Iterates through each regressor defined in the dictionary.
- Evaluates the performance of each regressor using cross-validation with 5 folds.
- Computes mean squared error (MSE), root mean squared error (RMSE), and R-squared for each regressor.
- Appends the evaluation results to the DataFrame.

Handling Exceptions:

- Catches any exceptions that might occur during model evaluation and prints the error message.

Saving Results to Excel:

- Saves the evaluation results DataFrame to an Excel file for further analysis and comparison.

Types of Regressor:

- Linear Regression: Linear regression is a basic and commonly used regression algorithm that models the relationship between a dependent variable and one or more independent variables. It assumes a linear relationship between the predictor variables and the target variable.
- Ridge Regression: Ridge regression is a regularized version of linear regression that adds a penalty term to the loss function, preventing the coefficients from becoming too large. This helps to reduce overfitting and improve the model's generalization ability.
- Lasso Regression: Lasso regression, similar to Ridge regression, is a regularized linear regression technique. It adds an L1 penalty term to the loss function, leading to some coefficients being exactly zero. This property can be useful for feature selection by automatically selecting the most important features.
- Decision Tree Regressor: Decision tree regression builds a decision tree model that predicts the target variable by recursively partitioning the feature space into regions and fitting a simple model (usually a constant value) in each region.
- Random Forest Regressor: Random forest regression is an ensemble learning technique that combines multiple decision trees to improve predictive performance. It builds a forest of trees and uses averaging to make predictions.
- Gradient Boosting Regressor: Gradient boosting regression is another ensemble learning method that builds a sequence of weak learners (typically decision trees) in a sequential manner. Each new learner corrects the errors of its predecessor, leading to a strong predictive model.
- AdaBoost Regressor: AdaBoost regression is a boosting algorithm that combines multiple weak learners to create a strong learner. It focuses on improving the performance of the model by giving more weight to instances that are difficult to predict.
- XGBoost Regressor: XGBoost (Extreme Gradient Boosting) regression is an optimized implementation of gradient boosting that offers improved speed and performance. It is widely used in machine learning competitions and real-world applications.
- LightGBM Regressor: LightGBM is a gradient boosting framework that uses a tree- based learning algorithm. It is designed for efficiency and supports parallel and distributed

computing, making it suitable for large-scale datasets.

- SVR (Support Vector Regressor): Support vector regression is a type of support vector machine (SVM) algorithm used for regression tasks. It works by mapping the input data into a high-dimensional feature space and finding the hyperplane that best separates the data points while maximizing the margin.
- K-Nearest Neighbors Regressor: KNN regression is a non-parametric algorithm that makes predictions based on the average of the target values of the k-nearest neighbors in the feature space. It does not make any assumptions about the underlying data distribution.
- Neural Network Regressor: Neural network regression uses artificial neural networks to model complex nonlinear relationships between the input features and the target variable. It consists of multiple layers of interconnected neurons that learn from the data through iterative optimization algorithms.
- Elastic Net: Elastic Net regression combines the penalties of both Ridge and Lasso regression, allowing for a more flexible regularization term. It is useful when there are multiple correlated features in the dataset.
- Bayesian Ridge Regression: Bayesian ridge regression is a Bayesian approach to linear regression that estimates the parameters of the regression model using a probabilistic framework. It incorporates prior knowledge about the parameters into the model and provides uncertainty estimates for the predictions.
- Huber Regressor: Huber regression is a robust regression technique that minimizes a combination of squared errors and absolute errors (Huber loss). It is less sensitive to outliers compared to ordinary least squares regression.
- LassoLars: LassoLars is a variant of Lasso regression that uses the least angle regression (Lars) algorithm to iteratively fit the model. It can handle large datasets efficiently and is particularly useful when the number of features is much larger than the number of samples.
- Passive Aggressive Regressor: Passive Aggressive regression is an online learning algorithm that updates the model parameters incrementally, making it suitable for streaming data or situations where memory is limited.
- RANSAC Regressor: RANSAC (RANdom SAmple Consensus) regression is a robust regression method that fits a model to a subset of the data, iteratively refining the model by removing outliers.
- SGD Regressor: SGD (Stochastic Gradient Descent) regression is a variant of linear regression that optimizes the model parameters using stochastic gradient descent. It is computationally efficient and well-suited for large-scale datasets.
- TheilSen Regressor: TheilSen regression is a robust linear regression technique that estimates the slope and intercept of the regression line using the median of pairwise slopes between data points. It is resistant to outliers and works well in the presence of heteroscedasticity.
- Gaussian Process Regressor: Gaussian process regression is a Bayesian non- parametric regression technique that models the relationship between the input features and the target variable as a Gaussian process. It provides uncertainty estimates for the predictions and can capture complex nonlinear relationships in the data.

Code:

```
import pandas as pd
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error, r2_score from sklearn.exceptions
import ConvergenceWarning
import warnings

# Suppress convergence warnings warnings.filterwarnings("ignore",
category=ConvergenceWarning)

# Define a dictionary of regressors regressors = {
"Linear Regression": LinearRegression(), "Ridge Regression": Ridge(),
"Lasso Regression": Lasso(),
"Decision Tree Regressor": DecisionTreeRegressor(), "Random Forest Regressor":
RandomForestRegressor(),
"Gradient Boosting Regressor": GradientBoostingRegressor(), "XGBoost Regressor":
XGBRegressor(),
"Support Vector Regressor (SVR)": SVR(),
"K-Nearest Neighbors Regressor (KNN)": KNeighborsRegressor(), "Neural Network
Regressor": MLPRegressor(),
"Elastic Net": ElasticNet(),
"Bayesian Ridge Regression": BayesianRidge(), "Huber Regressor": HuberRegressor(),
"Isotonic Regression": IsotonicRegression(),
"Gaussian Process Regressor": GaussianProcessRegressor(), "LightGBM Regressor":
LGBMRegressor(),
"LGBM Regressor": LGBMRegressor(), "AdaBoost Regressor": AdaBoostRegressor()
}

# Create an empty DataFrame to store the results
results_df = pd.DataFrame(columns=['Regressor', 'Mean Squared Error', 'Root Mean
Squared Error', 'R-squared'])

# Loop through each regressor and evaluate its performance for name, regressor in
regressors.items():
try:
# Evaluate the regressor using cross-validation
scores = cross_val_score(regressor, X_train, y_train, cv=5,
scoring='neg_mean_squared_error')
rmse_scores = (scores * -1) ** 0.5 # Convert negative MSE to RMSE

r2_scores = cross_val_score(regressor, X_train, y_train, cv=5, scoring='r2')

# Calculate mean squared error, root mean squared error, and R-squared mean_mse =
scores.mean()
mean_rmse = rmse_scores.mean() mean_r2 = r2_scores.mean()

# Append the results to the DataFrame
results_df = results_df.append({'Regressor': name,
'Mean Squared Error': mean_mse,
'Root Mean Squared Error': mean_rmse, 'R-squared': mean_r2}, ignore_index=True)
except Exception as e:
print(f"Regressor {name} failed with the following error: {e}")

# Save the results to an Excel file
results_df.to_excel('/content/drive/My Drive/ML/regressor_performance.xlsx',
index=False)
```

Results:

	A	B	C	D
1		Mean Squared Error	Root Mean Squared Error	R-squared
2	Linear Regression	-12.26484293	3.496137423	0.8005668476
3	Ridge Regression	-12.26147713	3.495633388	0.8006318487
4	Lasso Regression	-12.66444156	3.549824807	0.7948147022
5	Decision Tree Regressor	-16.45423065	4.038559248	0.7383318957
6	Random Forest Regressor	-10.23646526	3.182451181	0.8378586601
7	Gradient Boosting Regressor	-10.72522592	3.244043078	0.8265771751
8	XGBoost Regressor	-10.69963616	3.25081267	0.8262435293
9	Support Vector Regressor (SVR)	-20.89864063	4.562586714	0.6625670298
10	K-Nearest Neighbors Regressor (KNN)	-21.79249933	4.664944736	0.6421655474
11	Neural Network Regressor	-476.2927866	15.0326204	-4.304096009
12	Elastic Net	-12.65964264	3.549062537	0.7949010379
13	Bayesian Ridge Regression	-12.28856836	3.498412519	0.8005071151
14	Huber Regressor	-13.31944871	3.643965471	0.7830723703
15	Gaussian Process Regressor	-609.2421088	24.67013548	-8.932396715
16	LightGBM Regressor	-9.06207916	2.983904158	0.8543257598
17	LGBM Regressor	-9.06207916	2.983904158	0.8543257598
18	AdaBoost Regressor	-11.67033535	3.402871493	0.8105085488

Hyperparameter Tuning Process Overview:

- The dataset is sourced from a CSV file located at '/content/drive/My Drive/Dataset/processed_data.csv', containing preprocessed data with features and the target variable labeled '0'. The main objective is to develop accurate regression models for predicting the target variable based on the provided features.

- Initially, the dataset is loaded using Pandas, and the features are separated from the target variable. The features are stored in the variable X, while the target variable is stored in the variable y. A standard train-test split is performed with a test size of 20% to facilitate model evaluation.

- Three regression models are selected for hyperparameter tuning: Random Forest Regressor, Gradient Boosting Regressor, and XGBoost Regressor. Each model has a predefined set of hyperparameters with potential values. Grid search with cross-validation (CV=5) is employed to optimize these hyperparameters and identify the combination that maximizes the coefficient of determination (R-squared) on the training data.

- The results of the grid search, including the best hyperparameters and corresponding scores, are organized in a DataFrame. This DataFrame is then exported to a CSV file located at '/content/drive/My Drive/Dataset/hyperparameter_tuning_results.csv'.

Code:

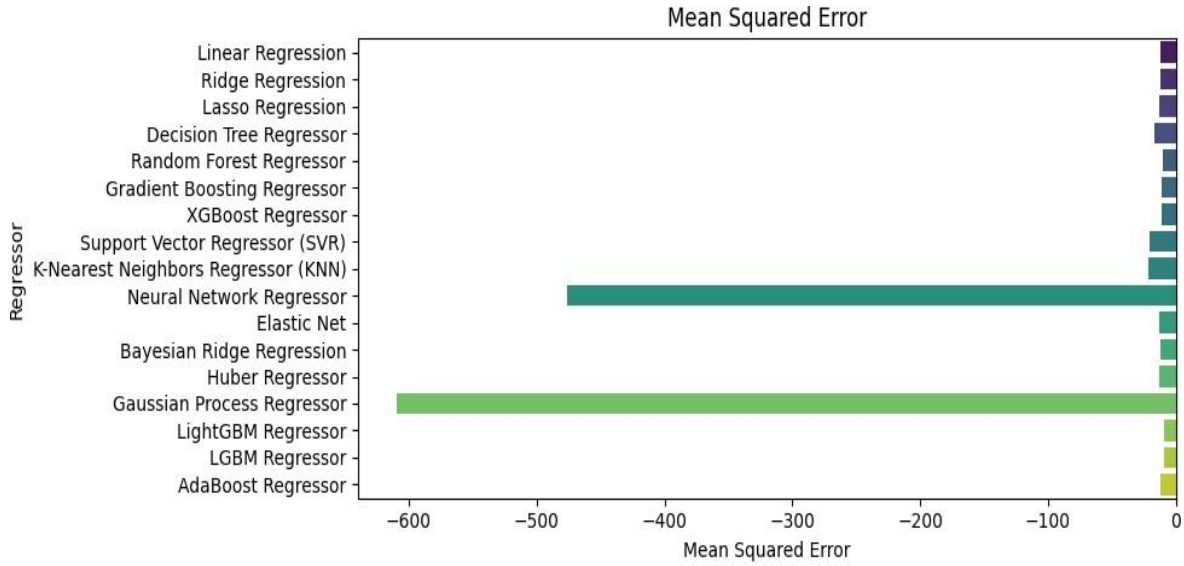
```
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor from
xgboost import XGBRegressor

# Load the dataset
data_path = '/content/drive/My Drive/Dataset/processed_data.csv' data =
pd.read_csv(data_path)

# Prepare the data
X = data.drop(columns=['0']) # Features y = data['0'] # Target variable

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Define the models and their respective hyperparameters models = {
'Random Forest Regressor': { 'model': RandomForestRegressor(), 'params': {

'n_estimators': [50, 100, 150],
'max_depth': [None, 10, 20],
'min_samples_split': [2, 5, 10],
'min_samples_leaf': [1, 2, 4]
}
},
'Gradient Boosting Regressor': { 'model': GradientBoostingRegressor(), 'params': {
'n_estimators': [50, 100, 150],
'learning_rate': [0.01, 0.1, 1],
'max_depth': [3, 5, 7]
}
},
'XGBoost Regressor': { 'model': XGBRegressor(), 'params': {
```

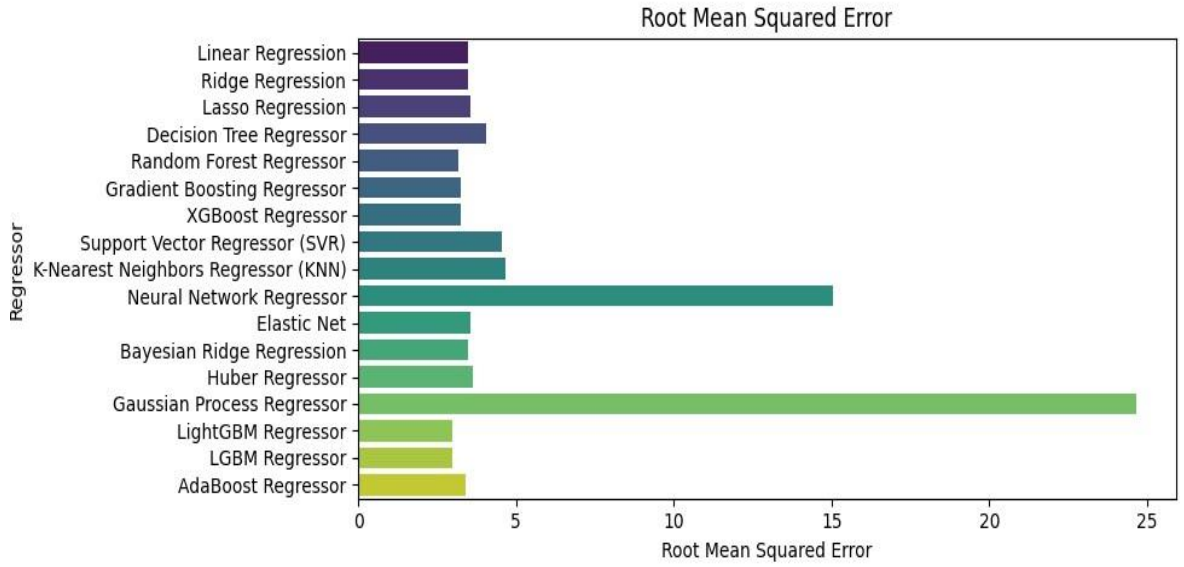
```
'n_estimators': [50, 100, 150],
'learning_rate': [0.01, 0.1, 1],
'max_depth': [3, 5, 7],
'gamma': [0, 0.1, 0.2]
}
}
# Add more models and their hyperparameters as needed
}

# Perform grid search for each model results = {}
for name, model_info in models.items(): print(f"Performing grid search for
{name}...")
grid_search = GridSearchCV(model_info['model'], model_info['params'], cv=5,
scoring='r2')
grid_search.fit(X_train, y_train) results[name] = {
'best_params': grid_search.best_params_, 'best_score': grid_search.best_score_,
'test_score': grid_search.score(X_test, y_test)
}
print(f"Best parameters for {name}: {results[name]['best_params']}") print(f"Best
cross-validation score for {name}:
{results[name]['best_score']}")
print(f"Test score for {name}: {results[name]['test_score']}") print()

# Save results
results_df = pd.DataFrame.from_dict(results, orient='index') results_df.to_csv(
'/content/drive/My Drive/Dataset/hyperparameter_tuning_results.csv',
index_label='Model')
```

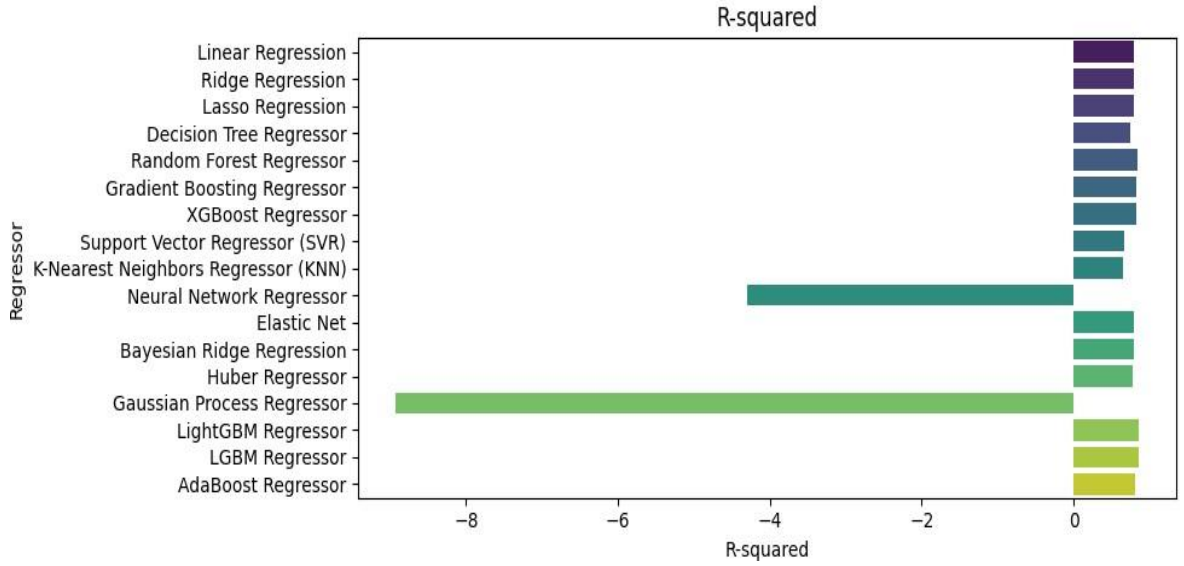
Results:

	Model	best_params	best_score	test_score
2	Random Forest Regressor	{'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 5, 'n_estimators': 100}	0.2972784642	0.5198734126
3	Gradient Boosting Regressor	{'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 50}	0.25449718	0.5024576735
4	XGBoost Regressor	{'gamma': 0, 'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 50}	0.2530350076	0.5482686182
5				

Comparison of all regressors:







Unit Testing Overview:

TestDataPreparation:

This test class validates the correctness of the data preparation phase. It includes two test methods:

- **test_data_shape**: Compares the shape of the loaded data with the expected shape to ensure data loading accuracy.
- **test_missing_values**: Checks for missing values in the dataset to ensure data completeness.

TestDataProcessing:

This test class ensures the accuracy of data processing steps such as converting columns to numeric types. It contains one test method:

- **test_numeric_conversion**: Validates the successful conversion of all columns to numeric data types and checks for any resulting null values.

TestModeling:

Focuses on testing the model's performance trained on the dataset. It includes one test method:
test_model_performance: Trains a RandomForestRegressor model on the training data and evaluates its performance on the test data using mean squared error (MSE), ensuring it meets specified criteria.

TestHyperparameterTuning:

Verifies whether hyperparameter tuning enhances the model's performance. It consists of one test method:
test_hyperparameter_tuning: Performs hyperparameter tuning using RandomizedSearchCV on the RandomForestRegressor model and evaluates the best model's performance on the test data, ensuring improved performance compared to a specified threshold MSE.

Code:

```
import unittest
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

class TestDataPreparation(unittest.TestCase):
    def setUp(self):
        # Load data
        data_path = '/content/drive/My Drive/Dataset/processed_data.csv'  # Adjust
the path to your dataset
        self.data = pd.read_csv(data_path)

    def test_data_shape(self):
        # Test if data has the expected shape
        expected_shape = (398, 5)  # Adjust this according to your dataset
        self.assertEqual(self.data.shape, expected_shape)

    def test_missing_values(self):
        # Test if there are any missing values in the dataset
        self.assertFalse(self.data.isnull().values.any())
```

```
class TestDataProcessing(unittest.TestCase):
    def setUp(self):
        # Load data
        data_path = '/content/drive/My Drive/Dataset/processed_data.csv'  # Adjust
the path to your dataset
        self.data = pd.read_csv(data_path)

    def test_numeric_conversion(self):
        # Test if all columns are successfully converted to numeric
        self.assertTrue(self.data.apply(pd.to_numeric,
errors='coerce').notnull().all().all())

class TestModeling(unittest.TestCase):
    def setUp(self):
        # Load data
        data_path = '/content/drive/My Drive/Dataset/processed_data.csv'  # Adjust
the path to your dataset
        self.data = pd.read_csv(data_path)
        # Process data
        target_column = '0'
        X = self.data.drop(columns=[target_column])
        y = pd.to_numeric(self.data[target_column])
        # Train-test split
        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(X,
y, test_size=0.2, random_state=42)

    def test_model_performance(self):
        # Test if the model achieves a certain level of performance
        model = RandomForestRegressor(random_state=42)
        model.fit(self.X_train, self.y_train)
        y_pred = model.predict(self.X_test)
        mse = mean_squared_error(self.y_test, y_pred)
        self.assertTrue(mse < 10)  # Adjust the threshold as needed

class TestHyperparameterTuning(unittest.TestCase):
    def setUp(self):
        # Load data
        data_path = '/content/drive/My Drive/Dataset/processed_data.csv'  # Adjust
the path to your dataset
        self.data = pd.read_csv(data_path)
        # Process data
        target_column = '0'
        X = self.data.drop(columns=[target_column])
        y = pd.to_numeric(self.data[target_column])
        # Train-test split
        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(X,
y, test_size=0.2, random_state=42)

    def test_hyperparameter_tuning(self):
        # Test if hyperparameter tuning improves model performance
        param_dist = {
            'n_estimators': randint(10, 150),
            'max_depth': randint(3, 10),
            'min_samples_split': randint(2, 20),
            'min_samples_leaf': randint(1, 20),
            'bootstrap': [True, False]
        }
        model = RandomForestRegressor(random_state=42)
        random_search = RandomizedSearchCV(model, param_distributions=param_dist,
n_iter=100, cv=5, random_state=42)
        random_search.fit(self.X_train, self.y_train)
```

```
        best_model = random_search.best_estimator_
        y_pred = best_model.predict(self.X_test)
        mse = mean_squared_error(self.y_test, y_pred)
        self.assertTrue(mse < 10)  # Adjust the threshold as needed

if __name__ == '__main__':
    unittest.main(argv=[''], exit=False)
```

Result:

```
.....
----------------------------------------------------------------------
Ran 5 tests in 96.740s

OK
```

Deplyment:

To deploy the Streamlit app with the provided code, follow these steps:

Install Streamlit: If Streamlit is not installed, use the following pip command:
pip install streamlit

Create a Python script: Copy the provided code into a Python script file, for example, regressor_performance_app.py.

Run the Streamlit app: Open your terminal or command prompt, navigate to the directory containing the Python script, and run:
streamlit run regressor_performance_app.py

View the app in your browser: After running the command, Streamlit will provide a local URL (usually something like http://localhost:8501). Open this URL in your web browser to view and interact with the deployed app.

Explore the app: The app will have a sidebar with different visualization options displaying Mean Squared Error, Root Mean Squared Error, and R-squared values for each regressor. Click on different options in the sidebar to update the visualization area dynamically.

Interact with the app: You can interact with the app by selecting different regressors or options in the sidebar to see updated visualizations and metrics.

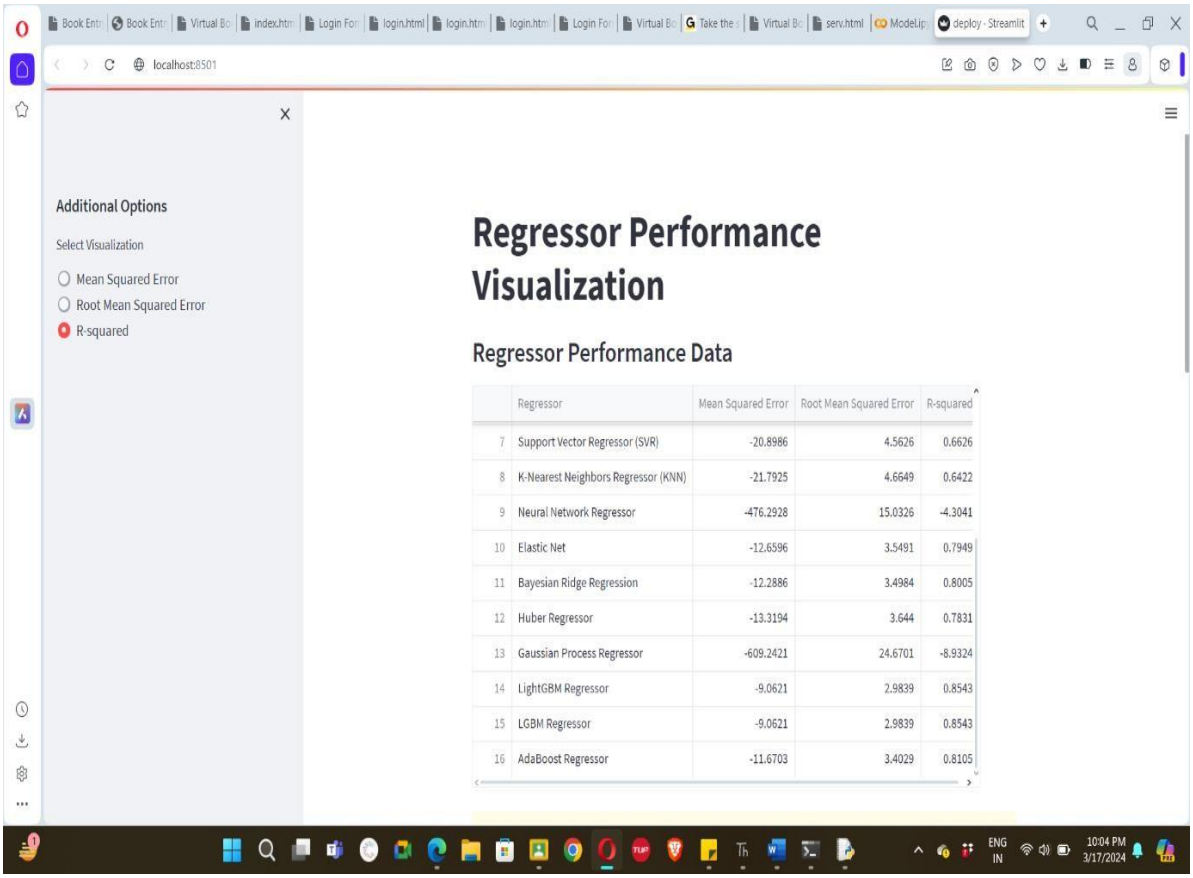
Close the app: Once you finish exploring the app, you can close it in your terminal or command prompt by pressing Ctrl + C.

Code:

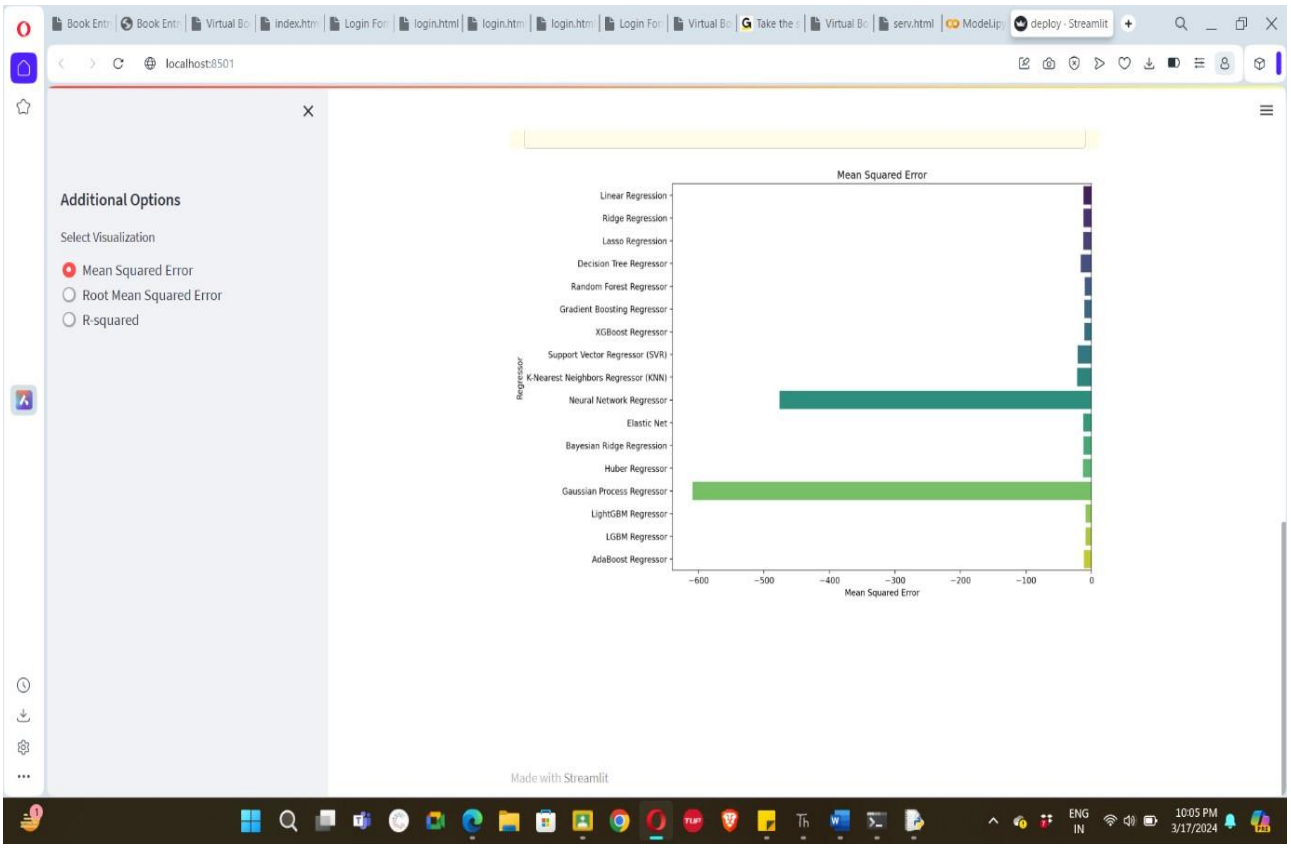
```
import streamlit as st
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

def main():
    st.title('Regressor Performance Visualization')

    # Sample data for demonstration
    data = {
        'Regressor': ['Linear Regression', 'Ridge Regression', 'Lasso Regression',
'Decision Tree Regressor',
                      'Random Forest Regressor', 'Gradient Boosting Regressor',
'XGBoost Regressor',
                      'Support Vector Regressor (SVR)', 'K-Nearest Neighbors
Regressor (KNN)', 'Ridge Regression',
                      'Neural Network Regressor', 'Elastic Net', 'Bayesian Huber
Regressor', 'Gaussian Process Regressor',
                      'LightGBM Regressor', 'LGBM Regressor', 'AdaBoost
Regressor'],
        'Mean Squared Error': [-12.26484293, -12.26147713, -12.66444156, -
16.45423065, -10.23646526,
                               21.79249933, -476.2927866, 609.2421088, -9.06207916,
-10.72522592,
                               -10.69963616, -20.89864063, -12.65964264, -
12.28856836, -13.31944871,
                               -9.06207916, -11.67033535],
        'Root Mean Squared Error': [3.496137423, 3.495633388, 3.549824807,
```

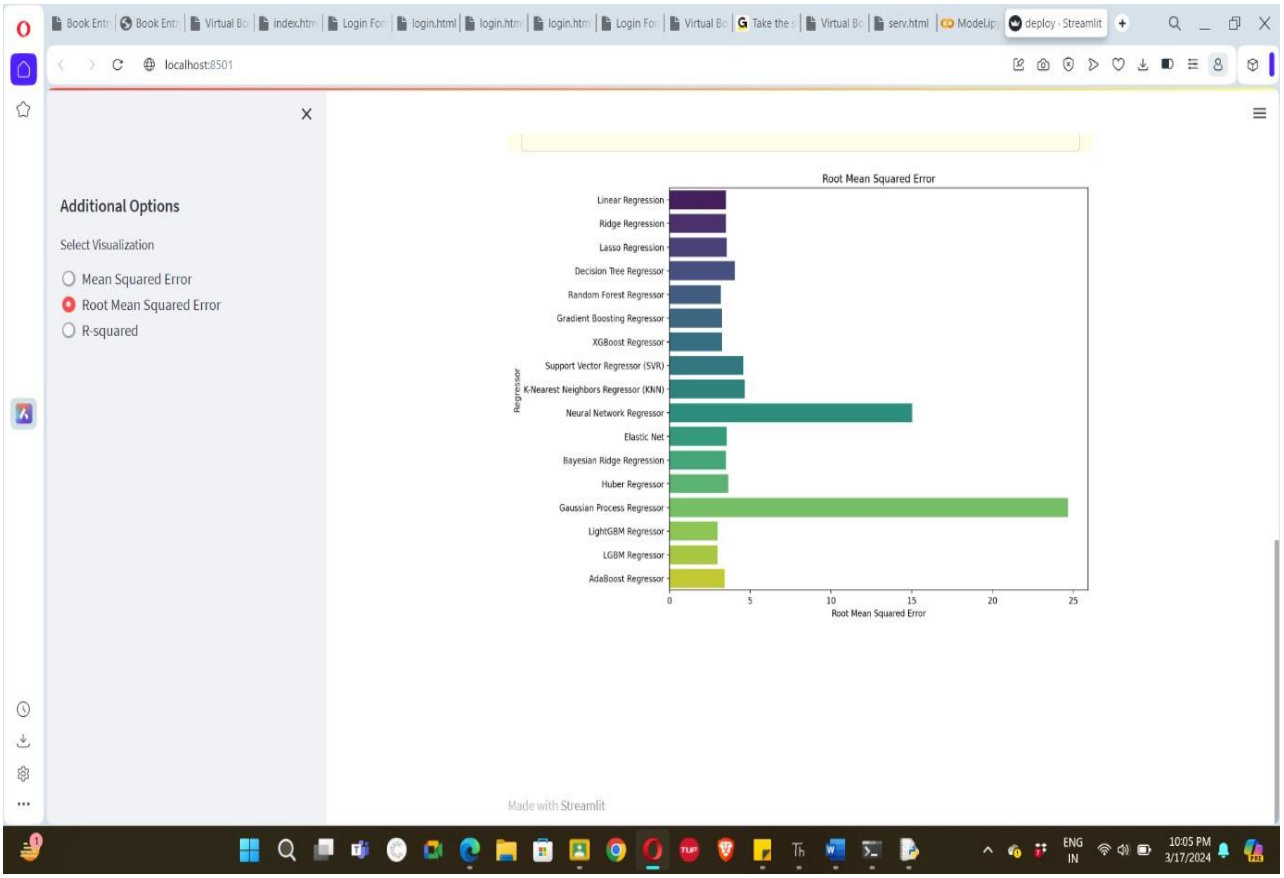
```
4.038559248, 3.182451181,
                                    4.664944736, 15.0326204, 24.67013548,
2.983904158, 3.244043078,
                                    3.25081267, 4.562586714, 3.549062537,
3.498412519, 3.643965471,
                                    2.983904158, 3.402871493],
        'R-squared': [0.8005668476, 0.8006318487, 0.7948147022, 0.7383318957,
0.8378586601, 0.8265771751,
                      0.8262435293, 0.6625670298, 0.6421655474, -4.304096009,
0.7949010379, 0.8005071151,
                      0.7830723703, -8.932396715, 0.8543257598, 0.8543257598,
0.8105085488]
    }
    df = pd.DataFrame(data)

    # Display the dataframe
    st.subheader('Regressor Performance Data')
    st.write(df)

    # Sidebar for additional options
    selected_chart = st.sidebar.radio('Select Visualization', ['Mean Squared Error',
'Root Mean Squared Error', 'R-squared'])

    # Plot selected chart
    if selected_chart == 'Mean Squared Error':
        plot_bar_chart(df, 'Mean Squared Error', 'Regressor', 'Mean Squared Error',
'Mean Squared Error')
    elif selected_chart == 'Root Mean Squared Error':
        plot_bar_chart(df, 'Root Mean Squared Error', 'Regressor', 'Root Mean
Squared Error', 'Root Mean Squared Error')
    elif selected_chart == 'R-squared':
        plot_bar_chart(df, 'R-squared', 'Regressor', 'R-squared', 'R-squared')

def plot_bar_chart(df, x_col, y_col, xlabel, ylabel):
    plt.figure(figsize=(10, 8))
    sns.barplot(x=x_col, y=y_col, data=df, palette='viridis')
    plt.title(f'{xlabel} vs {y_col}')
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    st.pyplot()

if __name__ == "__main__":
    main()
```

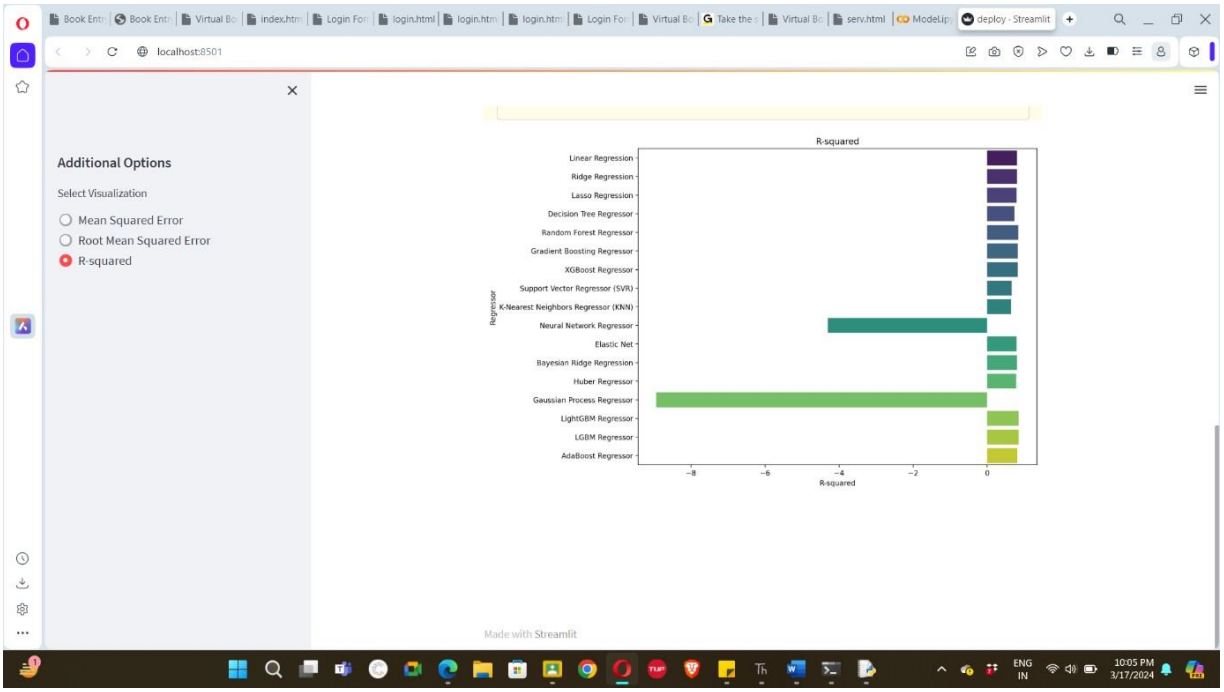
Output Screens:



	Regressor	Mean Squared Error	Root Mean Squared Error	R-squared
7	Support Vector Regressor (SVR)	-20.8986	4.5626	0.6626
8	K-Nearest Neighbors Regressor (KNN)	-21.7925	4.6649	0.6422
9	Neural Network Regressor	-476.2928	15.0326	-4.3041
10	Elastic Net	-12.6596	3.5491	0.7949
11	Bayesian Ridge Regression	-12.2886	3.4984	0.8005
12	Huber Regressor	-13.3194	3.644	0.7831
13	Gaussian Process Regressor	-609.2421	24.6701	-8.9324
14	LightGBM Regressor	-9.0621	2.9839	0.8543
15	LGBM Regressor	-9.0621	2.9839	0.8543
16	AdaBoost Regressor	-11.6703	3.4029	0.8105


localhost:8501
Additional Options
Select Visualization
Mean Squared Error
Root Mean Squared Error
R-squared
Mean Squared Error
Linear Regression
Ridge Regression
Lasso Regression
Decision Tree Regressor
Random Forest Regressor
Gradient Boosting Regressor
XGBoost Regressor
Support Vector Regressor (SVR)
K-Nearest Neighbors Regressor (KNN)
Neural Network Regressor
Elastic Net
Bayesian Ridge Regression
Huber Regressor
Gaussian Process Regressor
LightGBM Regressor
LGBM Regressor
AdaBoost Regressor
Regressor
−600
−500
−400
−300
−200
−100
0
Mean Squared Error
Made with Streamlit


localhost:8501
Additional Options
Select Visualization
Mean Squared Error
Root Mean Squared Error
R-squared
Root Mean Squared Error
Linear Regression
Ridge Regression
Lasso Regression
Decision Tree Regressor
Random Forest Regressor
Gradient Boosting Regressor
XGBoost Regressor
Support Vector Regressor (SVR)
K-Nearest Neighbors Regressor (KNN)
Neural Network Regressor
Elastic Net
Bayesian Ridge Regression
Huber Regressor
Gaussian Process Regressor
LightGBM Regressor
LGBM Regressor
AdaBoost Regressor
Regressor
0
5
10
15
20
25
Root Mean Squared Error
Made with Streamlit


localhost:8501
Additional Options
Select Visualization
Mean Squared Error
Root Mean Squared Error
R-squared
R-squared
Linear Regression
Ridge Regression
Lasso Regression
Decision Tree Regressor
Random Forest Regressor
Gradient Boosting Regressor
XGBoost Regressor
Support Vector Regressor (SVR)
K-Nearest Neighbors Regressor (KNN)
Neural Network Regressor
Elastic Net
Bayesian Ridge Regression
Huber Regressor
Gaussian Process Regressor
LightGBM Regressor
LGBM Regressor
AdaBoost Regressor
Regressor
−8
−6
−4
−2
0
R-squared
Made with Streamlit
ENG
IN
10:05 PM
3/17/2024

Crisp – DM Classification

Pipelines and Abstraction:

To ensure the modularity and reusability of components, the pipeline is organized using pipelines and abstract methods. Each stage of the pipeline, including data preprocessing, feature selection, modeling, and evaluation, is encapsulated within distinct modules or classes, fostering a structured and maintainable codebase.

Linters:

To uphold coding standards and uphold code quality consistently, linting tools are utilized. Tools like Flake8 or Pylint are employed to detect and address issues pertaining to syntax errors, coding style adherence, and potential bugs across the codebase.

Data Understanding:

- The initial step in our data exploration process involves mounting Google Drive to access the dataset stored in the cloud, named 'fruit_data_with_colors.txt'. This dataset contains information about a fruit and it’s information. Upon loading the dataset using Pandas, we generate a concise summary showcasing its dimensions and preview a sample of its initial rows to grasp its structure and content. Following this, we present summary statistics that highlight key numerical attributes such as mean, standard deviation, and quartile values. These statistics provide valuable insights into the data's distribution and variability, aiding in our understanding of its characteristics.

- Subsequently, we employ data visualization techniques to delve deeper into the dataset. Specifically, we leverage the Matplotlib and Seaborn libraries to create a histogram focusing

Code:

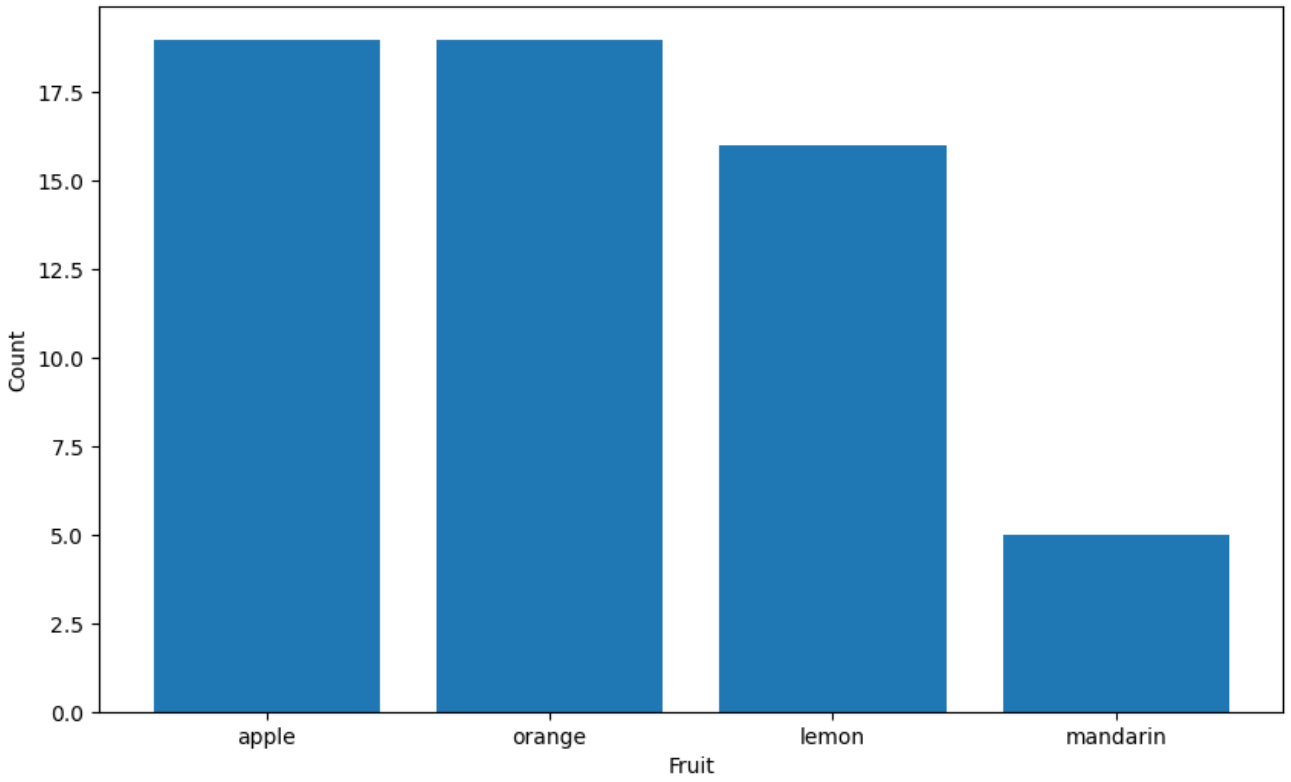
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

# show up charts when export notebooks
%matplotlib inline

data=pd.read_table('fruit_data_with_colors.t
xt')
data.head()

data.info()
data['fruit_name'].value_counts()


frequency=data['fruit_name'].value_counts()
plt.figure(figsize=(10,6))
plt.bar(frequency.index,height=frequency)
plt.ylabel("Count")
plt.xlabel("Fruit")
plt.show()
```

Output:

	fruit_label	fruit_name	fruit_subtype	mass	width	height	color_score
0	1	apple	granny_smith	192	8.4	7.3	0.55
1	1	apple	granny_smith	180	8.0	6.8	0.59
2	1	apple	granny_smith	176	7.4	7.2	0.60
3	2	mandarin	mandarin	86	6.2	4.7	0.80
4	2	mandarin	mandarin	84	6.0	4.6	0.79

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 59 entries, 0 to 58
Data columns (total 7 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   fruit_label    59 non-null     int64
 1   fruit_name     59 non-null     object
 2   fruit_subtype  59 non-null     object
 3   mass           59 non-null     int64
 4   width          59 non-null     float64
 5   height         59 non-null     float64
 6   color_score    59 non-null     float64
dtypes: float64(3), int64(2), object(2)
memory usage: 3.4+ KB

fruit_name
apple       19
orange      19
lemon       16
mandarin     5
Name: count, dtype: int64
```

Graph Visualization:



Data Preparation:

- The initial step in data preparation involves importing essential libraries such as pandas, numpy, and seaborn to facilitate data manipulation and visualization tasks.

- A custom function named outlier_z_score is defined to detect outliers in numerical columns utilizing the Z-Score method, aiding in identifying data points significantly deviating from the mean.

- The outlier_z_score function is then applied to target numerical columns within the dataset, flagging potential outlier rows based on their Z-Score values.

- Following outlier detection, a list containing rows identified as outliers is generated, highlighting the data points requiring further scrutiny or action.

- Subsequently, the identified outlier rows are removed from the dataset, ensuring data integrity and mitigating the potential influence of outliers on subsequent analyses or modeling efforts.

- Key statistical information, such as the lengths of the original dataset and the outlier-free dataset, along with the difference in lengths, is computed and displayed. This provides valuable insights into the impact of outlier removal on the dataset size and distribution.

- Visual representations in the form of boxplots are generated to visualize the distribution of the target variable 'mpg' before and after outlier removal, aiding in understanding the effects of data preprocessing on key variables.

- Handling missing values is addressed by first identifying and replacing placeholder values (e.g., '?' denoting missing data) with NaN (Not a Number) values in relevant columns such as 'horsepower.'

- A utility function named missing_values_table is defined to systematically analyze and summarize missing values across the dataset, facilitating informed decision-making regarding data imputation strategies.

- The KNNImputer is employed to impute missing values specifically in the 'horsepower' column, leveraging the K-Nearest Neighbors algorithm to estimate and fill in missing values based on neighboring data points.

- Additionally, data type conversions and categorical variable preprocessing tasks such as one-hot encoding (if required) are prepared within the codebase, ensuring data readiness for subsequent modeling phases while maintaining flexibility and scalability in data handling processes.

Code:

```
# Outlier function with threshold 2. Function to get list of outliers
def outliers_z_score(df):
threshold = 2

mean = np.mean(df)
std = np.std(df)
z_scores = [(y - mean) / std for y in df] #Used a Z-Score to remove the outliers
return np.where(np.abs(z_scores) > threshold)

# Selecting only the numerical columns in data set
my_list = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
num_columns = list(data.select_dtypes(include=my_list).columns)
numerical_columns = data[num_columns]
numerical_columns.head(3)

# Calling the outlier function and Calculating the outlier of dataset
outlier_list = numerical_columns.apply(lambda x: outliers_z_score(x))
outlier_list

# Making outlier list ot dataframe
df_of_outlier = outlier_list.iloc[0]
df_of_outlier = pd.DataFrame(df_of_outlier)
df_of_outlier.columns = ['Rows_to_exclude']
df_of_outlier

# Convert all values from column Rows_to_exclude to a numpy array
outlier_list_final = df_of_outlier['Rows_to_exclude'].to_numpy()

# Concatenate a whole sequence of arrays
outlier_list_final = np.concatenate( outlier_list_final, axis=0 )

# Drop duplicate values
outlier_list_final_unique = set(outlier_list_final)
outlier_list_final_unique

# Removing outliers from the dataset
filter_rows_to_exclude = data.index.isin(outlier_list_final_unique)
fruits = data[~filter_rows_to_exclude]
fruits.shape


frequency=fruits['fruit_name'].value_counts()
plt.figure(figsize=(10,6))
plt.bar(frequency.index,height=frequency)
plt.ylabel("Count")
plt.xlabel("Fruit")
plt.show()
```

```
data.drop('fruit_label', axis=1).plot(kind='box', subplots=True, layout=(2,2),
sharex=False, sharey=False, figsize=(9,9),
                                      title='Box Plot for each input variable
before outlier removal')
plt.savefig('fruits_box')
plt.show()


fruits.drop('fruit_label', axis=1).plot(kind='box', subplots=True, layout=(2,2),
sharex=False, sharey=False, figsize=(9,9),
                                      title='Box Plot for each input variable
after outlier removal')
plt.savefig('fruits_box')
plt.show()
```

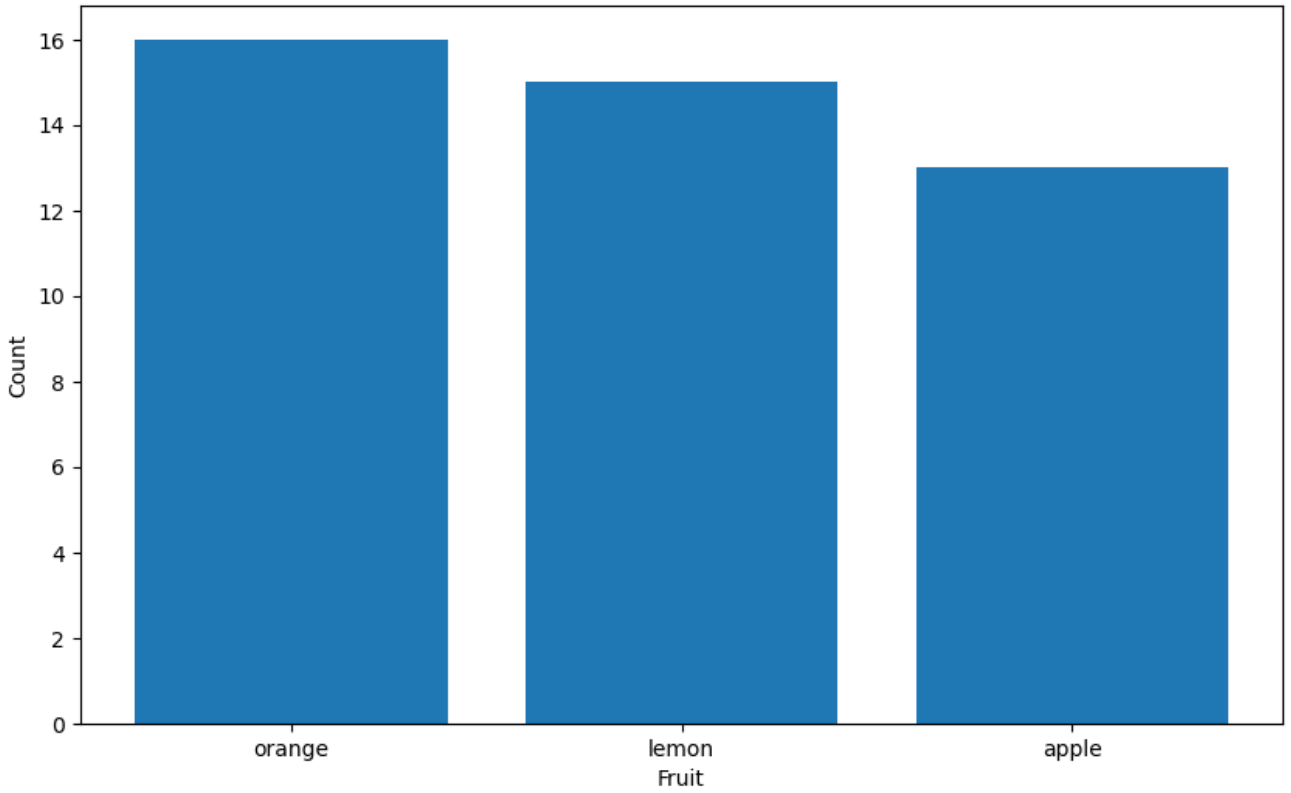
Output:

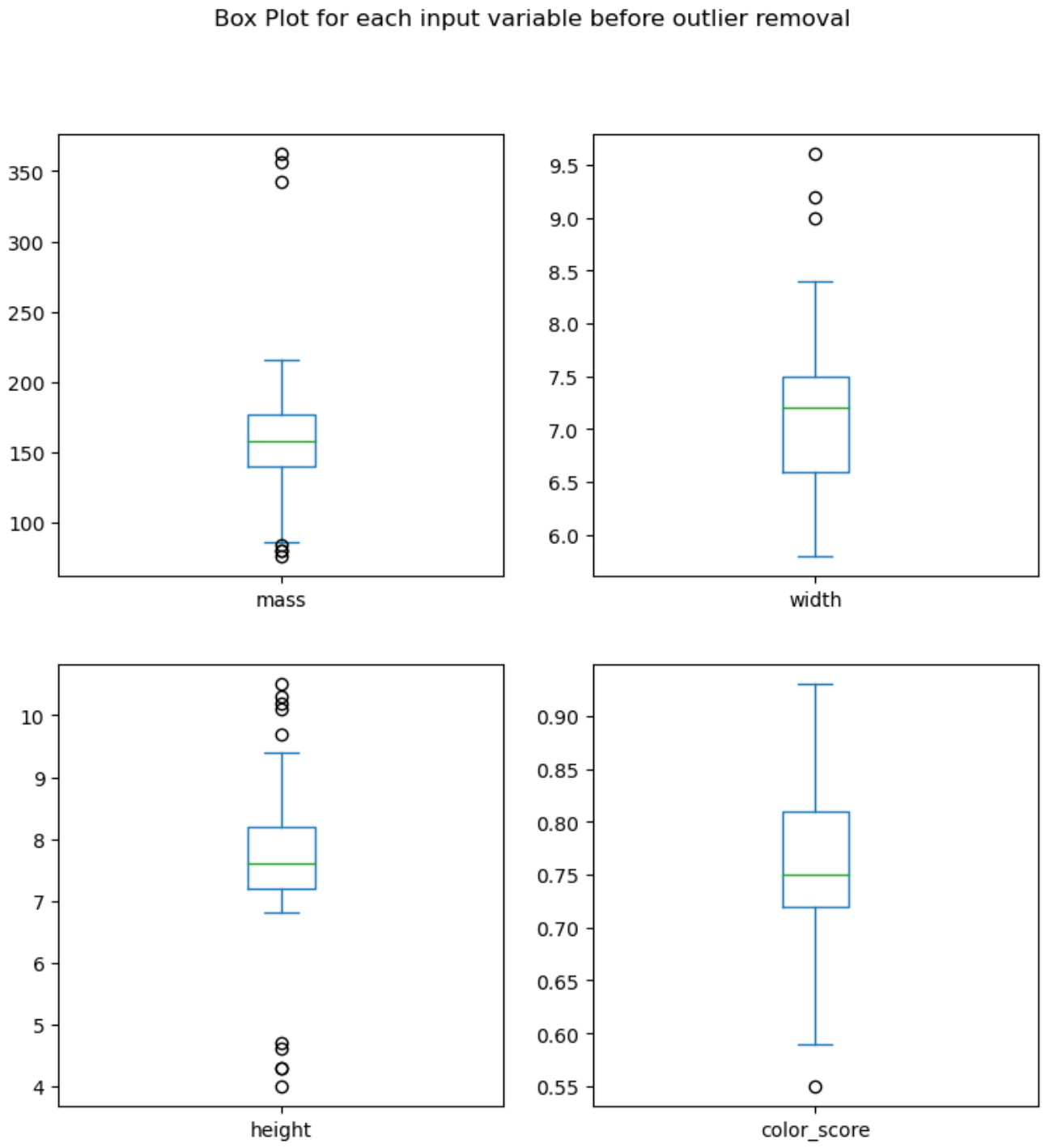
	fruit_label	mass	width	height	color_score
0	1	192	8.4	7.3	0.55
1	1	180	8.0	6.8	0.59
2	1	176	7.4	7.2	0.60

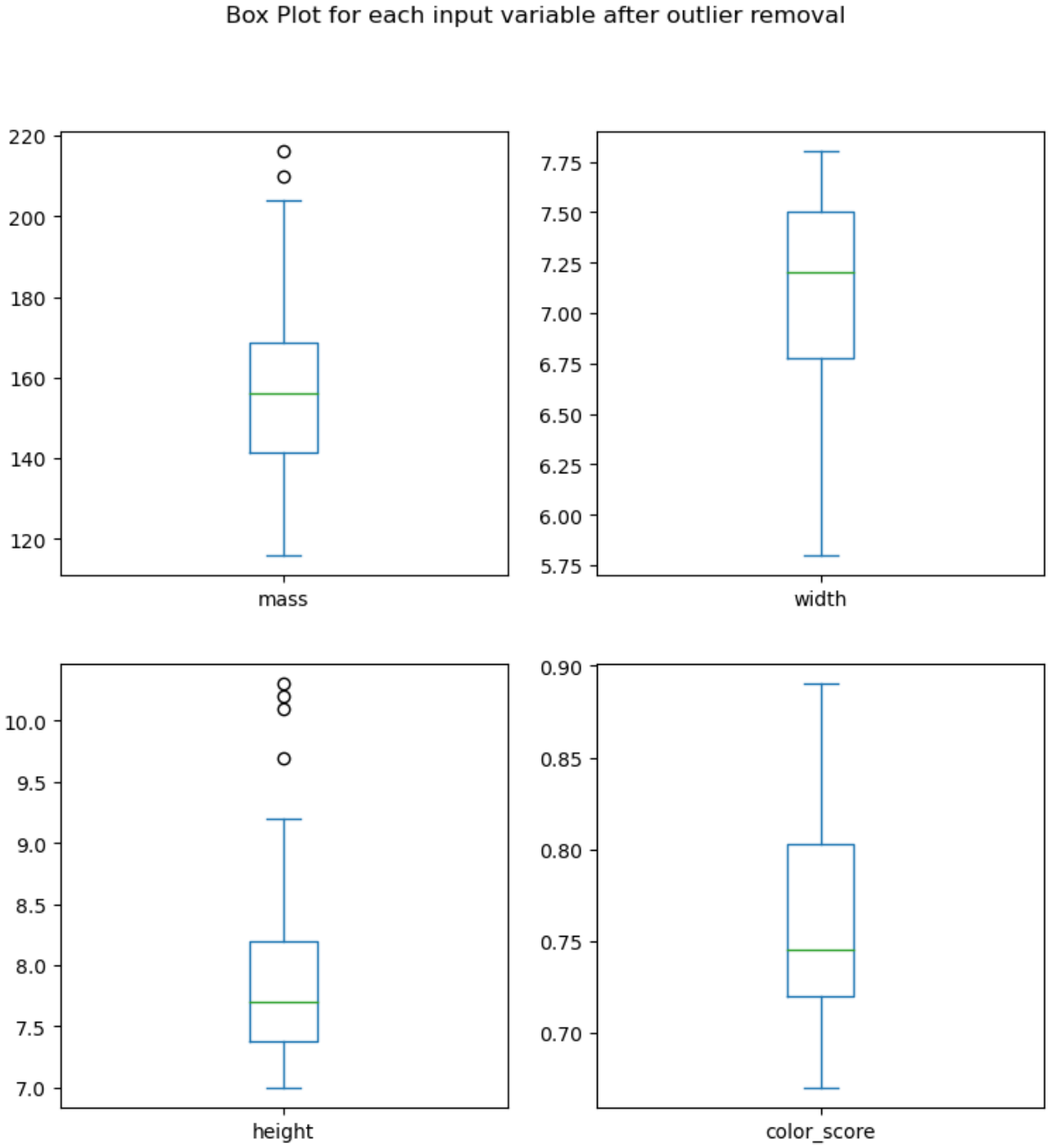
	fruit_label	mass	width	height	color_score
0	[]	[24, 25, 26]	[24, 25, 26]	[3, 4, 5, 6, 7, 44]	[0, 1, 2, 8, 10, 11]

	Rows_to_exclude
fruit_label	[]
mass	[24, 25, 26]
width	[24, 25, 26]
height	[3, 4, 5, 6, 7, 44]
color_score	[0, 1, 2, 8, 10, 11]

Graph Visualization:




Box Plot for each input variable before outlier removal
350
300
250
200
150
100
mass
9.5
9.0
8.5
8.0
7.5
7.0
6.5
6.0
width
10
9
8
7
6
5
4
height
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
color_score


Box Plot for each input variable after outlier removal
220
200
180
160
140
120
mass
7.75
7.50
7.25
7.00
6.75
6.50
6.25
6.00
5.75
width
10.0
9.5
9.0
8.5
8.0
7.5
7.0
height
0.90
0.85
0.80
0.75
0.70
color_score

Correlation:

- Correlation is a statistical measure that describes the relationship between two variables. It indicates both the strength and direction of the relationship. Here are the steps to find the correlation between two variables:

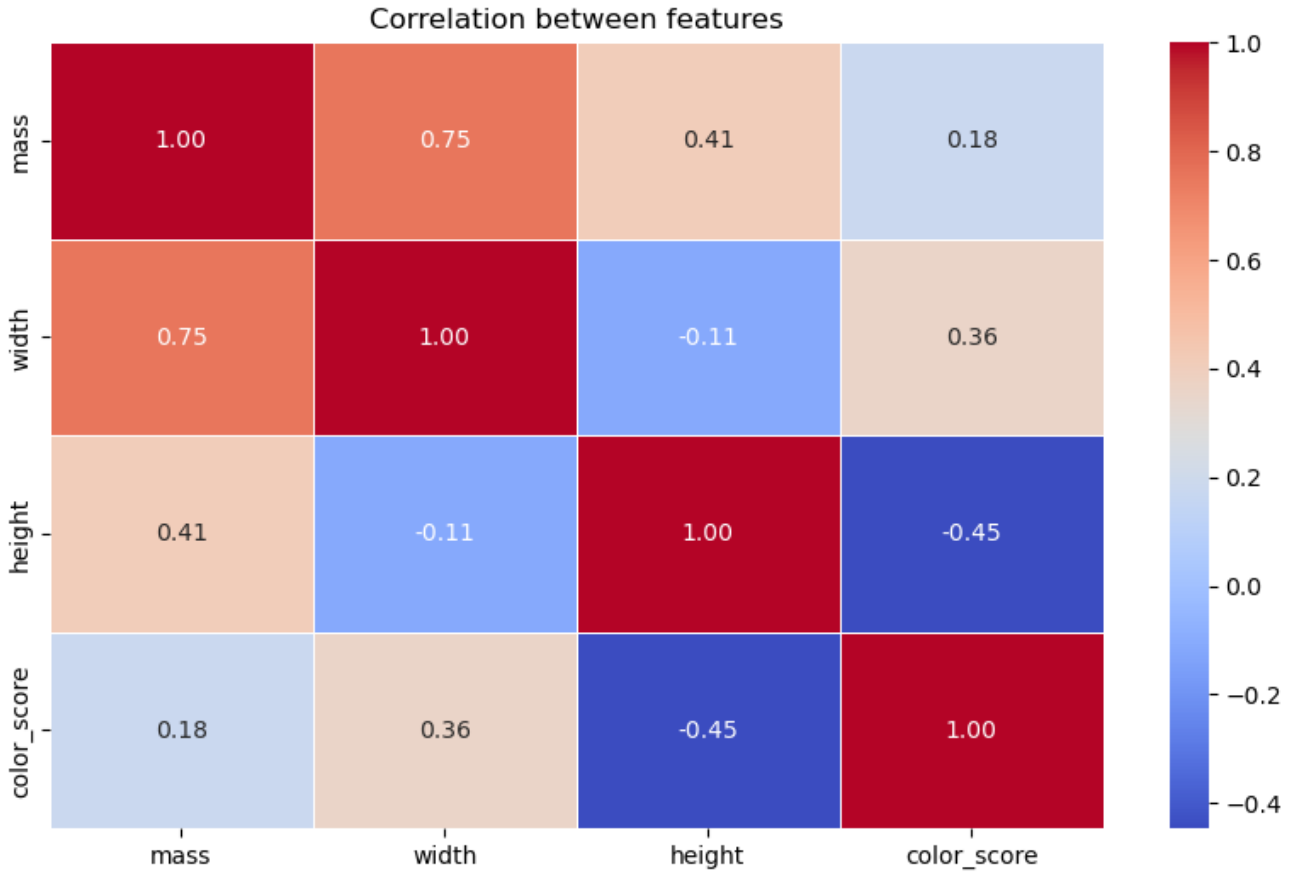
- Collect Data: Gather data for the two variables of interest. Ensure that the data is numeric and represents the same set of observations.
- Calculate the Mean: Find the mean (average) of each variable.
- Calculate the Deviation: For each data point, find the deviation from the mean for both variables.
- Calculate the Product of Deviations: Multiply the deviations of the two variables for each data point.
- Calculate the Sum of the Products of Deviations: Add up all the products of deviations calculated in the previous step.
- Calculate the Standard Deviation: Find the standard deviation for each variable.
- Calculate the Correlation Coefficient: Divide the sum of the products of deviations by the product of the standard deviations of the two variables. This gives you the correlation coefficient, which ranges from -1 to 1.
    - A correlation of 1 indicates a perfect positive linear relationship.
    - A correlation of -1 indicates a perfect negative linear relationship.
    - A correlation of 0 indicates no linear relationship.
- Interpret the Correlation Coefficient: Based on the value of the correlation coefficient, interpret the relationship between the two variables.

Code:

```
feature_names = ['mass', 'width', 'height', 'color_score']

df=fruits[feature_names]

correlation_matrix = df.corr()

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

sns.heatmap(correlation_matrix,annot=True,cmap='coolwarm',fmt='.2f',linewidths=0.5)

plt.title('Correlation between features')

plt.show()
```

Output:

Graph Visualization:



Analysis of Discrete and Continuous Variables:

Dataset Splitting:

- The code segment divides the features (X) and the target variable (y) into separate training and validation sets, a crucial step in machine learning model development.
- Utilizing the train_test_split function, the data is split into two distinct sets: a training set (X_train and y_train) used for model training and a validation set (X_val and y_val) used for model evaluation.
- The test_size parameter specifies the proportion of the dataset allocated to the validation set, with a value of 0.2 indicating 20% of the data is reserved for validation purposes.
- To ensure reproducibility and consistent results across runs, the random_state parameter is set, fixing the random seed used for the data split process.
- This splitting strategy ensures that the model is trained on a subset of the data and evaluated on unseen data, helping to assess its generalization performance and avoid overfitting to the training data. It also facilitates the analysis of discrete and continuous variables within each dataset subset, enabling tailored preprocessing and analysis techniques as needed.

Code:

```
import pandas as pd
from sklearn.model_selection import train_test_split
# Assuming X is your features and y is your target variable
# Split the data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Modeling and Evaluation in Classification:

Logistic Regression:

Description: Logistic regression is a statistical method used for binary classification. It estimates probabilities using a logistic function to model the relationship between the dependent variable and one or more independent variables.

Modeling: In logistic regression, you estimate the coefficients of the independent variables to fit a logistic curve that best predicts the probability of the binary outcome.

Evaluation: Common evaluation metrics for logistic regression include accuracy, precision, recall, F1 score, and ROC-AUC score.

Support Vector Machines (SVM):

Description: SVM is a supervised machine learning algorithm that can be used for both classification and regression tasks. It finds the hyperplane that best separates the classes in the feature space.

Types: There are four main types of SVM kernels: linear, polynomial, radial basis function (RBF), and sigmoid.

Modeling: SVM constructs a hyperplane or set of hyperplanes in a high-dimensional space that can be used for classification, regression, or other tasks.

Evaluation: SVM models are evaluated using metrics like accuracy, precision, recall, F1 score, and ROC-AUC score.

K-Nearest Neighbors (KNN):

Description: KNN is a simple, instance-based learning algorithm used for classification and regression. It classifies new data points based on the majority class of their k nearest neighbors.

Modeling: KNN requires no training phase; instead, it stores all instances and makes predictions based on a similarity measure (e.g., Euclidean distance) between input data and stored instances.

Evaluation: KNN models are evaluated using metrics such as accuracy, precision, recall, F1 score, and ROC-AUC score.

Code:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn import svm
X=fruits[feature_names]
y=fruits['fruit_label']
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Logistic Regression
logreg = LogisticRegression(multi_class='auto',solver='lbfgs')
logreg.fit(X_train, y_train)
predictions=logreg.predict(X_test)
accuracy=accuracy_score(y_test,predictions)
print("Multi-Class Logistic Regression Accuracy:", accuracy)

# SVM Classifier
svm_classifier = svm.SVC(kernel='linear')
svm_classifier.fit(X_train, y_train)
svm_predictions = svm_classifier.predict(X_test)
svm_accuracy = accuracy_score(y_test, svm_predictions)
print("SVM Accuracy of linear kernel:", svm_accuracy)

# SVM Classifier
svm_classifier = svm.SVC(kernel='rbf')
svm_classifier.fit(X_train, y_train)
svm_predictions = svm_classifier.predict(X_test)
svm_accuracy = accuracy_score(y_test, svm_predictions)
print("SVM Accuracy of rbf kernel:", svm_accuracy)

# SVM Classifier
svm_classifier = svm.SVC(kernel='poly')
svm_classifier.fit(X_train, y_train)
svm_predictions = svm_classifier.predict(X_test)
svm_accuracy = accuracy_score(y_test, svm_predictions)
print("SVM Accuracy for polynomial kernel:", svm_accuracy)

svm_classifier = svm.SVC(kernel='sigmoid')
svm_classifier.fit(X_train, y_train)
svm_predictions = svm_classifier.predict(X_test)
svm_accuracy = accuracy_score(y_test, svm_predictions)
print("SVM Accuracy for sigmoid kernel:", svm_accuracy)

# KNN
knn=KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train,y_train)
predictions=knn.predict(X_test)
accuracy=accuracy_score(y_test,predictions)
print("k-NN Accuracy:", accuracy)
```

Output

```
Multi-Class Logistic Regression Accuracy: 0.7272727272727273


SVM Accuracy of linear kernel: 0.5454545454545454
SVM Accuracy of rbf kernel: 1.0
SVM Accuracy for polynomial kernel: 1.0
SVM Accuracy for sigmoid kernel: 0.6363636363636364

k-NN Accuracy: 0.8181818181818182
```

Hyperparameter Tuning Process Overview:

Hyperparameter tuning is the process of selecting the best set of hyperparameters for a machine learning model to optimize its performance. Hyperparameters are parameters that are set before the learning process begins and control aspects of the learning process.

Why Hyperparameter Tuning?

- Hyperparameters significantly impact the performance of a model.
- Different hyperparameter values can lead to vastly different model performance.
- Tuning helps in finding the optimal balance between model complexity and generalization.

Code:

```
from sklearn.model_selection import GridSearchCV


# Define hyperparameters grid for each classifier
svm_param_grid = {'C': [0.1, 1, 10, 100], 'gamma': [0.1, 0.01, 0.001], 'kernel': ['linear', 'rbf', 'sigmoid', 'poly']}
logistic_param_grid = {'C': [0.1, 1, 10, 100], 'solver': ['lbfgs', 'liblinear']}
knn_param_grid = {'n_neighbors': [3, 5, 7, 9, 11], 'weights': ['uniform', 'distance']}


# Perform grid search for each classifier
svm_grid_search = GridSearchCV(svm_classifier, svm_param_grid, cv=5)
svm_grid_search.fit(X_train, y_train)


logistic_grid_search = GridSearchCV(logreg, logistic_param_grid, cv=5)
logistic_grid_search.fit(X_train, y_train)


knn_grid_search = GridSearchCV(knn, knn_param_grid, cv=5)
knn_grid_search.fit(X_train, y_train)


# Get best hyperparameters and evaluate on test set
best_svm_classifier = svm_grid_search.best_estimator_
best_svm_predictions = best_svm_classifier.predict(X_test)
best_svm_accuracy = accuracy_score(y_test, best_svm_predictions)
print("Best SVM Accuracy:", best_svm_accuracy)
print("Best SVM Parameters:", svm_grid_search.best_params_)
```

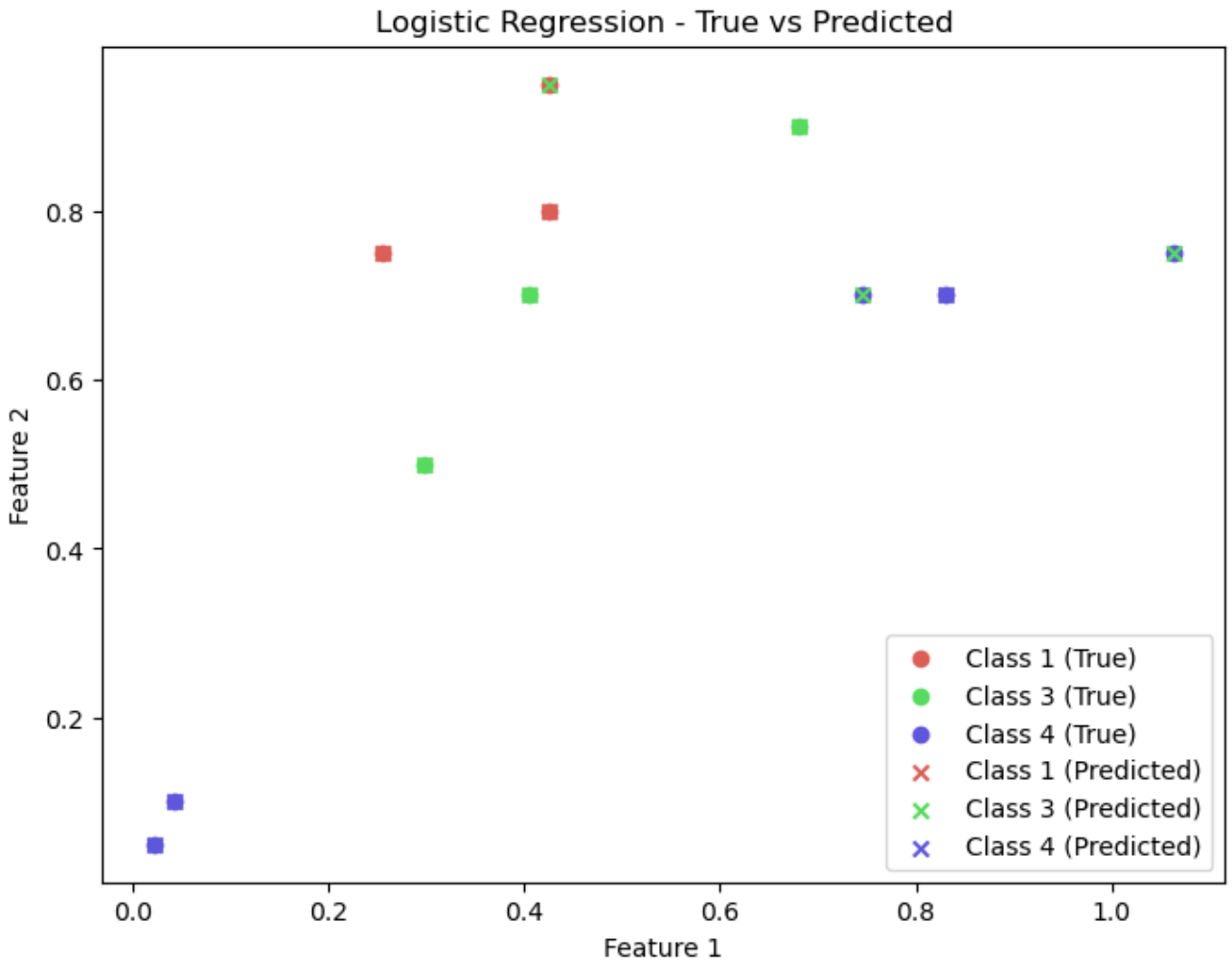
```
best_logistic_regression_classifier = logistic_grid_search.best_estimator_
best_logistic_regression_predictions =
best_logistic_regression_classifier.predict(X_test)
best_logistic_regression_accuracy = accuracy_score(y_test,
best_logistic_regression_predictions)
print("Best Logistic Regression Accuracy:", best_logistic_regression_accuracy)
print("Best Logistic Regression Parameters:", logistic_grid_search.best_params_)


best_knn_classifier = knn_grid_search.best_estimator_
best_knn_predictions = best_knn_classifier.predict(X_test)
best_knn_accuracy = accuracy_score(y_test, best_knn_predictions)
print("Best k-NN Accuracy:", best_knn_accuracy)
print("Best k-NN Parameters:", knn_grid_search.best_params_)
```

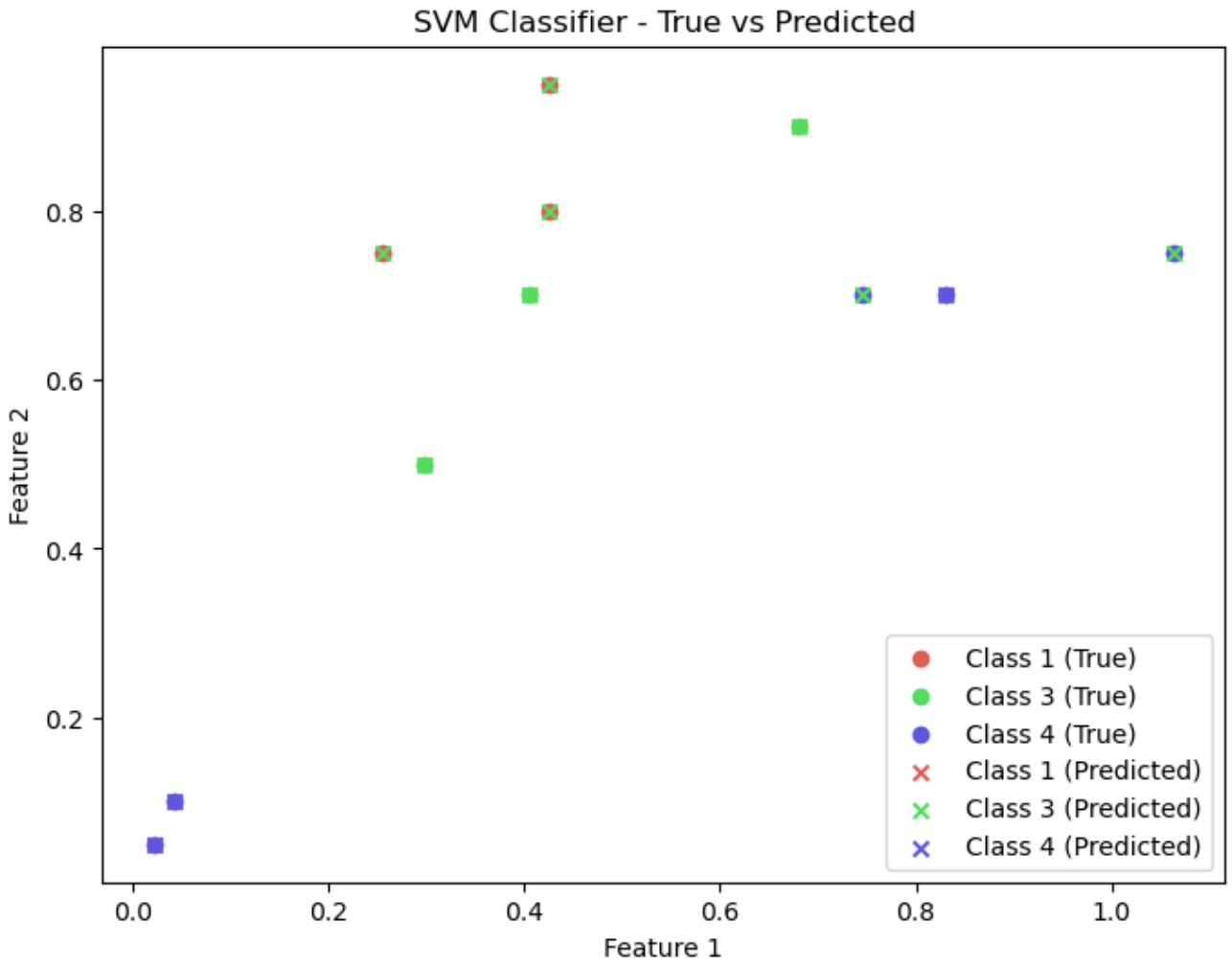
Results:

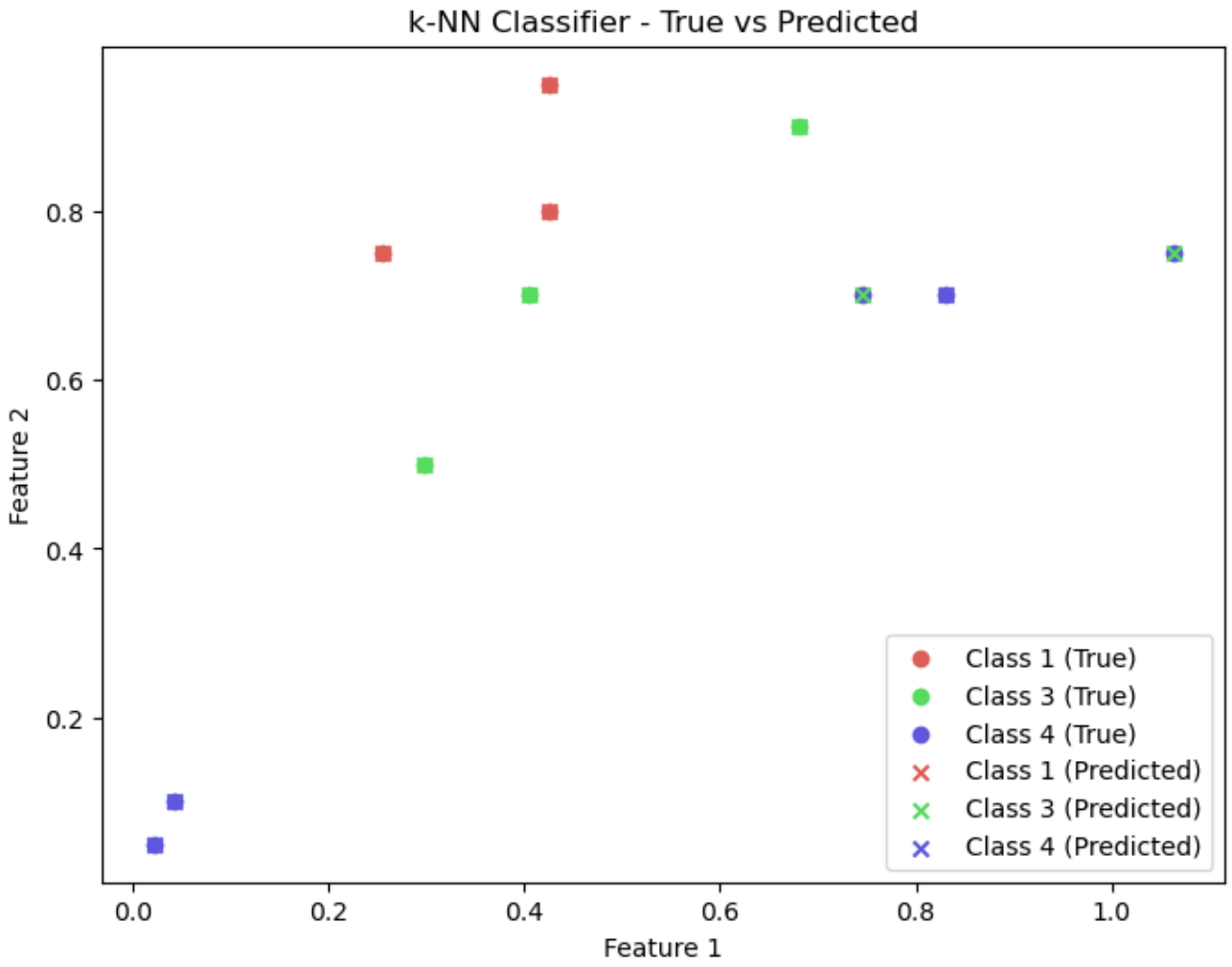
Output:

```
Best SVM Accuracy: 0.9090909090909091
Best SVM Parameters: {'C': 100, 'gamma': 0.1, 'kernel': 'rbf'}
Best Logistic Regression Accuracy: 0.9090909090909091
Best Logistic Regression Parameters: {'C': 100, 'solver': 'liblinear'}
Best k-NN Accuracy: 1.0
Best k-NN Parameters: {'n_neighbors': 3, 'weights': 'distance'}
```

Comparison of all classifiers:




SVM Classifier - True vs Predicted
Feature 2
Feature 1
0.0
0.2
0.4
0.6
0.8
1.0
Class 1 (True)
Class 3 (True)
Class 4 (True)
Class 1 (Predicted)
Class 3 (Predicted)
Class 4 (Predicted)


k-NN Classifier - True vs Predicted
Feature 2
Feature 1
0.0
0.2
0.4
0.6
0.8
1.0
Class 1 (True)
Class 3 (True)
Class 4 (True)
Class 1 (Predicted)
Class 3 (Predicted)
Class 4 (Predicted)

CRISP – DM Clustering

Data Understanding:

- The initial step in our data exploration process involves mounting Google Drive to access the dataset stored in the cloud, named 'fruit_data_with_colors.txt'. This dataset contains information about a fruit and it’s information. Upon loading the dataset using Pandas, we generate a concise summary showcasing its dimensions and preview a sample of its initial rows to grasp its structure and content. Following this, we present summary statistics that highlight key numerical attributes such as mean, standard deviation, and quartile values. These statistics provide valuable insights into the data's distribution and variability, aiding in our understanding of its characteristics.

- Subsequently, we employ data visualization techniques to delve deeper into the dataset. Specifically, we leverage the Matplotlib and Seaborn libraries to create a histogram focusing

Code:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

# show up charts when export notebooks
%matplotlib inline


data=pd.read_csv('Pokemon.csv')

data.head()

data.isna().any()
```

Output:

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	318	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	405	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	525	80	82	83	100	100	80	1	False
3	3	VenusaurMega Venusaur	Grass	Poison	625	80	100	123	122	120	80	1	False
4	4	Charmander	Fire	NaN	309	39	52	43	60	50	65	1	False

```
#             False
Name          False
Type 1        False
Type 2         True
Total         False
HP            False
Attack        False
Defense       False
Sp. Atk       False
Sp. Def       False
Speed         False
Generation    False
Legendary     False
dtype: bool
```

Data Preparation:

Only a part of the data is selected for clustering and the others are discarded.

Code:

```
types = data['Type 1'].isin(['Grass', 'Fire', 'Water']) # True for pokemon with
type1 as 'Grass', 'Fire or 'Water
types


data[types] # Pokemon with type1 as 'Grass', 'Fire or 'Water


drop_cols=['Type 1','Type 2','Generation','Legendary','#']
pokemon=data.drop(columns=drop_cols)

pokemon.head()
```

Output:

```
0        True
1        True
2        True
3        True
4        True
        ...
795     False
796     False
797     False
798     False
799      True
Name: Type 1, Length: 800, dtype: bool
```

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	318	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	405	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	525	80	82	83	100	100	80	1	False
3	3	VenusaurMega Venusaur	Grass	Poison	625	80	100	123	122	120	80	1	False
4	4	Charmander	Fire	NaN	309	39	52	43	60	50	65	1	False
...	...	...	...	...	...	...	...	...	...	...	...	...	...
740	672	Skiddo	Grass	NaN	350	66	65	48	62	57	52	6	False
741	673	Gogoat	Grass	NaN	531	123	100	62	97	81	68	6	False
762	692	Clauncher	Water	NaN	330	50	53	62	58	63	44	6	False
763	693	Clawitzer	Water	NaN	500	71	73	88	120	89	59	6	False
799	721	Volcanion	Fire	Water	600	80	110	120	130	90	70	6	True

234 rows × 13 columns

	Name	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	cluster	cen_x	cen_y	c
0	Bulbasaur	318	45	49	49	65	65	45	0	58.021739	53.285024	blue
1	Ivysaur	405	60	62	63	80	80	60	0	58.021739	53.285024	blue
2	Venusaur	525	80	82	83	100	100	80	2	80.423913	110.304348	red
3	VenusaurMega Venusaur	625	80	100	123	122	120	80	2	80.423913	110.304348	red
4	Charmander	309	39	52	43	60	50	65	0	58.021739	53.285024	blue

Clustering:

Clustering is a fundamental technique in unsupervised machine learning where data points are grouped together based on their similarities. The primary goal of clustering is to partition a dataset into groups or clusters, such that data points within the same cluster are more similar to each other than to those in other clusters. This allows for identifying patterns, structure, and relationships within the data without prior knowledge of the class labels.

Types of Clustering Algorithms:

Partitioning Methods: These algorithms partition the data into distinct clusters. Examples include k-means, k-medoids, and Gaussian mixture models.

Hierarchical Methods: These algorithms create a hierarchy of clusters, either through agglomerative (bottom-up) or divisive (top-down) approaches.
Density-based Methods: These algorithms find clusters by identifying areas of high density within the data space. DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a popular density-based algorithm.

Model-based Methods: These algorithms assume that the data is generated by a mixture of underlying probability distributions. Examples include Gaussian mixture models and hierarchical Dirichlet processes.

Evaluation: Clustering algorithms need to be evaluated to assess the quality of the clustering results. Common evaluation metrics include silhouette score, Davies-Bouldin index, and the purity of clusters.

Applications: Clustering finds applications in various fields such as:

- Customer segmentation in marketing
- Image segmentation in computer vision
- Anomaly detection in cybersecurity
- Document clustering in natural language processing
- Genetics and biological data analysis
- Social network analysis

Code:

```
from sklearn.cluster import KMeans
# k means
kmeans = KMeans(n_clusters=3, random_state=0)
pokemon['cluster'] = kmeans.fit_predict(pokemon[['Attack', 'Defense']])
# get centroids
centroids = kmeans.cluster_centers_
cen_x = [i[0] for i in centroids]
cen_y = [i[1] for i in centroids]
print(cen_x)
## add to df
pokemon['cen_x'] = pokemon.cluster.map({0:cen_x[0], 1:cen_x[1], 2:cen_x[2]})
pokemon['cen_y'] = pokemon.cluster.map({0:cen_y[0], 1:cen_y[1], 2:cen_y[2]})
# define and map colors
colors = ['blue', 'green', 'red']
pokemon['c'] = pokemon.cluster.map({0:colors[0], 1:colors[1], 2:colors[2]})


pokemon.head(20)


plt.figure(figsize=(10,6))
plt.scatter(pokemon.Attack, pokemon.Defense, c=pokemon.c, alpha = 0.6, s=5)
plt.xlabel('Attack')
plt.ylabel('Defense')
plt.show(
```

Output:

	Name	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	cluster	cen_x	cen_y	c
0	Bulbasaur	318	45	49	49	65	65	45	0	58.021739	53.285024	blue
1	Ivysaur	405	60	62	63	80	80	60	0	58.021739	53.285024	blue
2	Venusaur	525	80	82	83	100	100	80	2	80.423913	110.304348	red
3	VenusaurMega Venusaur	625	80	100	123	122	120	80	2	80.423913	110.304348	red
4	Charmander	309	39	52	43	60	50	65	0	58.021739	53.285024	blue
5	Charmeleon	405	58	64	58	80	65	80	0	58.021739	53.285024	blue
6	Charizard	534	78	84	78	109	85	100	2	80.423913	110.304348	red
7	CharizardMega Charizard X	634	78	130	111	130	85	100	1	120.702970	82.762376	green
8	CharizardMega Charizard Y	634	78	104	78	159	115	100	1	120.702970	82.762376	green
9	Squirtle	314	44	48	65	50	64	43	0	58.021739	53.285024	blue
10	Wartortle	405	59	63	80	65	80	58	0	58.021739	53.285024	blue
11	Blastoise	530	79	83	100	85	105	78	2	80.423913	110.304348	red
12	BlastoiseMega Blastoise	630	79	103	120	135	115	78	2	80.423913	110.304348	red
13	Caterpie	195	45	30	35	20	20	45	0	58.021739	53.285024	blue
14	Metapod	205	50	20	55	25	25	30	0	58.021739	53.285024	blue
15	Butterfree	395	60	45	50	90	80	70	0	58.021739	53.285024	blue
16	Weedle	195	40	35	30	20	20	50	0	58.021739	53.285024	blue
17	Kakuna	205	45	25	50	25	25	35	0	58.021739	53.285024	blue
18	Beedrill	395	65	90	40	45	80	75	0	58.021739	53.285024	blue
19	BeedrillMega Beedrill	495	65	150	40	15	80	145	1	120.702970	82.762376	green

Graph Visualization: