

PSG COLLEGE OF TECHNOLOGY

**CRISP – DM REPORT**

**19Z610 – MACHINE LEARNING LABORATORY**

**NAME:** R K Ranjith

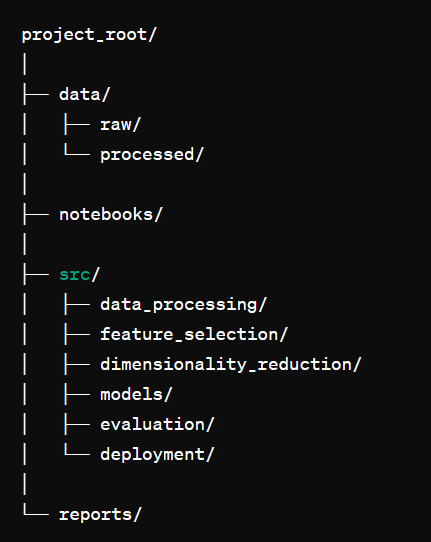
**ROLL NO.:** 21z335

**CLASS:** B.E CSE – G2

# 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())

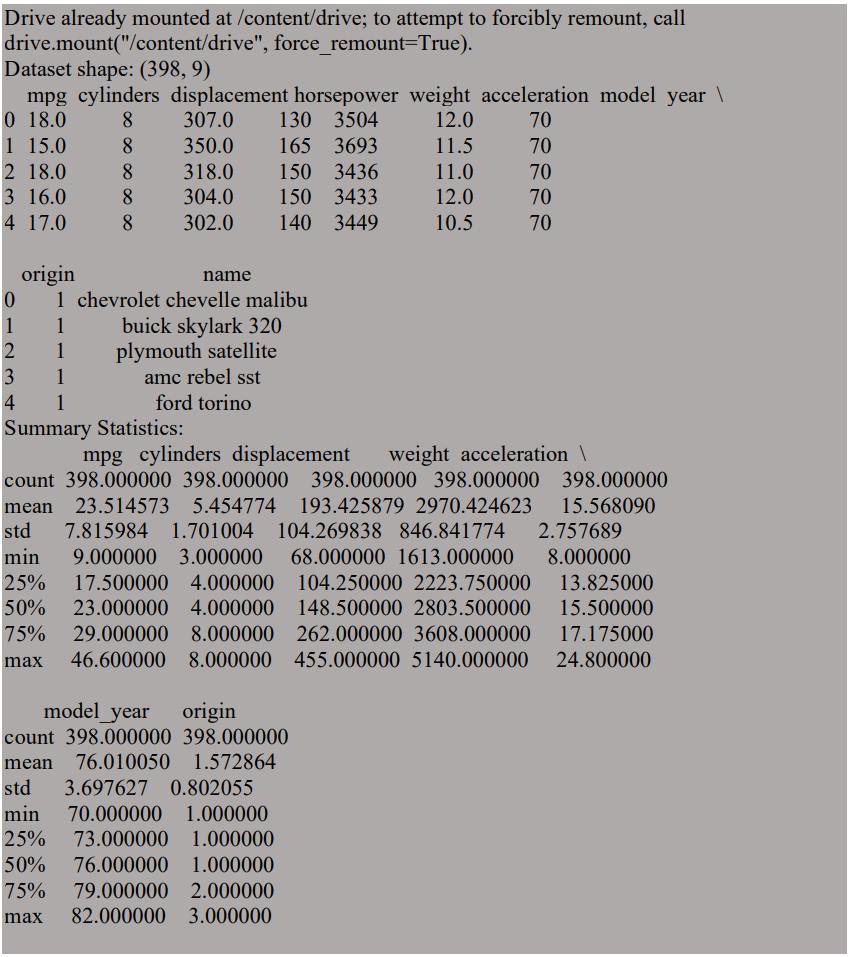
# Visualizations (example: Histogram of MPG) import matplotlib.pyplot as plt

import seaborn as sns

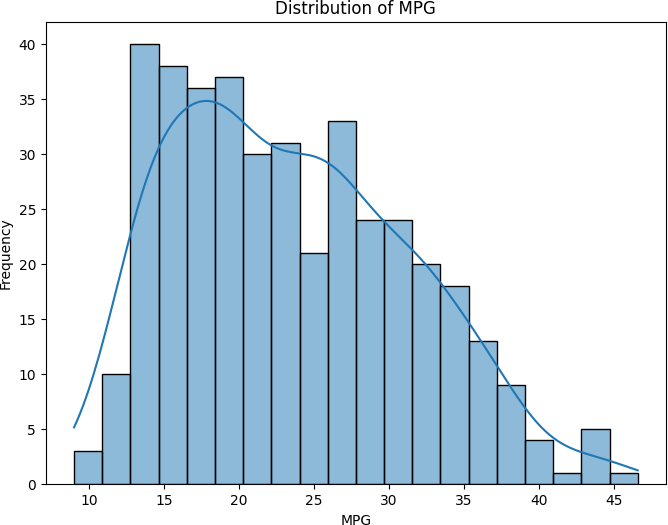
plt.figure(figsize=(8, 6)) sns.histplot(data['mpg'], bins=20, kde=True) plt.title('Distribution of MPG')

plt.xlabel('MPG') plt.ylabel('Frequency') plt.show()

## **Output:**



**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

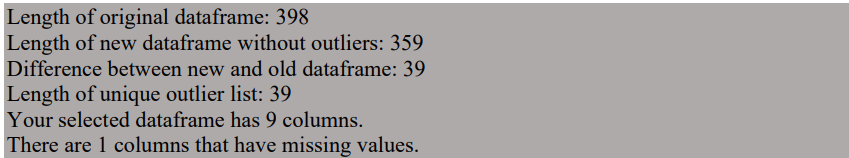
# One-hot encoding for categorical variables # non\_numeric\_columns =

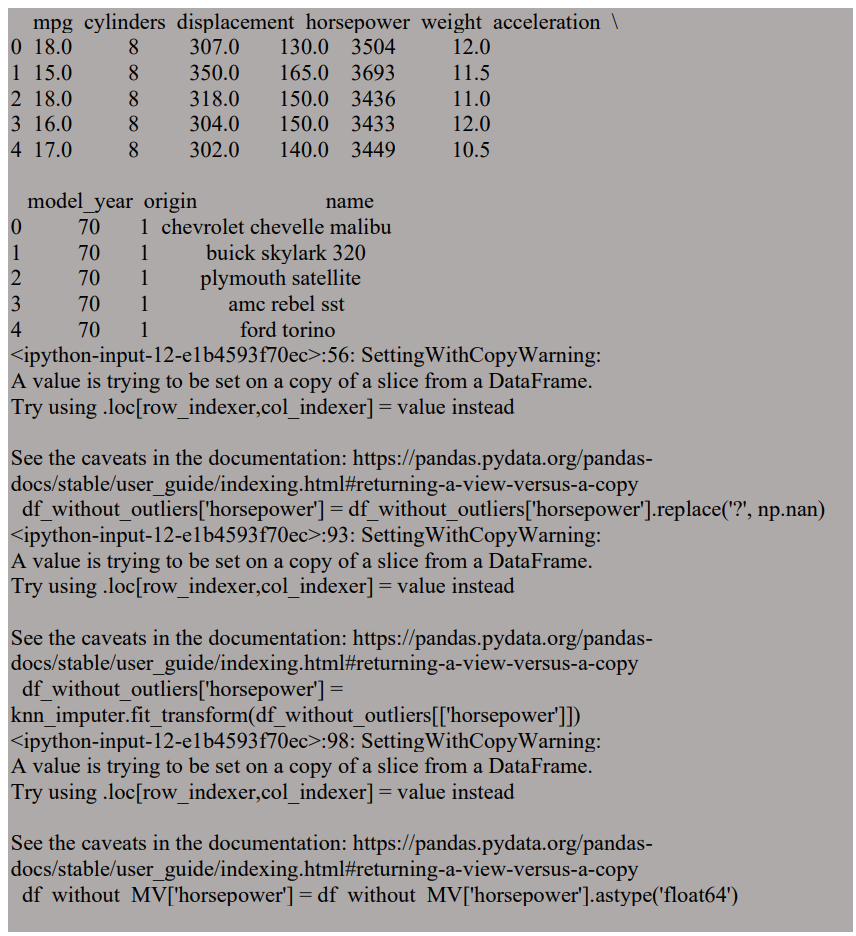
df\_without\_MV.select\_dtypes(include=['object']).columns

# df\_processed = pd.get\_dummies(df\_without\_MV, columns=non\_numeric\_columns) df\_processed = df\_without\_MV

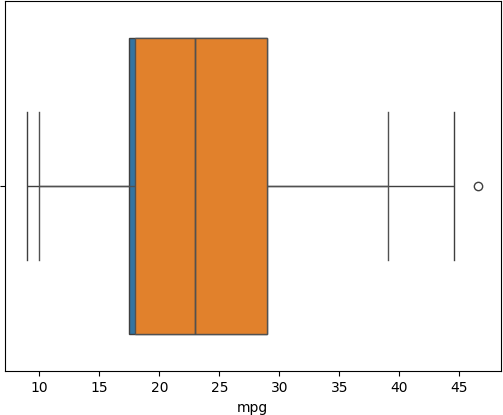
print(df\_processed.head())

# Output:





**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)

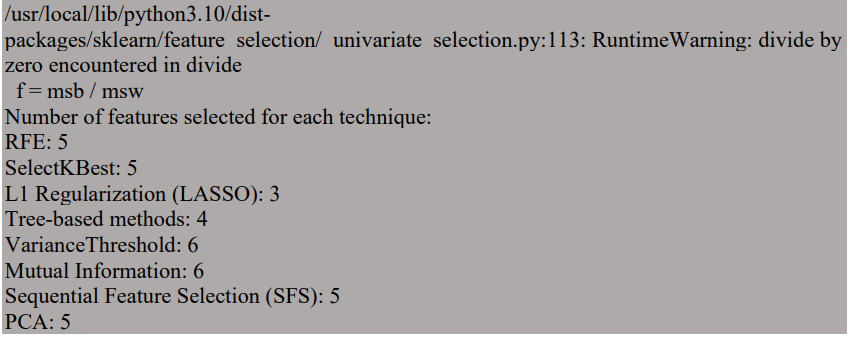
X\_pca\_selected = pca.fit\_transform(X\_subset)

# 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])

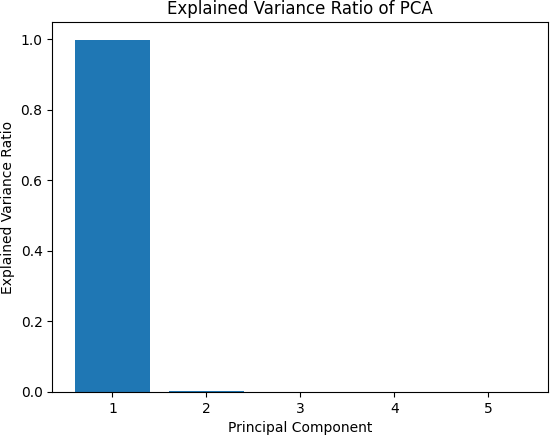
print("L1 Regularization (LASSO):", X\_lasso\_selected.shape[1]) print("Tree-based methods:", X\_tree\_selected.shape[1]) print("VarianceThreshold:", X\_var\_selected.shape[1]) print("Mutual Information:", X\_mi\_selected.shape[1])

print("Sequential Feature Selection (SFS):", X\_sfs\_selected.shape[1]) print("PCA:", X\_pca\_selected.shape[1])

**Output:**



**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

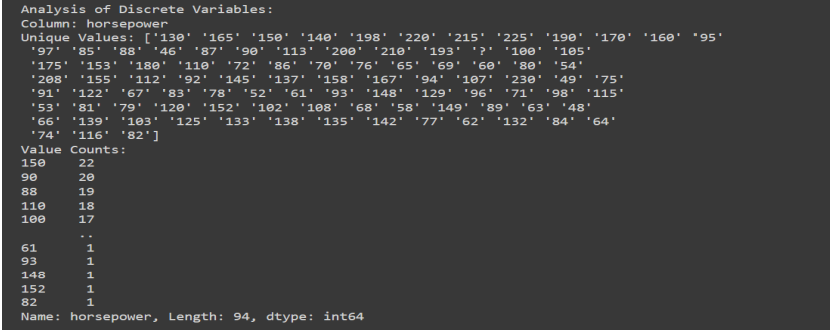
sns.countplot(x=col, data=data\_encoded) plt.title(f"Distribution of {col} (After Encoding)")

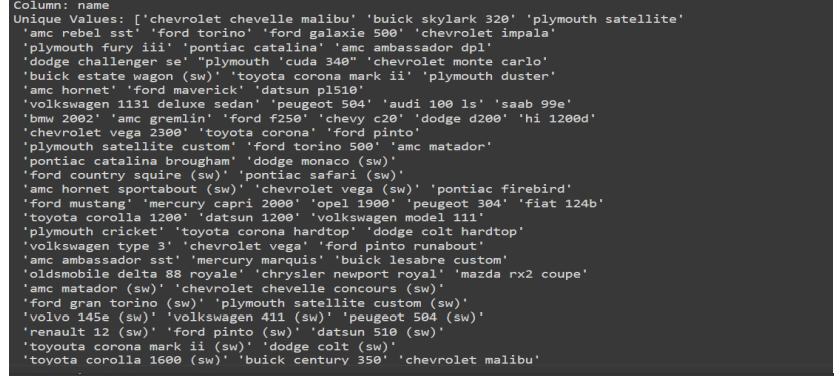
else:

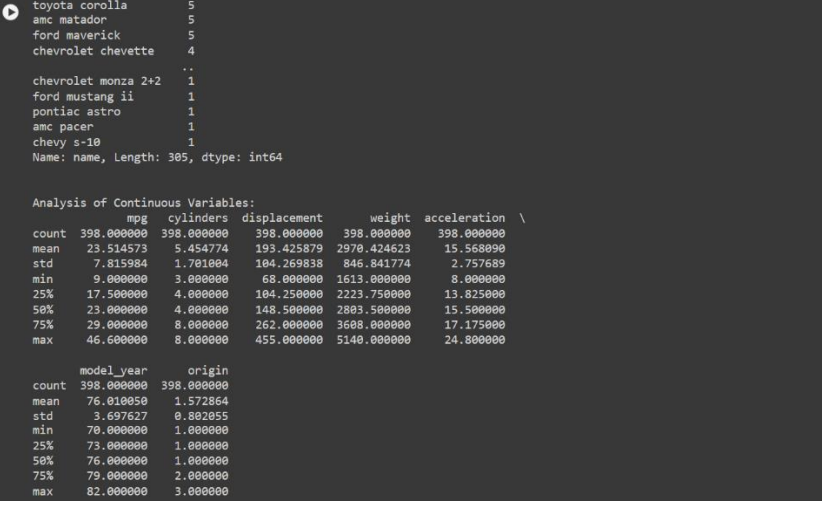
print(f"Column '{col}' not found in the encoded data.") plt.tight\_layout()

plt.show()

**Output:**







**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)

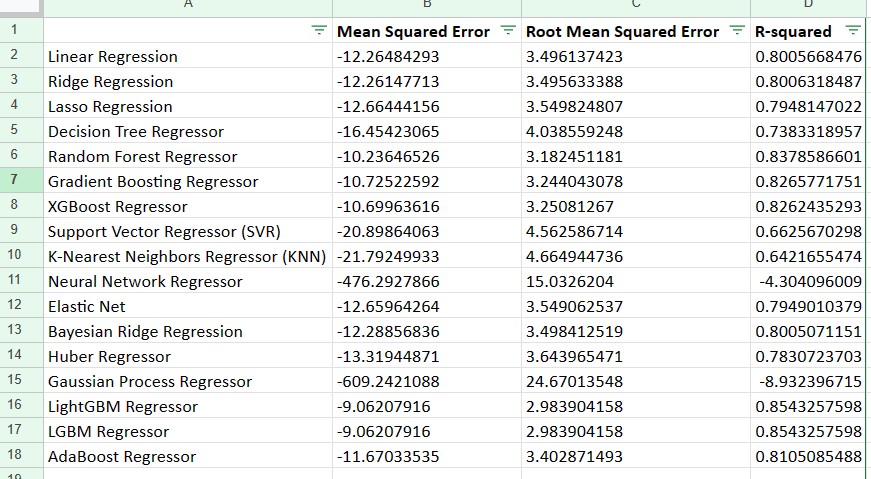
except Exception as e:

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:



**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}:

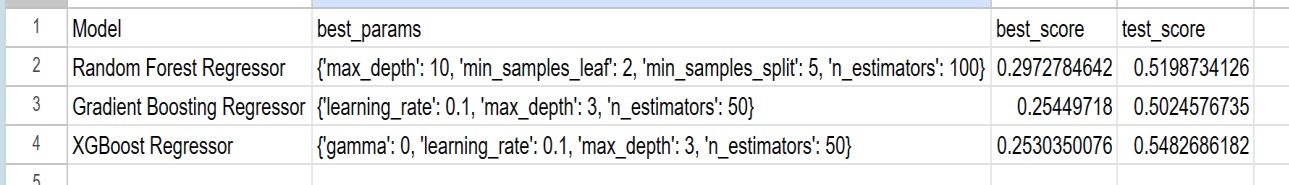
{results[name]['best\_score']}")

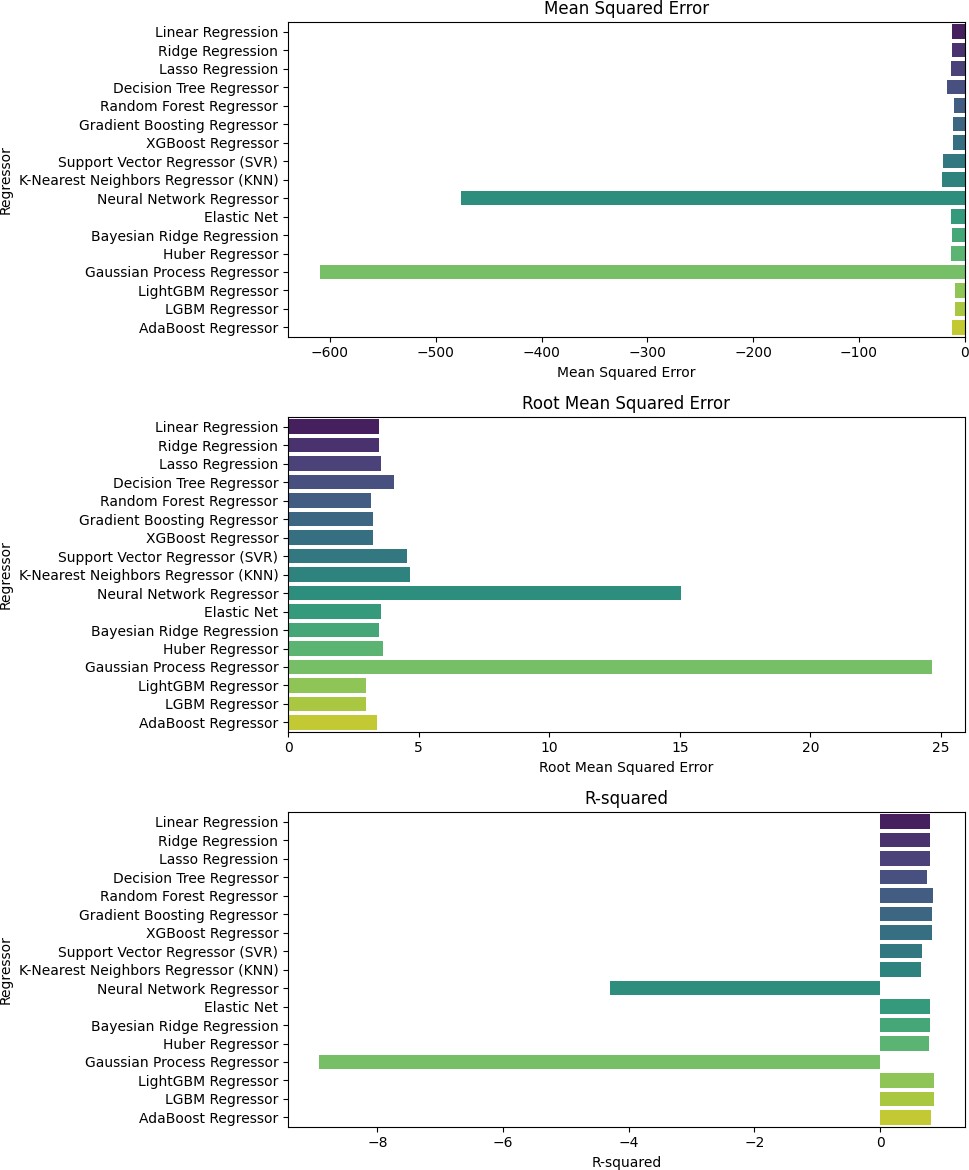
print(f"Test score for {name}: {results[name]['test\_score']}") print()

# 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:



**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)

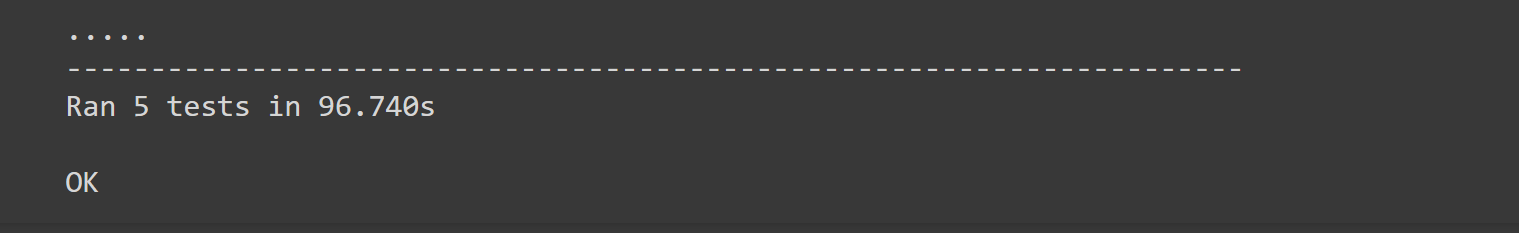
mse = mean\_squared\_error(self.y\_test, y\_pred)

self.assertTrue(mse < 10) # Adjust the threshold as needed

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main(argv=[''], exit=False)

# Result:



**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:

