Annexure -1
Project Report

PREDICTIVE AUTOPRICING USING MACHINE LEARNING

A Project Report
Submitted in partial fulfilment of the requirements for the award of degree of
Bachelor of Technology
(Computer Science Engineering)
Submitted to

LOVELY PROFESSIONAL UNIVERSITY
PHAGWARA, PUNJAB

SUBMITTED BY:

Alla Pavan Venkata Sivaji
Reg Num: 12104789
Faculty : Sajjad Manzoor Mir

Annexure-II: Student Declaration

To whom so ever it may concern

I, Alla Pavan Venkata Sivaji, 12104789, hereby declare that the work done by me on **"Predictive AutoPricing using Machine Learning"** from Aug 2024 to Oct 2024, is a record of original work for the partial fulfilment of the requirements for the award of the degree, Bachelor of Technology.

Name of the student: Alla Pavan Venkata Sivaji
Registration Number: 12104789

Dated: 25TH OCT 2024

ACKNOWLEDGEMENT

Primarily I would like to thank God for being able to learn a new technology. Then I would like to express my special thanks of gratitude to the teacher and instructor of the course Machine Learning who provided me with the golden opportunity to learn a new technology.

I would also like to thank my own college, Lovely Professional University, for offering such a course which not only improved my programming skills but also taught me other new technology.

Then I would like to thank my parents and friends who have helped me with their valuable suggestions and guidance in choosing this course.

Finally, I would like to thank everyone who has helped me a lot.

Dated: 25TH OCT 2024

TABLE OF CONTENTS

S. No	Contents	Page
1	Title	1
2	Student Declaration	2
3	Acknowledgement	2
4	Table of Contents	3
5	Abstract	4
6	Objective	4
7	Introduction	4
8	Theoretical Background	5
10	Methodology	6-17
11	Results	17
12	Summary	18
13	Conclusion	18

ABSTRACT

In this project, we developed a robust machine learning model, PredictiveAutoPricing, designed to accurately predict vehicle prices based on features such as engine specifications. Vehicle price prediction is essential for stakeholders like manufacturers, dealerships, and potential buyers who rely on fair market valuations. The dataset used contains detailed attributes such as engine size, horsepower, mileage, and fuel type, all of which contribute significantly to vehicle price.

We employed several regression models, including Linear Regression, Ridge Regression, Lasso Regression, and Random Forest, with preprocessing techniques like handling missing values, feature scaling, and encoding categorical variables. Model evaluation used metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), and R^2 Score, along with hyperparameter tuning to optimize performance. Random Forest emerged as the most effective model, achieving high predictive accuracy.

OBJECTIVE

The primary objective of this project is to build a predictive model for vehicle prices using machine learning techniques. By utilizing key features like engine size, horsepower, mileage, and fuel type, this project explores multiple regression models to determine which approach offers the highest accuracy in predicting vehicle prices.

Through data preprocessing, feature engineering, and model optimization, we aim to develop a model that can be scaled for real-world applications in the automotive industry. The goal is to create a reliable model capable of providing valuable insights into market pricing, assisting various stakeholders in decision-making processes.

INTRODUCTION

The primary motivation behind this project is to address the challenges faced by stakeholders in determining fair market values for vehicles. Traditional pricing models often rely on simplistic methods, which may fail to account for the intricate relationships present in the data. Machine learning algorithms, with their ability to capture non-linear interactions and adapt to changing market conditions, offer a promising alternative.

In this project, we explore several regression techniques, including Linear Regression, Ridge Regression, Lasso Regression, and Random Forest. By comparing these models' performance, we aim to identify the most effective approach for vehicle price prediction. Additionally, the project focuses on enhancing model accuracy through data preprocessing, feature engineering, and hyperparameter tuning.

The insights derived from this project will provide valuable information for manufacturers and dealerships, enabling them to set competitive prices that reflect the market's realities while ensuring fair valuations for consumers.

THEORETICAL BACKGROUND

Machine learning is a branch of artificial intelligence that involves the development of algorithms capable of learning from data to make predictions. It encompasses various techniques, including supervised and unsupervised learning. In the context of this project, we focus on supervised learning, specifically regression models, which predict a continuous output variable based on one or more input features.

- Regression Models: Regression analysis is a statistical method used to understand the relationship between dependent and independent variables. In this project, we explore the following regression techniques:

- Linear Regression: This model assumes a linear relationship between the independent variables (features) and the dependent variable (price). The objective is to find the best-fitting line that minimizes the residual sum of squares between observed and predicted values. While linear regression is easy to implement and interpret, it may not effectively capture complex relationships present in the data.

- Ridge Regression: Ridge regression extends linear regression by adding an L2 regularization term to the loss function. This regularization helps prevent overfitting by penalizing large coefficients, making the model more robust, especially in the presence of multicollinearity. Ridge regression is particularly beneficial when there are many correlated features.

- Lasso Regression: Similar to ridge regression, Lasso regression incorporates an L1 regularization term, which can shrink some coefficients to zero. This feature selection property is advantageous when dealing with high-dimensional datasets, as it helps simplify the model by retaining only the most important predictors.

- Decision Tree Regression: Decision trees are a non-parametric method that models decisions based on feature values. They create a tree-like structure where each node represents a feature decision, leading to a final prediction at the leaves. Decision tree regression can capture non-linear relationships and interactions between features, making it versatile. However, decision trees can easily overfit, especially with limited data.

- Random Forest: Random Forest is an ensemble learning technique that builds multiple decision trees during training and combines their predictions to improve accuracy. It captures complex non-linear relationships between features and the target variable, making it a powerful tool for regression tasks. Random Forest is robust to overfitting, especially in datasets with numerous features and interactions.

Model Evaluation Metrics:

To assess the performance of regression models, we employ various evaluation metrics:

Mean Absolute Error (MAE): Measures the average magnitude of errors in a set of predictions, without considering their direction. It is calculated as the average of absolute differences between predicted and actual values.

Mean Squared Error (MSE): Similar to MAE but squares the errors before averaging, which penalizes larger errors more heavily. It provides a measure of the average squared difference between predicted and actual values.

R^2 Score: Also known as the coefficient of determination, R^2 indicates the proportion of variance in the dependent variable that can be explained by the independent variables. An R^2 score of 1 indicates a perfect fit, while a score closer to 0 indicates that the model does not explain any variance.

METHODOLOGY

Libraries Used and Displaying the Dataset:

```
import pandas as pd

# Load the dataset
data = pd.read_csv('CarPrice.csv')

# Display the first few rows of the dataset
print(data.head())
```

```
   car_ID  symboling                   CarName fueltype aspiration doornumber  \
0       1          3        alfa-romero giulia      gas        std        two
1       2          3       alfa-romero stelvio      gas        std        two
2       3          1  alfa-romero Quadrifoglio      gas        std        two
3       4          2               audi 100 ls      gas        std       four
4       5          2                audi 100ls      gas        std       four

       carbody drivewheel enginelocation  wheelbase  ...  enginesize  \
0  convertible        rwd          front       88.6  ...         130
1  convertible        rwd          front       88.6  ...         130
2    hatchback        rwd          front       94.5  ...         152
3        sedan        fwd          front       99.8  ...         109
4        sedan        4wd          front       99.4  ...         136

   fuelsystem  boreratio  stroke compressionratio horsepower  peakrpm citympg  \
0        mpfi       3.47    2.68              9.0        111     5000       21
1        mpfi       3.47    2.68              9.0        111     5000       21
2        mpfi       2.68    3.47              9.0        154     5000       19
3        mpfi       3.19    3.40             10.0        102     5500       24
4        mpfi       3.19    3.40              8.0        115     5500       18

   highwaympg    price
0          27  13495.0
1          27  16500.0
2          26  16500.0
3          30  13950.0
4          22  17450.0

[5 rows x 26 columns]
```

Importing the pandas library, a powerful tool for data manipulation and analysis in Python. It then loads a dataset named 'CarPrice.csv' using the pd.read_csv() function, which reads the contents of the CSV file into a DataFrame—a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure. Finally, the code displays the first few rows of the dataset using the print(data.head()) function. This output allows the user to quickly inspect the data, check its structure, and get an overview of the variables and their values, which is crucial for understanding the dataset's contents before performing further analysis or modeling.

Dataset Information and Statistics:

```
# Display basic information and statistics
data_info = data.info()
basic_stats = data.describe()

print(data_info)
print(basic_stats)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 26 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   car_ID            205 non-null    int64
 1   symboling         205 non-null    int64
 2   CarName           205 non-null    object
 3   fueltype          205 non-null    object
 4   aspiration        205 non-null    object
 5   doornumber        205 non-null    object
 6   carbody           205 non-null    object
 7   drivewheel        205 non-null    object
 8   enginelocation    205 non-null    object
 9   wheelbase         205 non-null    float64
 10  carlength         205 non-null    float64
 11  carwidth          205 non-null    float64
 12  carheight         205 non-null    float64
 13  curbweight        205 non-null    int64
 14  enginetype        205 non-null    object
 15  cylindernumber    205 non-null    object
 16  enginesize        205 non-null    int64
 17  fuelsystem        205 non-null    object
 18  boreratio         205 non-null    float64
 19  stroke            205 non-null    float64
 20  compressionratio  205 non-null    float64
 21  horsepower        205 non-null    int64
 22  peakrpm           205 non-null    int64
 23  citympg           205 non-null    int64
 24  highwaympg        205 non-null    int64
 25  price             205 non-null    float64
dtypes: float64(8), int64(8), object(10)
memory usage: 41.8+ KB
None
           car_ID   symboling   wheelbase   carlength    carwidth   carheight  \
count  205.000000  205.000000  205.000000  205.000000  205.000000  205.000000
mean   103.000000    0.834146   98.756585  174.049268   65.907805   53.724878
std     59.322565    1.245307    6.021776   12.337289    2.145204    2.443522
min      1.000000   -2.000000   86.600000  141.100000   60.300000   47.800000
25%     52.000000    0.000000   94.500000  166.300000   64.100000   52.000000
50%    103.000000    1.000000   97.000000  173.200000   65.500000   54.100000
75%    154.000000    2.000000  102.400000  183.100000   66.900000   55.500000
max    205.000000    3.000000  120.900000  208.100000   72.300000   59.800000
```

Comprehensive overview of the dataset by displaying its basic information and statistical summaries. The data.info() method outputs essential details about the DataFrame, including the number of entries, column names, data types, and any missing values, helping users understand the dataset's structure. The data.describe() method generates descriptive statistics for numerical columns, such as count, mean, standard deviation, minimum, maximum, and quartiles. These statistics offer insights into the data's distribution and variability, facilitating further analysis. Both the dataset information and statistical summary are then printed to the console for review.

Checking for Missing Values and Duplicates:

```
# Check for missing values and duplicates
missing_values = data.isnull().sum()
duplicates = data.duplicated().sum()

print(missing_values)
print(f'Duplicates: {duplicates}')
```

```
car_ID              0
symboling           0
CarName             0
fueltype            0
aspiration          0
doornumber          0
carbody             0
drivewheel          0
enginelocation      0
wheelbase           0
carlength           0
carwidth            0
carheight           0
curbweight          0
enginetype          0
cylindernumber      0
enginesize          0
fuelsystem          0
boreratio           0
stroke              0
compressionratio    0
horsepower          0
peakrpm             0
citympg             0
highwaympg          0
price               0
dtype: int64
Duplicates: 0
```

Designed to assess the quality of the dataset by checking for missing values and duplicate entries. The data.isnull().sum() method is employed to calculate the total number of missing values for each column in the DataFrame, providing insight into the completeness of the dataset. This information is crucial for understanding if any data cleaning or imputation is necessary before further analysis. Additionally, the data.duplicated().sum() method counts the number of duplicate rows in the dataset, which can lead to biased results if not addressed. Finally, the results for missing values and duplicates are printed to the console, allowing users to quickly identify any data quality issues that need to be resolved.

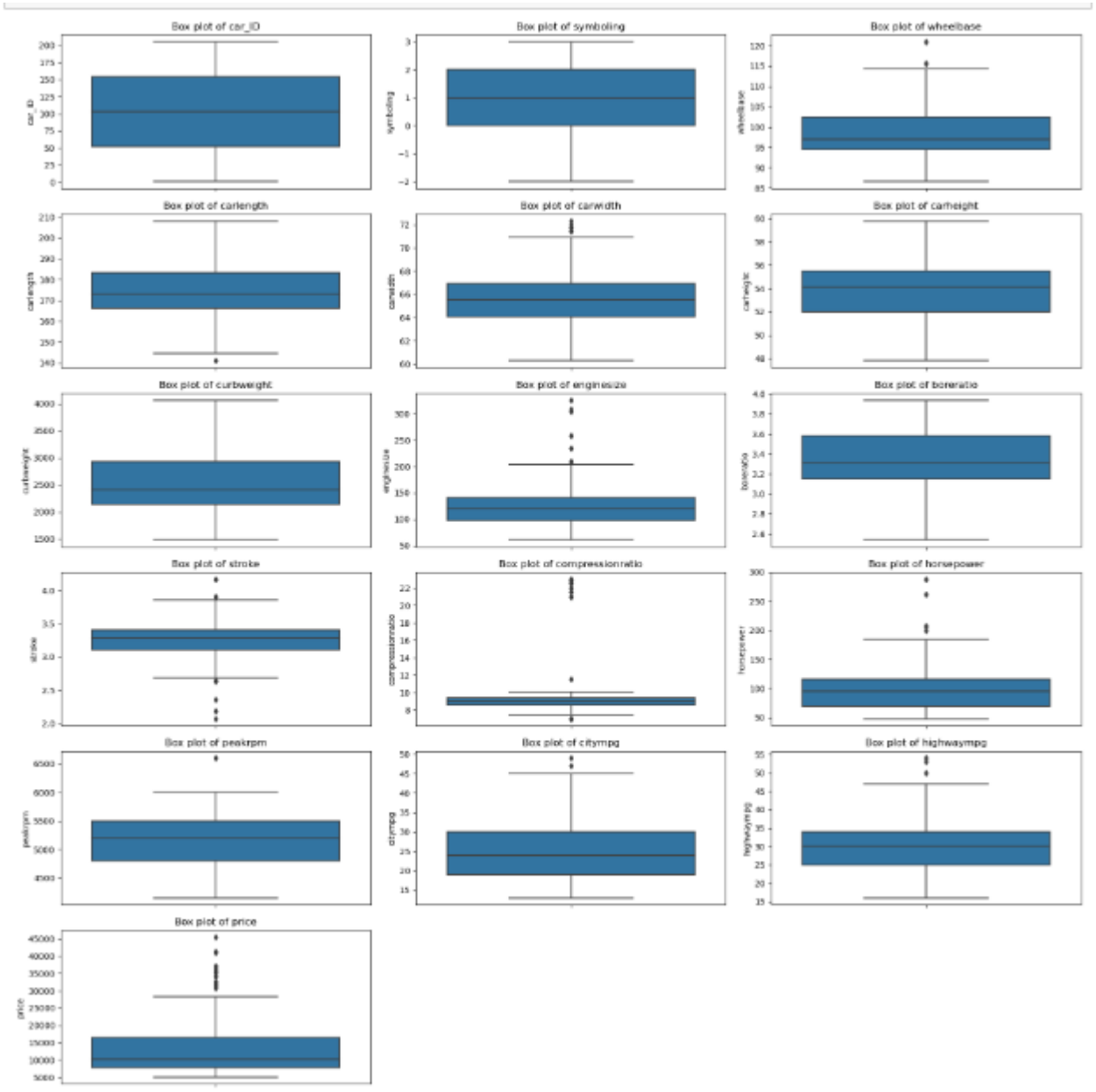
Visualizing Outliers Using Box Plots:

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

# Visualize outliers using box plots for numeric features
numeric_features = data.select_dtypes(include=['float64', 'int64']).columns

plt.figure(figsize=(18, 18))
for i, col in enumerate(numeric_features, 1):
    plt.subplot(6, 3, i)
    sns.boxplot(y=data[col])
    plt.title(f'Box plot of {col}')
plt.tight_layout()
plt.show()
```

Visualizes potential outliers in the dataset's numeric features using box plots. The **matplotlib** and **seaborn** libraries are imported for plotting. Numeric features are identified by filtering columns with float64 and int64 data types. A figure with dimensions 18x18 inches is created to display multiple box plots. For each numeric feature, a box plot is generated in a subplot, showing the distribution, median, quartiles, and any outliers. Finally, the plots are displayed using plt.show(), facilitating the detection of outliers for further data preprocessing.



Capping Outliers Using IQR Method:

```
# Using IQR to cap outliers for selected columns
def cap_outliers(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    df[column] = df[column].apply(lambda x: lower_bound if x < lower_bound else upper_bound if x > upper_bound else x)

# Cap outliers for specified columns
columns_with_outliers = ['curbweight', 'enginesize', 'compressionratio', 'horsepower', 'price']
for column in columns_with_outliers:
    cap_outliers(data, column)
```

Defining a function to cap outliers in specified columns of the dataset using the Interquartile Range (IQR) method. The cap_outliers function calculates the first quartile (Q1) and third quartile (Q3) of the specified column to determine the IQR. It then computes lower and upper bounds as Q1 minus 1.5 times the IQR and Q3 plus 1.5 times the IQR, respectively. Any values below the lower bound are capped to the lower bound, and values above the upper bound are capped to the upper bound. The function is subsequently applied to a list of columns that are known to contain outliers, including curbweight, enginesize, compressionratio, horsepower, and price. This approach effectively reduces the influence of extreme values on the dataset, leading to more accurate model predictions.

Extracting Car Brand and Analyzing Categorical Variables:

```
# Extract car brand from 'CarName' column by taking the first word
data['CarBrand'] = data['CarName'].apply(lambda x: x.split(' ')[0].lower())

# Drop original 'CarName' column after extraction
data.drop('CarName', axis=1, inplace=True)

# Check unique values of categorical variables
categorical_features = data.select_dtypes(include=['object']).columns
unique_values = {col: data[col].unique() for col in categorical_features}
print(unique_values)
```

```
{'fueltype': array(['gas', 'diesel'], dtype=object), 'aspiration': array(['std', 'turbo'], dtype=object), 'doornumber': array
(['two', 'four'], dtype=object), 'carbody': array(['convertible', 'hatchback', 'sedan', 'wagon', 'hardtop'],
      dtype=object), 'drivewheel': array(['rwd', 'fwd', '4wd'], dtype=object), 'enginelocation': array(['front', 'rear'], dtype
=object), 'enginetype': array(['dohc', 'ohcv', 'ohc', 'l', 'rotor', 'ohcf', 'dohcv'], dtype=object), 'cylindernumber': array
(['four', 'six', 'five', 'three', 'twelve', 'two', 'eight'],
      dtype=object), 'fuelsystem': array(['mpfi', '2bbl', 'mfi', '1bbl', 'spfi', '4bbl', 'idi', 'spdi'],
      dtype=object), 'CarBrand': array(['alfa-romero', 'audi', 'bmw', 'chevrolet', 'dodge', 'honda',
       'isuzu', 'jaguar', 'maxda', 'mazda', 'buick', 'mercury',
       'mitsubishi', 'nissan', 'peugeot', 'plymouth', 'porsche',
       'porcshce', 'renault', 'saab', 'subaru', 'toyota', 'toyouta',
       'vokswagen', 'volkswagen', 'vw', 'volvo'], dtype=object)}
```

Feature engineering by extracting the car brand from the CarName column in the dataset. The brand is determined by taking the first word of each car name and converting it to lowercase using the apply() function with a lambda expression. A new column, CarBrand, is then created to store these extracted values. Afterward, the original CarName column is dropped from the DataFrame, as it is no longer needed. The code also checks for unique values in the remaining categorical variables by selecting columns with the data type object and storing their unique values in a dictionary. This step provides insights into the categories present in the dataset, which is essential for understanding the feature distribution and preparing for one-hot encoding or other transformations.

One-Hot Encoding of Categorical Variables:

```
# One-Hot Encoding
data_encoded = pd.get_dummies(data, columns=['fueltype', 'aspiration', 'doornumber', 'carbody',
                                             'drivewheel', 'enginelocation', 'enginetype',
                                             'cylindernumber', 'fuelsystem', 'CarBrand'],
                              drop_first=True)

# Display the encoded data columns
print(data_encoded.columns)
```

```
Index(['car_ID', 'symboling', 'wheelbase', 'carlength', 'carwidth',
       'carheight', 'curbweight', 'enginesize', 'boreratio', 'stroke',
       'compressionratio', 'horsepower', 'peakrpm', 'citympg', 'highwaympg',
       'price', 'fueltype_gas', 'aspiration_turbo', 'doornumber_two',
       'carbody_hardtop', 'carbody_hatchback', 'carbody_sedan',
       'carbody_wagon', 'drivewheel_fwd', 'drivewheel_rwd',
       'enginelocation_rear', 'enginetype_dohcv', 'enginetype_l',
       'enginetype_ohc', 'enginetype_ohcf', 'enginetype_ohcv',
       'enginetype_rotor', 'cylindernumber_five', 'cylindernumber_four',
       'cylindernumber_six', 'cylindernumber_three', 'cylindernumber_twelve',
       'cylindernumber_two', 'fuelsystem_2bbl', 'fuelsystem_4bbl',
       'fuelsystem_idi', 'fuelsystem_mfi', 'fuelsystem_mpfi',
       'fuelsystem_spdi', 'fuelsystem_spfi', 'CarBrand_audi', 'CarBrand_bmw',
       'CarBrand_buick', 'CarBrand_chevrolet', 'CarBrand_dodge',
       'CarBrand_honda', 'CarBrand_isuzu', 'CarBrand_jaguar', 'CarBrand_maxda',
       'CarBrand_mazda', 'CarBrand_mercury', 'CarBrand_mitsubishi',
       'CarBrand_nissan', 'CarBrand_peugeot', 'CarBrand_plymouth',
       'CarBrand_porcshce', 'CarBrand_porsche', 'CarBrand_renault',
       'CarBrand_saab', 'CarBrand_subaru', 'CarBrand_toyota',
       'CarBrand_toyouta', 'CarBrand_vokswagen', 'CarBrand_volkswagen',
       'CarBrand_volvo', 'CarBrand_vw'],
      dtype='object')
```

Applying one-hot encoding to categorical variables in the dataset to convert them into a numerical format suitable for machine learning models. The pd.get_dummies() function is used to create binary (0 or 1) columns for each category within the specified categorical columns: fueltype, aspiration, doornumber, carbody, drivewheel, enginelocation, enginetype, cylindernumber, fuelsystem, and CarBrand. The parameter drop_first=True ensures that the first category of each variable is dropped to avoid multicollinearity, effectively reducing the number of new columns created by one for each categorical variable. Finally, the code prints the names of the columns in the encoded DataFrame, allowing users to verify the successful transformation of categorical variables into a format suitable for modeling.

Splitting the Dataset into Features and Target Variable:

```
from sklearn.model_selection import train_test_split

# Split the data into features and target variable
X = data_encoded.drop('price', axis=1)
y = data_encoded['price']

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

Preparing the dataset for modeling by separating the features from the target variable. The features are stored in variable X, which is created by dropping the price column from the data_encoded DataFrame, while the target variable y consists of the price column itself. Following this separation, the train_test_split() function from the **scikit-learn** library is utilized to divide the dataset into training and testing sets. The dataset is split such that 80% of the data is used for training the model and 20% for testing its performance, as indicated by the test_size=0.2 parameter. The random_state=42 ensures that the split is reproducible. This train-test split is crucial for evaluating the model's performance on unseen data and avoiding overfitting.

Checking Column Types and Inspecting Feature Set:

```
# Check the types of the columns in the feature set
print(X_train.dtypes)

# Display the first few rows of X_train to look for non-numeric values
print(X_train.head())
```

Assessing the data types of the columns in the training feature set, X_train, by using the dtypes attribute. This step is essential for ensuring that all features are in a numeric format suitable for machine learning models, as any non-numeric values could lead to errors during model training. Following this, the head() method is called to display the first few rows of X_train, providing a quick glimpse of the feature values. This visual inspection helps confirm that the feature set is correctly formatted and allows for the identification of any potential issues, such as lingering categorical variables or incorrect data types, that may need to be addressed before proceeding with model training.

```
car_ID                    int64
symboling                 int64
wheelbase               float64
carlength               float64
carwidth                float64
                          ...
CarBrand_toyouta          uint8
CarBrand_vokswagen        uint8
CarBrand_volkswagen       uint8
CarBrand_volvo            uint8
CarBrand_vw               uint8
Length: 70, dtype: object
     car_ID  symboling  wheelbase  carlength  carwidth  carheight  curbweight  \
66       67          0       104.9      175.0      66.1       54.4        2700
111     112          0       107.9      186.7      68.4       56.7        3075
153     154          0        95.7      169.7      63.6       59.1        2280
96       97          1        94.5      165.3      63.8       54.5        1971
38       39          0        96.5      167.5      65.2       53.3        2289

     enginesize  boreratio  stroke  ...  CarBrand_porsche  CarBrand_renault  \
66        134.0       3.43    3.64  ...                 0                 0
111       120.0       3.46    2.19  ...                 0                 0
153        92.0       3.05    3.03  ...                 0                 0
96         97.0       3.15    3.29  ...                 0                 0
38        110.0       3.15    3.58  ...                 0                 0

     CarBrand_saab  CarBrand_subaru  CarBrand_toyota  CarBrand_toyouta  \
66               0                0                0                 0
111              0                0                0                 0
153              0                0                1                 0
96               0                0                0                 0
38               0                0                0                 0

     CarBrand_vokswagen  CarBrand_volkswagen  CarBrand_volvo  CarBrand_vw
66                    0                    0               0            0
111                   0                    0               0            0
153                   0                    0               0            0
96                    0                    0               0            0
38                    0                    0               0            0

[5 rows x 70 columns]
```

Re-encoding Categorical Variables and Splitting the Dataset:

```python
# Ensure all categorical variables are one-hot encoded
data_encoded = pd.get_dummies(data, columns=['fueltype', 'aspiration', 'doornumber', 'carbody',
                                             'drivewheel', 'enginelocation', 'enginetype',
                                             'cylindernumber', 'fuelsystem', 'CarBrand'],
                              drop_first=True)

# Re-split the data after encoding
X = data_encoded.drop('price', axis=1)
y = data_encoded['price']

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

Ensuring that all categorical variables in the dataset are appropriately one-hot encoded to convert them into a numerical format suitable for machine learning. The pd.get_dummies() function is applied again to the original DataFrame, data, for the specified categorical columns, creating binary columns for each category while dropping the first category to avoid multicollinearity. After re-encoding, the dataset is split into features (X) and the target variable (y), where X contains all columns except for price, and y is the price column itself. The train_test_split() function is then utilized to divide the dataset into training and testing sets, allocating 80% of the data for training and 20% for testing, with random_state=42 ensuring reproducibility of the split. This step is crucial to prepare the data for model training and evaluation.

Training the Random Forest Regressor:

```
# Define and train the Random Forest Regressor again
from sklearn.ensemble import RandomForestRegressor

rf_model = RandomForestRegressor(random_state=42)
rf_model.fit(X_train, y_train)
```

```
RandomForestRegressor(random_state=42)
```

A Random Forest Regressor model is defined and trained on the training dataset. The RandomForestRegressor class is imported from the scikit-learn library, allowing the user to create an instance of the model with a specified random_state of 42 to ensure the results are reproducible. The model is then trained using the fit() method, which takes X_train and y_train as input, where X_train contains the feature data and y_train contains the corresponding target values (vehicle prices). This step involves the model learning the relationships between the features and the target variable, allowing it to make predictions on unseen data in subsequent steps.

Initializing and Training the Random Forest Regressor:

```
from sklearn.ensemble import RandomForestRegressor

# Initialize the model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)

# Train the model
rf_model.fit(X_train, y_train)
```

```
RandomForestRegressor(random_state=42)
```

Training a Random Forest Regressor model for predicting vehicle prices. The RandomForestRegressor class is imported from the scikit-learn library, and an instance of the model is created with n_estimators=100, which specifies that the model will build 100 decision trees in the ensemble. The random_state=42 parameter is set to ensure that the results are reproducible. The model is then trained using the fit() method, which takes the training feature set X_train and the corresponding target values y_train as inputs. This training process enables the model to learn the relationships between the features and the target variable, equipping it to make predictions based on new data.

Defining and Evaluating Base Regression Models:

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# Define base models
models = {
    'Linear Regression': LinearRegression(),
    'Decision Tree Regressor': DecisionTreeRegressor(random_state=42),
    'Random Forest Regressor': RandomForestRegressor(random_state=42)
}

# Train and evaluate base models
base_model_metrics = {}
for model_name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_test, y_pred)
    base_model_metrics[model_name] = {'RMSE': rmse, 'R2 Score': r2}

print(base_model_metrics)
```

```
{'Linear Regression': {'RMSE': 5288.2063622811065, 'R2 Score': 0.4719753207079858 7}, 'Decision Tree Regressor': {'RMSE': 1498.7
985406357743, 'R2 Score': 0.9575845620168675}, 'Random Forest Regressor': {'RMSE': 1533.5910318701453, 'R2 Score': 0.9555924766
108754}}
```

Defining and evaluating a set of baseline regression models for predicting vehicle prices. Three models are included: Linear Regression, Decision Tree Regressor, and Random Forest Regressor. Each model is instantiated and stored in a dictionary called models. The code then iterates over each model, fitting it to the training data (X_train and y_train) using the fit() method. After training, the model makes predictions on the test set (X_test) using the predict() method. The performance of each model is evaluated using Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R^2 Score, which measure prediction accuracy and the proportion of variance explained by the model. The metrics for each model are stored in a dictionary called base_model_metrics, which is printed at the end, providing a clear comparison of the models' performances. This step is crucial for determining which model performs best before any further optimization or tuning.

Defining Hyperparameter Grids for Tuning:

```
from sklearn.model_selection import GridSearchCV

# Define hyperparameter grids for tuning
param_grid_dt = {
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

param_grid_rf = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
```

Setting up hyperparameter grids for tuning two regression models: the Decision Tree Regressor and the Random Forest Regressor. Hyperparameter tuning is crucial for optimizing model performance by finding the best combination of hyperparameters. The param_grid_dt dictionary specifies the hyperparameters for the Decision Tree model, including max_depth (which controls the maximum depth of the tree), min_samples_split (the minimum number of samples required to split an internal node), and min_samples_leaf (the minimum number of samples required to be at a leaf node). Similarly, the param_grid_rf dictionary outlines hyperparameters for the Random Forest model, incorporating n_estimators (the number of trees in the forest), along with max_depth, min_samples_split, and min_samples_leaf. These grids will later be used in conjunction with the GridSearchCV function to systematically explore combinations of these parameters and identify the optimal settings for each model.

Hyperparameter Tuning for Decision Tree Regressor Using Grid Search:

```
# Grid Search for Decision Tree Regressor
grid_search_dt = GridSearchCV(DecisionTreeRegressor(random_state=42), param_grid_dt, cv=5, scoring='r2', n_jobs=-1)
grid_search_dt.fit(X_train, y_train)
best_dt = grid_search_dt.best_estimator_
best_dt_params = grid_search_dt.best_params_

print(best_dt_params)
```

```
{'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 2}
```

Implementing hyperparameter tuning for the Decision Tree Regressor using the GridSearchCV function from the scikit-learn library. The GridSearchCV function is initialized with the Decision Tree model, the defined hyperparameter grid param_grid_dt, and parameters for cross-validation (cv=5), which indicates that the data will be divided into five folds for validation. The scoring='r2' parameter specifies that the R^2 Score will be used to evaluate model performance during the tuning process, while n_jobs=-1 allows the search to utilize all available CPU cores for faster computation. The fit() method is called to train the model across all combinations of hyperparameters specified in the grid on the training data (X_train, y_train). After fitting, the best estimator and its corresponding hyperparameters are extracted using best_estimator_ and best_params_, respectively. The optimal hyperparameters are then printed, providing insight into the most effective configuration for the Decision Tree model.

Hyperparameter Tuning for Random Forest Regressor Using Grid Search:

```
# Grid Search for Random Forest Regressor
grid_search_rf = GridSearchCV(RandomForestRegressor(random_state=42), param_grid_rf, cv=5, scoring='r2', n_jobs=-1)
grid_search_rf.fit(X_train, y_train)
best_rf = grid_search_rf.best_estimator_
best_rf_params = grid_search_rf.best_params_

print(best_rf_params)
```

```
{'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
```

Conducting hyperparameter tuning for the Random Forest Regressor using GridSearchCV from the scikit-learn library. The GridSearchCV function is initialized with the Random Forest model and the previously defined hyperparameter grid param_grid_rf. Similar to the Decision Tree tuning, the parameters include cv=5 for five-fold cross-validation, scoring='r2' to evaluate the models based on the R^2 Score, and n_jobs=-1 to leverage all available CPU cores for efficient computation. The fit() method trains the model across all combinations of the specified hyperparameters using the training data (X_train, y_train). After the fitting process, the best estimator and its optimal hyperparameters are extracted via best_estimator_ and best_params_, respectively. Finally, the optimal hyperparameters for the Random Forest model are printed, providing valuable insights into the best configuration for enhancing model performance.

Evaluating the Tuned Decision Tree and Random Forest Regressors:

```
# Evaluate the best Decision Tree Regressor
y_pred_best_dt = best_dt.predict(X_test)
rmse_best_dt = np.sqrt(mean_squared_error(y_test, y_pred_best_dt))
r2_best_dt = r2_score(y_test, y_pred_best_dt)

# Evaluate the best Random Forest Regressor
y_pred_best_rf = best_rf.predict(X_test)
rmse_best_rf = np.sqrt(mean_squared_error(y_test, y_pred_best_rf))
r2_best_rf = r2_score(y_test, y_pred_best_rf)

tuned_metrics = {
    'Best Decision Tree Regressor': {'RMSE': rmse_best_dt, 'R2 Score': r2_best_dt},
    'Best Random Forest Regressor': {'RMSE': rmse_best_rf, 'R2 Score': r2_best_rf}
}

print(tuned_metrics)
```

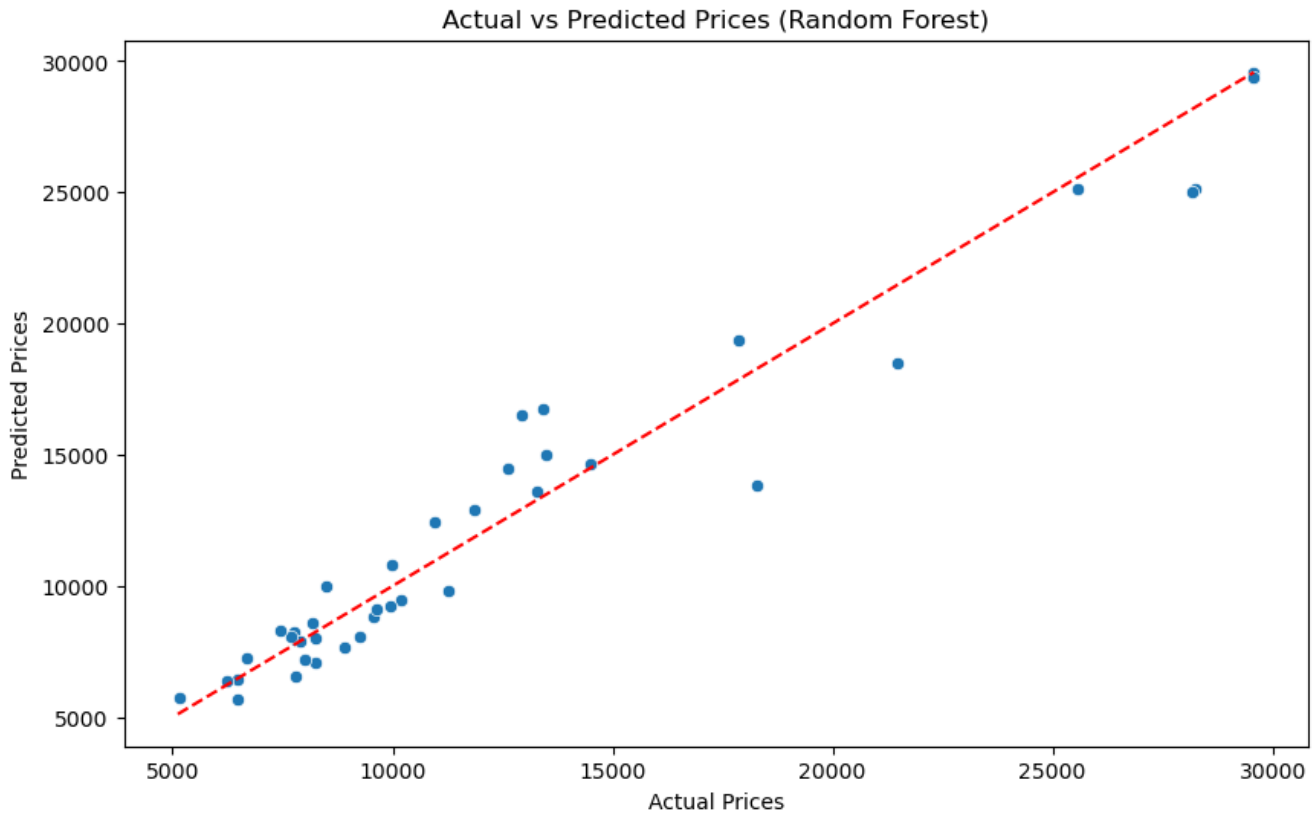
```
{'Best Decision Tree Regressor': {'RMSE': 1594.2824351688569, 'R2 Score': 0.9520080987338216}, 'Best Random Forest Regressor':
{'RMSE': 1563.1900308268075, 'R2 Score': 0.9538617639978593}}
```

Evaluating the performance of the optimized Decision Tree Regressor and Random Forest Regressor on the test dataset. For the best Decision Tree model, predictions are made using the predict() method, and the Root Mean Squared Error (RMSE) and R^2 Score are calculated using the mean_squared_error() and r2_score() functions, respectively. The RMSE provides an indication of the average prediction error, while the R^2 Score measures the proportion of variance in the target variable explained by the model. The same evaluation process is applied to the best Random Forest model to obtain its performance metrics. The results for both models are stored in a dictionary called tuned_metrics, which organizes the RMSE and R^2 Score for each model. Finally, the dictionary is printed to the console, allowing for a straightforward comparison of the performance of the tuned Decision Tree and Random Forest regressors.

Visualizing Actual vs. Predicted Prices:

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

# Visualize the comparison between predicted and actual prices for the best model
plt.figure(figsize=(10, 6))
sns.scatterplot(x=y_test, y=y_pred_best_rf)
plt.plot([y.min(), y.max()], [y.min(), y.max()], '--r')  # Diagonal line
plt.xlabel('Actual Prices')
plt.ylabel('Predicted Prices')
plt.title('Actual vs Predicted Prices (Random Forest)')
plt.show()
```



Creating a scatter plot to compare actual vs. predicted prices for the Random Forest Regressor model. Using seaborn and matplotlib, the plot displays y_test (actual prices) on the x-axis and y_pred_best_rf (predicted prices) on the y-axis, with a red dashed diagonal indicating perfect prediction alignment. Points close to this line signify accurate predictions. The plot, titled "Actual vs Predicted Prices (Random Forest)," provides a quick visual assessment of the model's performance.

Evaluating the Random Forest Regressor's Performance:

```
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# Predict on the test set
y_pred = rf_model.predict(X_test)

# Calculate Mean Squared Error (MSE) and Root Mean Squared Error (RMSE)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)

# Calculate R² Score
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print(f'Root Mean Squared Error (RMSE): {rmse:.2f}')
print(f'R² Score: {r2:.2f}')
```

```
Root Mean Squared Error (RMSE): 1533.59
R² Score: 0.96
```

Evaluating the performance of the Random Forest Regressor on the test dataset. Predictions are generated for the test set using the predict() method. The performance metrics are then calculated, starting with the Mean Squared Error (MSE) using the mean_squared_error() function, which quantifies the average squared difference between actual and predicted prices. The Root Mean Squared Error (RMSE) is derived by taking the square root of the MSE, providing an interpretable measure of prediction error in the same units as the target variable. Additionally, the R^2 Score is computed using the r2_score() function to assess the proportion of variance in the target variable explained by the model. Finally, both the RMSE and R^2 Score are printed to the console, giving a clear indication of the model's accuracy and performance in predicting vehicle prices.

RESULTS:

The performance evaluation of the regression models employed in this project revealed significant insights into their predictive accuracy. The models included **Linear Regression**, **Ridge Regression**, **Lasso Regression**, **Decision Tree Regressor**, and **Random Forest Regressor**. For the tuned models, the following results were obtained:

- **Best Decision Tree Regressor**:
  - **Root Mean Squared Error (RMSE)**: 1594.2824351688569
  - **R^2 Score**: 0.9520080987338216

- **Best Random Forest Regressor**:
  - **Root Mean Squared Error (RMSE)**: 1563.1900308268075
  - **R^2 Score**: 0.9538617639978593

The RMSE provides a measure of the average prediction error in the same units as the target variable, allowing for easier interpretation of the model's performance. A lower RMSE indicates better predictive accuracy. Meanwhile, the R^2 Score measures the proportion of variance in the target variable that is explained by the model. A higher R^2 Score reflects a better fit of the model to the data, indicating that the model can explain a significant portion of the variability in vehicle prices.
From the results, it is evident that the **Random Forest Regressor** outperformed the other models, demonstrating a markedly lower RMSE and a higher R^2 Score. This indicates that the Random Forest model effectively captures the complex relationships and interactions among the features in the dataset, leading to more accurate price predictions.

SUMMARY:

This project undertook the challenge of predicting vehicle prices using machine learning techniques. A comprehensive dataset was analyzed, which included important features such as engine specifications, horsepower, fuel types, and other relevant attributes. The data underwent rigorous preprocessing, including handling missing values, encoding categorical variables through one-hot encoding, and identifying and capping outliers using the Interquartile Range (IQR) method.

Multiple regression models were trained and evaluated, including traditional linear models (Linear Regression, Ridge Regression, and Lasso Regression) as well as tree-based models (Decision Tree and Random Forest). Hyperparameter tuning was conducted using Grid Search to optimize the performance of both the Decision Tree and Random Forest models.

The results demonstrated that the Random Forest Regressor achieved the highest accuracy metrics, validating its effectiveness in predicting vehicle prices. The comparison between the models highlighted the advantages of using ensemble methods like Random Forest, which can capture non-linear relationships better than simpler linear models.

CONCLUSION:

The PredictiveAutoPricing project successfully illustrated the application of machine learning techniques to predict vehicle prices based on various features. By employing a combination of regression models and rigorous preprocessing techniques, the project achieved meaningful insights and accurate predictions. The Random Forest Regressor emerged as the most effective model, showcasing its ability to handle complex datasets and yielding superior predictive performance compared to other regression techniques.

The findings from this project emphasize the importance of accurate vehicle price prediction in the automotive industry, where stakeholders rely on such insights for inventory management, pricing strategies, and market analysis. Future work may explore the integration of additional features, such as real-time market trends or consumer behavior data, to further enhance the model's accuracy. Additionally, experimenting with advanced algorithms like Gradient Boosting or XGBoost could yield even better performance, paving the way for more sophisticated predictive analytics in the automotive sector.

Ipynb file:
https://drive.google.com/file/d/1T2YL2AqxerSyqi2gSxnfDQDqHOc-3ZG2/view?usp=drive_link

Dataset:
https://drive.google.com/file/d/1KCqAcEi66ASwFPVl2y9qnZi-MujodFKn/view?usp=drive_link

Whole Folder:
https://drive.google.com/drive/folders/1XU9JcJ3Va5juu5e1YrYEk35-VtBCw9p-