BIRZEIT UNIVERSITY

Electrical and Computer Engineering Department
Machine Learning and Data Science - ENCS5341
Assignment #2

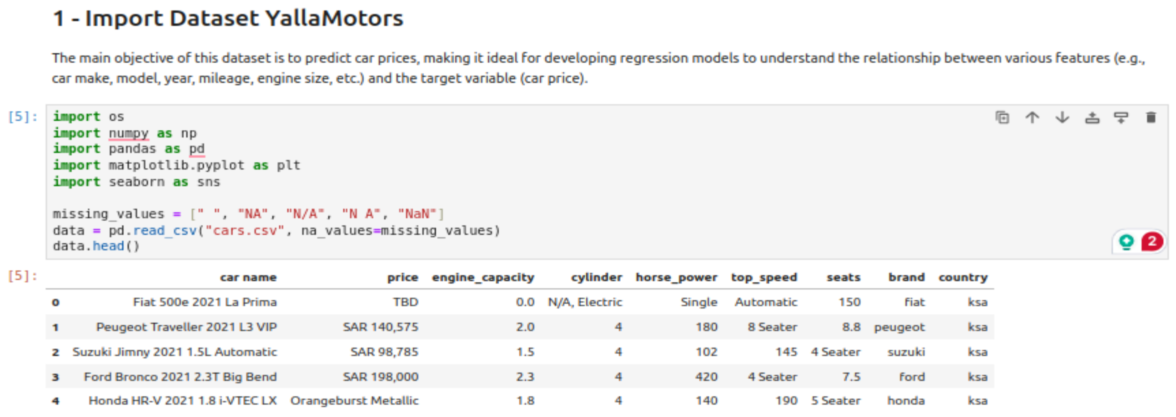
---

Prepared By: Sondos Shahin 1200166, Tala Abahra 1201002
Instructor: Dr. Yazan Abu Farha  Section: 1
Date: Nov 25, 2024
Topic: Machine Learning Assignment: 2
https://github.com/sondosshahin/Machine-Learning-Project-Regression-Analysis-and-Model-Selection-

---

## Importing The Dataset

To begin, we needed to import the necessary libraries for data cleaning: **Pandas** for importing and managing datasets, and **NumPy** for performing mathematical operations. Then, we loaded the dataset using `pd.read_csv`. For this Project, we will be using the **Cars Dataset** from **Kaggle**, available at https://www.kaggle.com/datasets/ahmedwaelnasef/cars-dataset



### 1 - Import Dataset YallaMotors

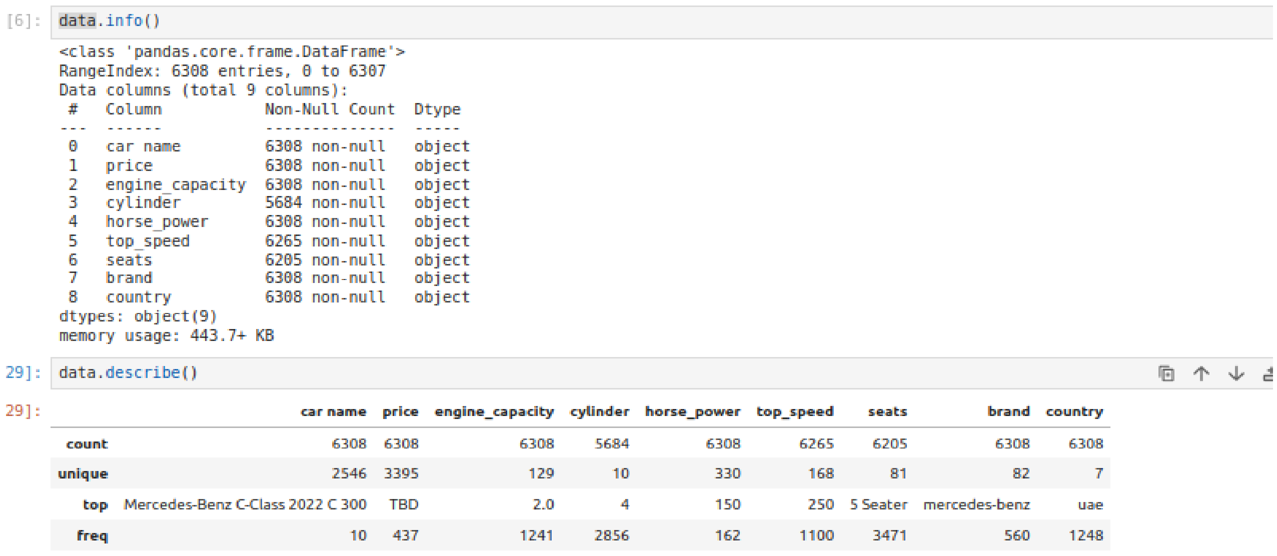
The main objective of this dataset is to predict car prices, making it ideal for developing regression models to understand the relationship between various features (e.g., car make, model, year, mileage, engine size, etc.) and the target variable (car price).

```
[5]: import os
     import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     import seaborn as sns

     missing_values = [" ", "NA", "N/A", "N A", "NaN"]
     data = pd.read_csv("cars.csv", na_values=missing_values)
     data.head()
```

[5]:

| | car name | price | engine_capacity | cylinder | horse_power | top_speed | seats | brand | country |
|---|---|---|---|---|---|---|---|---|---|
| 0 | Fiat 500e 2021 La Prima | TBD | 0.0 | N/A, Electric | Single | Automatic | 150 | fiat | ksa |
| 1 | Peugeot Traveller 2021 L3 VIP | SAR 140,575 | 2.0 | 4 | 180 | 8 Seater | 8.8 | peugeot | ksa |
| 2 | Suzuki Jimny 2021 1.5L Automatic | SAR 98,785 | 1.5 | 4 | 102 | 145 | 4 Seater | suzuki | ksa |
| 3 | Ford Bronco 2021 2.3T Big Bend | SAR 198,000 | 2.3 | 4 | 420 | 4 Seater | 7.5 | ford | ksa |
| 4 | Honda HR-V 2021 1.8 i-VTEC LX | Orangeburst Metallic | 1.8 | 4 | 140 | 190 | 5 Seater | honda | ksa |

This dataset, scraped from the YallaMotors website using Python and the Requests-HTML library, contains 6,308 entries and 9 columns, making it ideal for Exploratory Data Analysis (EDA) and Machine Learning tasks like Linear Regression. The columns include car name, price, engine capacity, cylinder power, horse power, top speed, seats, brand, and country. Most columns have non-null values, though "cylinder," "top speed," and "seats" contain some missing data. The dataset is primarily of object data type, requiring conversion for numerical analysis. The most frequent car name is "Mercedes-Benz C-Class 2022 C 300," and the UAE is the most common country.



```
[6]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6308 entries, 0 to 6307
Data columns (total 9 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   car name         6308 non-null   object
 1   price            6308 non-null   object
 2   engine_capacity  6308 non-null   object
 3   cylinder         5684 non-null   object
 4   horse_power      6308 non-null   object
 5   top_speed        6265 non-null   object
 6   seats            6205 non-null   object
 7   brand            6308 non-null   object
 8   country          6308 non-null   object
dtypes: object(9)
memory usage: 443.7+ KB
```

```
29]: data.describe()
```

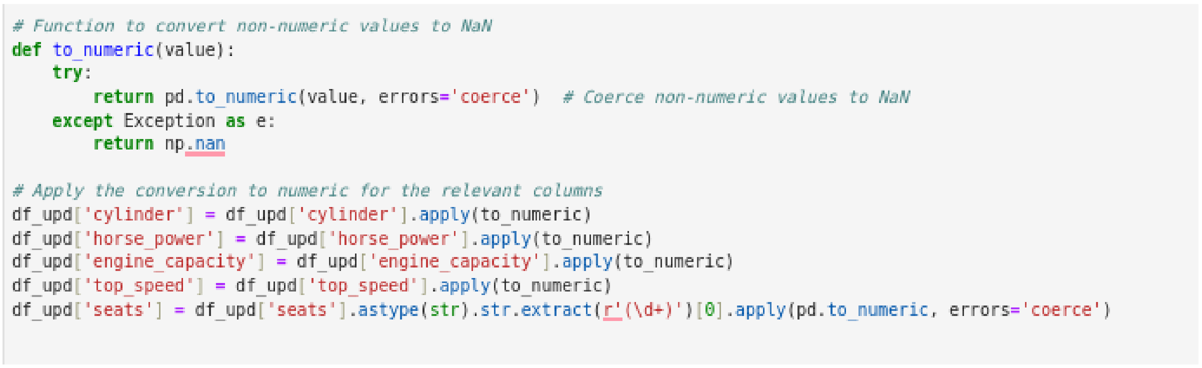
29]:

| | car name | price | engine_capacity | cylinder | horse_power | top_speed | seats | brand | country |
|---|---|---|---|---|---|---|---|---|---|
| **count** | 6308 | 6308 | 6308 | 5684 | 6308 | 6265 | 6205 | 6308 | 6308 |
| **unique** | 2546 | 3395 | 129 | 10 | 330 | 168 | 81 | 82 | 7 |
| **top** | Mercedes-Benz C-Class 2022 C 300 | TBD | 2.0 | 4 | 150 | 250 | 5 Seater | mercedes-benz | uae |
| **freq** | 10 | 437 | 1241 | 2856 | 162 | 1100 | 3471 | 560 | 1248 |

## Data Pre-Processing :

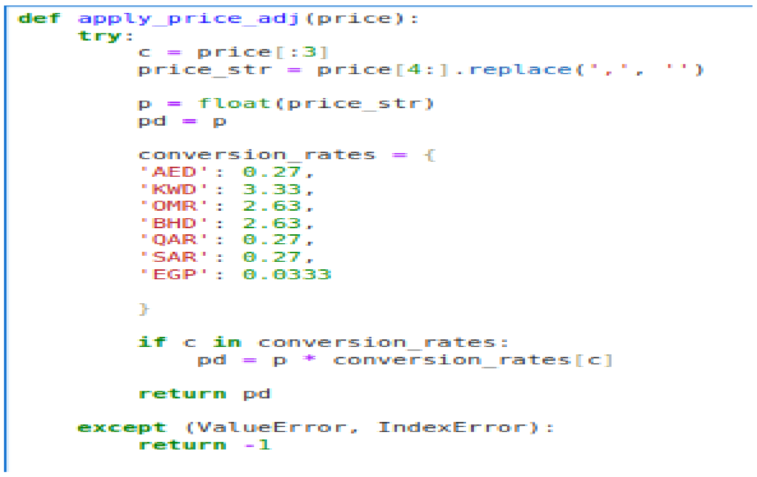
This step is crucial because it ensures the dataset is clean and consistent, which is essential for accurate analysis and modeling. By addressing issues such as missing values, inconsistent formats, and varying scales, we prepare the data for better performance in machine learning models. Standardizing the data allows models to interpret features correctly and avoids biases caused by disparities in the data, leading to more accurate predictions and insights.

Through the available tables, we noticed that some numeric values contained words. Therefore, we converted these values in the tables to missing values. As for the seats, we kept only the numerical values present. This process helps in cleaning the data.



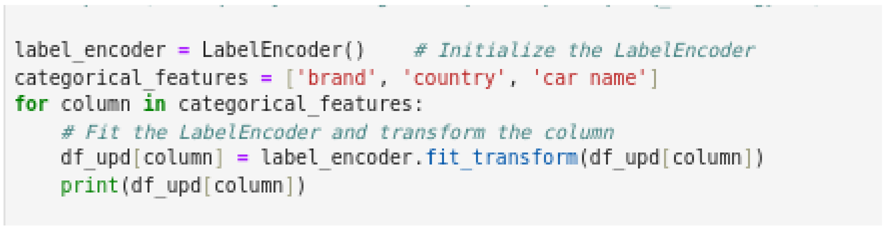
```python
# Function to convert non-numeric values to NaN
def to_numeric(value):
    try:
        return pd.to_numeric(value, errors='coerce')  # Coerce non-numeric values to NaN
    except Exception as e:
        return np.nan

# Apply the conversion to numeric for the relevant columns
df_upd['cylinder'] = df_upd['cylinder'].apply(to_numeric)
df_upd['horse_power'] = df_upd['horse_power'].apply(to_numeric)
df_upd['engine_capacity'] = df_upd['engine_capacity'].apply(to_numeric)
df_upd['top_speed'] = df_upd['top_speed'].apply(to_numeric)
df_upd['seats'] = df_upd['seats'].astype(str).str.extract(r'(\d+)')[0].apply(pd.to_numeric, errors='coerce')
```

Through the available tables, we noticed that some car-related numeric values contained words or were listed in different currencies. For example, car prices were sometimes given as "TBD" or in various currencies. To standardize the data, we converted all car prices to USD. Additionally, for the number of seats, we retained only the numeric values. This cleaning process ensures that the car data is consistent and ready for analysis, enabling more accurate insights and predictions.



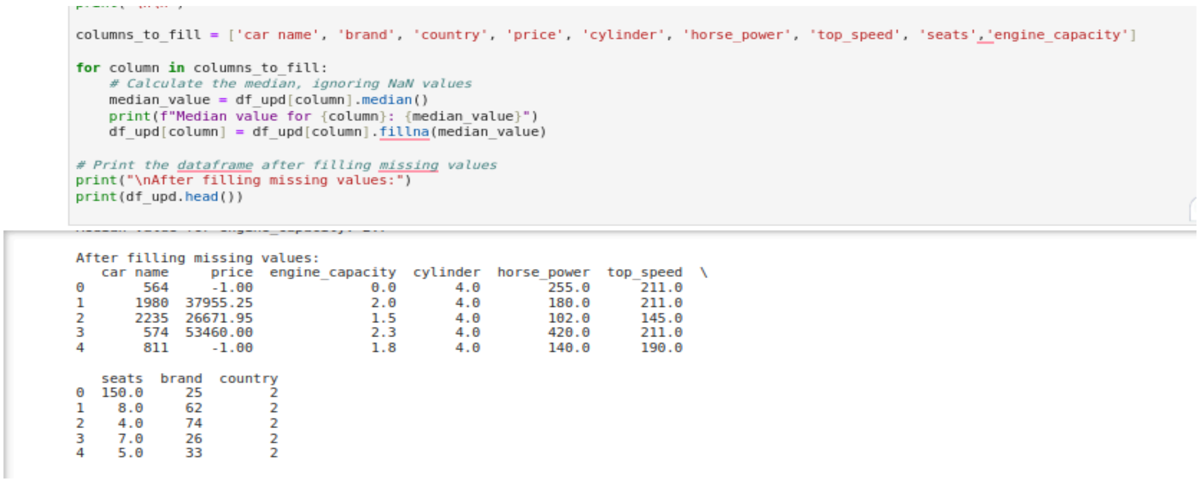
```python
def apply_price_adj(price):
    try:
        c = price[:3]
        price_str = price[4:].replace(',', '')

        p = float(price_str)
        pd = p

        conversion_rates = {
        'AED': 0.27,
        'KWD': 3.33,
        'OMR': 2.63,
        'BHD': 2.63,
        'QAR': 0.27,
        'SAR': 0.27,
        'EGP': 0.0333

        }

        if c in conversion_rates:
            pd = p * conversion_rates[c]

        return pd

    except (ValueError, IndexError):
        return -1
```

We used the LabelEncoder from sklearn.preprocessing to transform categorical variables into numerical values. The LabelEncoder is applied to columns like 'brand', 'country', and 'car name' in the dataset. For each of these columns, the LabelEncoder assigns a unique integer to each distinct category. After encoding, the original categorical values are replaced with integers that represent them. The transformed columns are then added back to the dataset. To confirm the transformation, we print the unique encoded values for each column, which allows us to verify that the encoding process has been applied correctly.



```python
label_encoder = LabelEncoder()     # Initialize the LabelEncoder
categorical_features = ['brand', 'country', 'car name']
for column in categorical_features:
    # Fit the LabelEncoder and transform the column
    df_upd[column] = label_encoder.fit_transform(df_upd[column])
    print(df_upd[column])
```

After calculating the median for each column, we replace the missing values (NaN) in each specified column with the calculated median value. This process ensures that the dataset becomes complete by filling in missing data, particularly for numerical columns. By using the median, we prevent potential distortion that might arise from using the mean, especially if the data contains outliers. The final step is printing the first few rows of the dataset to confirm that the missing values have been successfully filled with the median for each respective column.



```python
columns_to_fill = ['car name', 'brand', 'country', 'price', 'cylinder', 'horse_power', 'top_speed', 'seats','engine_capacity']

for column in columns_to_fill:
    # Calculate the median, ignoring NaN values
    median_value = df_upd[column].median()
    print(f"Median value for {column}: {median_value}")
    df_upd[column] = df_upd[column].fillna(median_value)

# Print the dataframe after filling missing values
print("\nAfter filling missing values:")
print(df_upd.head())
```

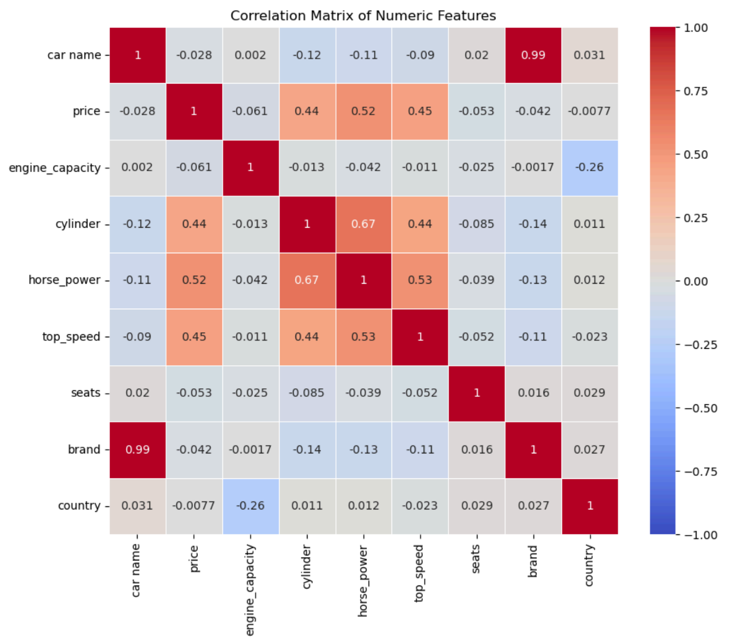
```
After filling missing values:
   car name     price  engine_capacity  cylinder  horse_power  top_speed  \
0       564     -1.00              0.0       4.0        255.0      211.0
1      1980  37955.25              2.0       4.0        180.0      211.0
2      2235  26671.95              1.5       4.0        102.0      145.0
3       574  53460.00              2.3       4.0        420.0      211.0
4       811     -1.00              1.8       4.0        140.0      190.0

   seats  brand  country
0  150.0     25        2
1    8.0     62        2
2    4.0     74        2
3    7.0     26        2
4    5.0     33        2
```

**Correlation:**

After processing and encoding the categorical features, now all the features in our dataset are numerical. We found the correlation matrix to have an initial idea about which features affect our target most (is correlated with the target), and which features have no effect on the target value.

The correlation between the features and the target would give us a brief understanding and would help us expect the results of the regression models that we are going to build. From the correlation matrix we can decide if the models we built are working as expected or if they have problems in their implementations.

Correlation is a statistical analysis that is used to measure and describe the strength and direction of the relationship between two numerical features. Strength indicates how closely two variables are related to each other, and direction indicates how one variable would change its value as the value of the other variable changes. If the correlation is positive, it means that both features are changing in the same direction (both increasing or both decreasing), whereas if it is negative, it means that features are changing in different directions (if one is increasing the other is decreasing).



| | car name | price | engine_capacity | cylinder | horse_power | top_speed | seats | brand | country |
|---|---|---|---|---|---|---|---|---|---|
| car name | 1 | -0.028 | 0.002 | -0.12 | -0.11 | -0.09 | 0.02 | 0.99 | 0.031 |
| price | -0.028 | 1 | -0.061 | 0.44 | 0.52 | 0.45 | -0.053 | -0.042 | -0.0077 |
| engine_capacity | 0.002 | -0.061 | 1 | -0.013 | -0.042 | -0.011 | -0.025 | -0.0017 | -0.26 |
| cylinder | -0.12 | 0.44 | -0.013 | 1 | 0.67 | 0.44 | -0.085 | -0.14 | 0.011 |
| horse_power | -0.11 | 0.52 | -0.042 | 0.67 | 1 | 0.53 | -0.039 | -0.13 | 0.012 |
| top_speed | -0.09 | 0.45 | -0.011 | 0.44 | 0.53 | 1 | -0.052 | -0.11 | -0.023 |
| seats | 0.02 | -0.053 | -0.025 | -0.085 | -0.039 | -0.052 | 1 | 0.016 | 0.029 |
| brand | 0.99 | -0.042 | -0.0017 | -0.14 | -0.13 | -0.11 | 0.016 | 1 | 0.027 |
| country | 0.031 | -0.0077 | -0.26 | 0.011 | 0.012 | -0.023 | 0.029 | 0.027 | 1 |

The correlation matrix indicates that the price (our target) has some correlation with the features: cylinder, horse_power and top_speed. While having weak to no correlation with other features such as seats, car_name, engine_capacity, brand and country.

## Splitting The Dataset :

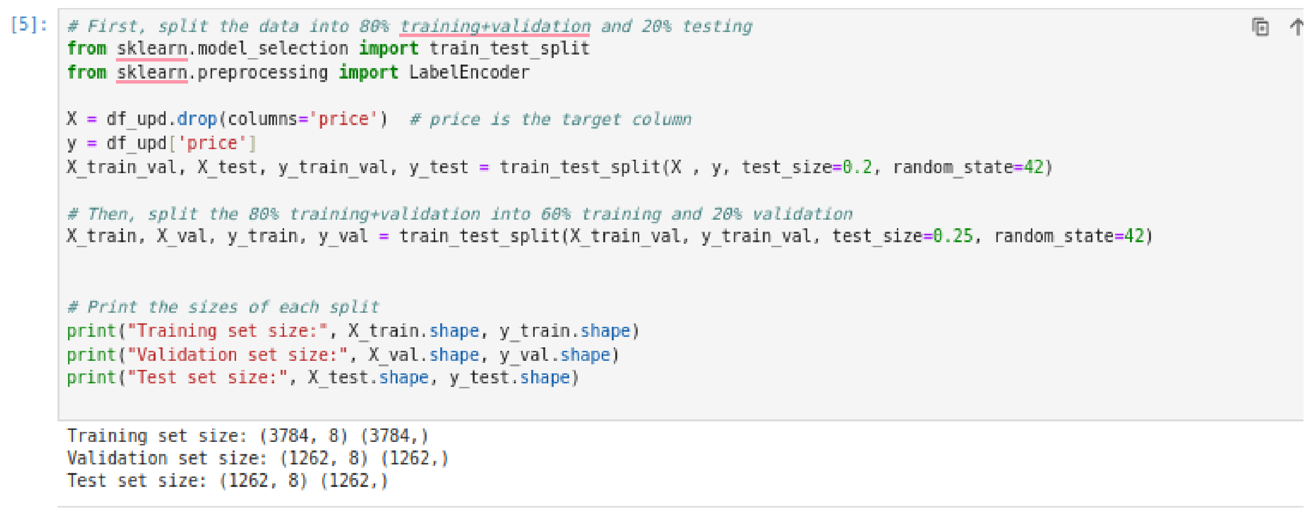
Once the data is cleaned and preprocessed, the next step is to split the dataset into training, validation, and test sets. A typical approach is to allocate 60% of the data for training, 20% for validation, and 20% for testing. This split ensures that the model has enough data to learn from, while also having separate datasets for model evaluation and testing.

1. **Training Set (60%)**: This subset is used to train the machine learning model. It contains the bulk of the data and is used by the model to learn the patterns and relationships between the features and the target variable.
2. **Validation Set (20%)**: This subset is used during model training to fine-tune the model's hyperparameters. It helps in evaluating different configurations of the model to ensure optimal

performance. The validation set provides feedback on the model's generalization ability, without affecting the final evaluation.

3. **Test Set (20%)**: The test set is used after the model has been trained and fine-tuned. It is used to evaluate the final model's performance and assess how well the model generalizes to new, unseen data. The test set provides an unbiased estimate of the model's accuracy and effectiveness.

Split the dataset into training, validation, and test sets. A common split would be 60% for training, 20% for validation, and 20% for testing.

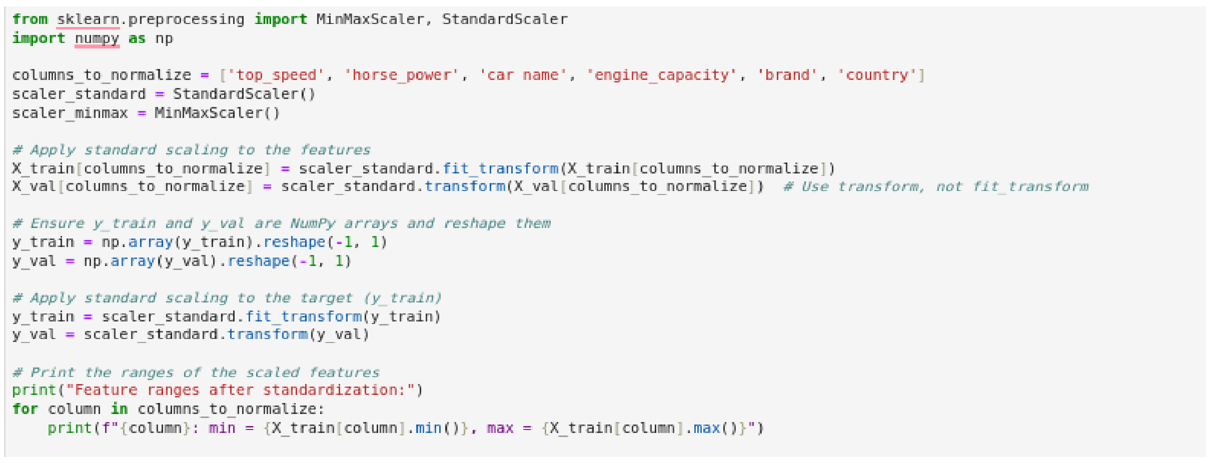


```
[5]: # First, split the data into 80% training+validation and 20% testing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

X = df_upd.drop(columns='price')  # price is the target column
y = df_upd['price']
X_train_val, X_test, y_train_val, y_test = train_test_split(X , y, test_size=0.2, random_state=42)

# Then, split the 80% training+validation into 60% training and 20% validation
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25, random_state=42)


# Print the sizes of each split
print("Training set size:", X_train.shape, y_train.shape)
print("Validation set size:", X_val.shape, y_val.shape)
print("Test set size:", X_test.shape, y_test.shape)
```

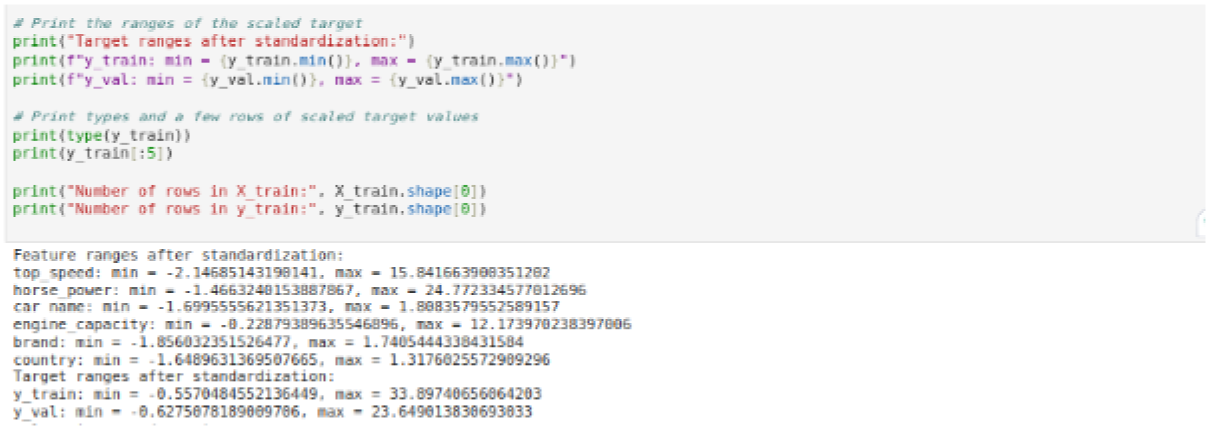
```
Training set size: (3784, 8) (3784,)
Validation set size: (1262, 8) (1262,)
Test set size: (1262, 8) (1262,)
```

## Standardizing and normalizing:

Standardizing and normalizing data ensure all features contribute equally to model training, regardless of scale, improving performance and accuracy. It speeds up convergence for gradient-based algorithms, enhances stability for regularized models, and ensures consistent target values in regression tasks. These steps are crucial for building reliable and efficient machine learning models.Normalization should generally be done **after splitting the data** into training, validation, and test sets. This ensures that the scaling process is only based on the training data, preventing information leakage from the validation or test sets into the model during training. Using the training set statistics (e.g., mean and standard deviation) to scale the other sets maintains the integrity of the evaluation process and ensures the model generalizes well to unseen data.



```
from sklearn.preprocessing import MinMaxScaler, StandardScaler
import numpy as np

columns_to_normalize = ['top_speed', 'horse_power', 'car name', 'engine_capacity', 'brand', 'country']
scaler_standard = StandardScaler()
scaler_minmax = MinMaxScaler()

# Apply standard scaling to the features
X_train[columns_to_normalize] = scaler_standard.fit_transform(X_train[columns_to_normalize])
X_val[columns_to_normalize] = scaler_standard.transform(X_val[columns_to_normalize])  # Use transform, not fit_transform

# Ensure y_train and y_val are NumPy arrays and reshape them
y_train = np.array(y_train).reshape(-1, 1)
y_val = np.array(y_val).reshape(-1, 1)

# Apply standard scaling to the target (y_train)
y_train = scaler_standard.fit_transform(y_train)
y_val = scaler_standard.transform(y_val)

# Print the ranges of the scaled features
print("Feature ranges after standardization:")
for column in columns_to_normalize:
    print(f"{column}: min = {X_train[column].min()}, max = {X_train[column].max()}")
```



```
# Print the ranges of the scaled target
print("Target ranges after standardization:")
print(f"y_train: min = {y_train.min()}, max = {y_train.max()}")
print(f"y_val: min = {y_val.min()}, max = {y_val.max()}")

# Print types and a few rows of scaled target values
print(type(y_train))
print(y_train[:5])

print("Number of rows in X_train:", X_train.shape[0])
print("Number of rows in y_train:", y_train.shape[0])
```

```
Feature ranges after standardization:
top_speed: min = -2.14685143190141, max = 15.841663900351202
horse_power: min = -1.4663240153887867, max = 24.772334577012696
car name: min = -1.6995555621351373, max = 1.8083579552589157
engine_capacity: min = -0.22879389635546896, max = 12.173970238397006
brand: min = -1.856032351526477, max = 1.7405444338431584
country: min = -1.6489631369507665, max = 1.3176025572909296
Target ranges after standardization:
y_train: min = -0.5570484552136449, max = 33.89740656064203
y_val: min = -0.6275078189009706, max = 23.649013830693033
```

## Building Regression Models:

Regression analysis is a statistical method used to identify and measure the relationships between variables in a dataset. It helps describe how one or more independent variables impact a dependent or target variable. Various types of regression models are suitable for different situations. For instance, linear regression can be used to examine the relationship between a person's height and weight. Regression analysis is valuable for making predictions and understanding how specific variables influence a target outcome, which can be crucial for businesses and decision-making processes.
Each model was evaluated on both the validation and testing sets, and performance metrics such as mean squared error and R^2 were computed.

**Linear Models:** Linear regression is a regression modeling technique that interprets the relationship between inputs and outputs as a straight line or plane. It provides a straightforward approach, making it easy to understand and apply to real-world scenarios. Even when relationships are not naturally linear, we often attempt to identify patterns and use linear models to explain them. The goal of linear regression is to establish a relationship between one or more independent variables (features) and a continuous dependent variable (target). When there is a single feature, it is referred to as Univariate Linear Regression, whereas models with multiple features are called Multiple Linear Regression.

The linear regression process begins by initializing the model with LinearRegression() and training it on the dataset using model.fit(X_train, y_train). The model learns the coefficients and intercept, which define the regression equation. Predictions are made on the validation and test datasets, with performance evaluated using metrics like mean_squared_error and r2_score.

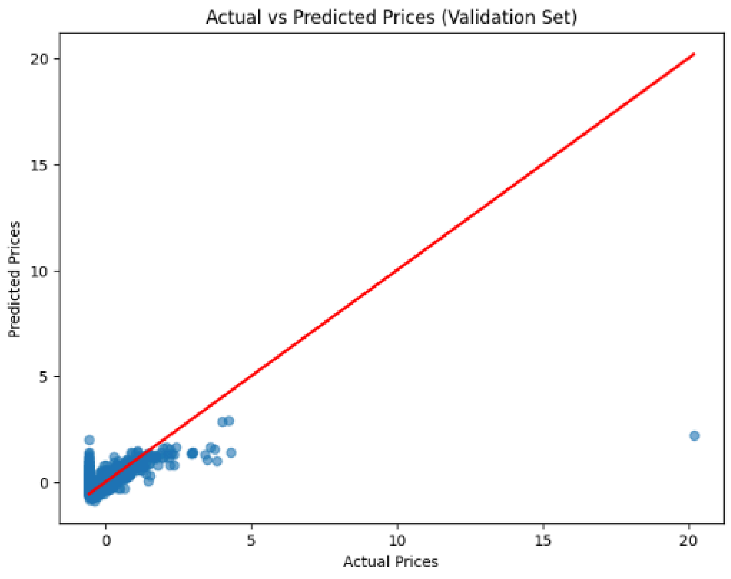
The linear regression model can be represented by the following equation

$$Y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + .... + \theta_n x_n$$

The line for which the error between the predicted values and the observed values is minimum is called the best fit line or the regression line. These errors are also called residuals. The residuals can be visualized by the vertical lines from the observed data value to the regression line.
The equation took the form:
**y = -0.61 + (0.20) * car name + (-0.05) * engine_capacity + (0.12) * cylinder + (0.21) * horse_power + (0.24) * top_speed + (-0.00) * seats + (-0.17) * brand + (-0.03) * country.**

Each coefficient reflects how much a feature influences the target variable, with positive values indicating a positive correlation and negative values indicating an inverse relationship. Some features, like "seats," have minimal impact, as seen by the coefficient near zero.

```
Linear regression equation:
y = -0.61 + (0.20) * car name + (-0.05) * engine_capacity + (0.12) * cylinder + (0.21) * horse_power + (0.24) * top_speed + (-0.00) * seats
+ (-0.17) * brand + (-0.03) * country


Training Mean Squared Error: 0.69
Training R² Score: 0.31
Validation Mean Squared Error: 0.49
Validation R² Score: 0.33
Testing Mean Squared Error: 0.56
Testing R² Score: 0.34
```



**Ridge and Lasso Regression** are two of the most popular algorithms used in the field of machine learning. Both algorithms are used to reduce the errors of a **linear regression** model, but there are key differences between the two which make them suitable for different types of problems.

## Ridge Regression

Ridge regression is a type of regularized regression model. This means it is a variation of the standard linear regression model that includes a regularized term in the cost function. The purpose of this is to prevent Overfitting. Ridge Regression adds an L2 regularization term to the linear equation. That’s why it is also known as L2 Regularization or L2 Norm.
The main aim of Ridge Regression is to reduce Overfitting.
In ridge regression, the cost function is changed by adding a penalty term to the square of the magnitude of the coefficients.

$$Cost\ Function = \frac{1}{n}\sum_{i=1}^{n}\left(h_\theta(x)^i - y^i\right)^2 + \lambda \sum_{i=1}^{n} |slope|$$

$$\lambda = Hyperparameter$$

In Ridge regression, the regularization strength α\alpha plays a crucial role in determining the model's complexity and performance. To find the optimal value of α, **Grid Search** is commonly used for **hyperparameter tuning**. Grid Search is an exhaustive search technique that evaluates multiple hyperparameter values to find the best configuration for the model. In this context, Grid Search helps determine the ideal value of α for Ridge regression by testing different values of α and assessing the model's performance based on validation data.

The code implements Ridge Regression with hyperparameter tuning using Grid Search. It performs the following steps:

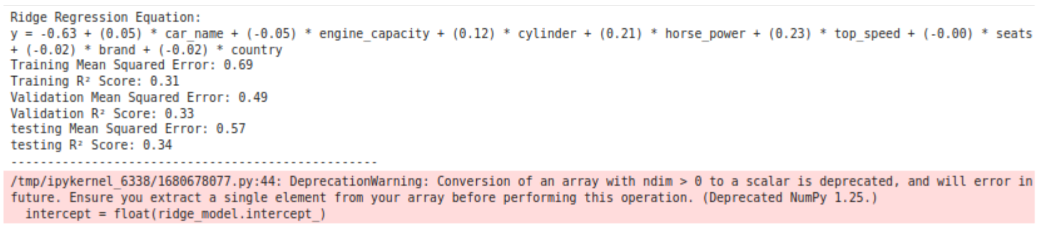
1. **Ridge Regression Model**: A Ridge model is created and trained on the training data using the `Ridge()` class from scikit-learn.
2. **Grid Search**: It searches for the best value of the regularization parameter (α\alphaα) by testing different values in the range [0.01, 0.1, 1, 10, 100] using `GridSearchCV` from scikit-learn.
3. **Evaluation**: It evaluates the model on both training and validation data, calculating Mean Squared Error (MSE) and $R^2$ score using `mean_squared_error()` and `r2_score()` from scikit-learn.
4. **Equation Display**: It prints the Ridge regression equation with the optimal coefficients and intercept derived from the best model found during the grid search.
5. **Plotting**: It generates plots to visualize the relationship between actual vs. predicted values and the effect of α\alphaα on model performance. The first plot compares actual vs predicted values, and the second plot shows the effect of α\alphaα on the validation Mean Squared Error.

This method helps find the best regularization strength to optimize the model's performance while avoiding overfitting.

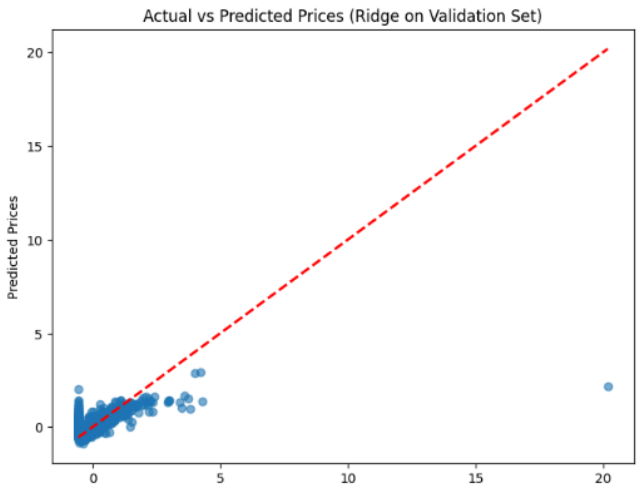
The Ridge Regression equation based on the output is:

**y = -0.63 + (0.05) * car_name + (-0.05) * engine_capacity + (0.12) * cylinder + (0.21) * horse_power + (0.23) * top_speed + (-0.00) * seats + (-0.02) * brand + (-0.02) * country**

So we can see that Ridge Regression applies regularization, shrinking the coefficients compared to Linear Regression. For example, the coefficient for car_name is smaller in Ridge regression (0.05 vs. 0.20), showing the effect of regularization. This helps prevent overfitting, leading to a more generalizable model compared to Linear Regression, which doesn't have regularization.



```
Ridge Regression Equation:
y = -0.63 + (0.05) * car_name + (-0.05) * engine_capacity + (0.12) * cylinder + (0.21) * horse_power + (0.23) * top_speed + (-0.00) * seats
+ (-0.02) * brand + (-0.02) * country
Training Mean Squared Error: 0.69
Training R² Score: 0.31
Validation Mean Squared Error: 0.49
Validation R² Score: 0.33
testing Mean Squared Error: 0.57
testing R² Score: 0.34
--------------------------------------------------
/tmp/ipykernel_6338/1680678077.py:44: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in
future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)
  intercept = float(ridge_model.intercept_)
```



## Lasso Regression

L1 Regularization, Lasso regression, also known as L1 regularization, is a linear regression technique that adds a penalty to the loss function to prevent overfitting. This penalty is based on the absolute values of the coefficients. Lasso regression is a linear regression version that includes a penalty equal to the absolute value of the coefficient magnitude. By encouraging sparsity, this L1 regularization term reduces overfitting and helps some coefficients to zero, facilitating feature selection.

The standard loss function (mean squared error) is modified to include a regularization term:

$$L_{lasso}(\hat{\beta}) = \sum_{i=1}^{n}(y_i - x_i'\hat{\beta})^2 + \lambda \sum_{j=1}^{m}|\hat{\beta}_j|.$$

In Lasso regression, the regularization strength α\alpha is essential for determining the model's complexity and feature selection. Grid Search, a hyperparameter tuning technique, is commonly used to find the optimal α\alpha. This exhaustive method evaluates various values of α\alpha, such as [0.01, 0.1, 1, 10, 100], to identify the configuration that minimizes the validation error.

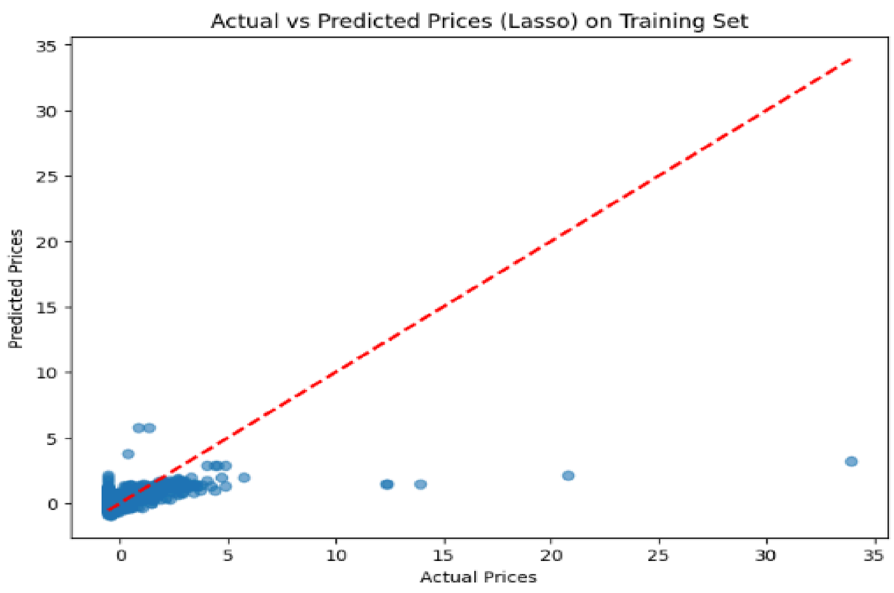
The implementation of Lasso regression with Grid Search involves several steps:

First, a Lasso model is created and trained on the data using the `Lasso()` class from scikit-learn. Then, Grid Search tests different α\alpha values using `GridSearchCV` to determine the best regularization parameter. The model's performance is evaluated on both training and validation data by calculating metrics like Mean Squared Error (MSE) and $R^2$ scores with scikit-learn's `mean_squared_error()` and `r2_score()`.

The final model equation is displayed, showing the optimal coefficients and intercept derived during the grid search. Visualization is also performed to analyze the relationship between actual vs. predicted values and the effect of α\alpha on validation performance. Lasso regression applies regularization by shrinking some coefficients to zero, effectively selecting relevant features.

This process results in a more interpretable and generalizable model that balances performance and complexity, making it superior to standard linear regression in cases prone to overfitting or involving irrelevant features.

```
Lasso Regression Equation:
y = -0.63 + (0.02) * car_name + (-0.04) * engine_capacity + (0.12) * cylinder + (0.20) * horse_power + (0.23) * top_speed + (-0.00) * seats
+ (0.00) * brand + (-0.01) * country
Training Mean Squared Error: 0.69
Training R² Score: 0.31
--------------------------------------------------
Validation Mean Squared Error: 0.49
Validation R² Score: 0.33
--------------------------------------------------
testing Mean Squared Error: 0.57
testing R² Score: 0.33
--------------------------------------------------
```



## Comparison of Ridge, Lasso, and Linear Regression Results:

### Equations and Coefficients:

**Ridge Regression:**Equation: Shrinks coefficients slightly compared to Linear Regression due to regularization. For example, `car_name` drops from 0.20 (Linear) to 0.05.Regularization reduces overfitting but keeps all features contributing, even with small weights. **Lasso Regression:**Equation: Applies stronger regularization, pushing some coefficients to exactly 0 (e.g., `brand` and `country` are 0). This leads to feature selection, simplifying the model. However, coefficients for `car_name` and `horse_power` are slightly smaller compared to Ridge Regression but not drastically different. **Linear Regression:**Equation: Coefficients are unregularized and larger, reflecting a standard least-squares fit without penalties. For example, `car_name` has the highest coefficient (0.20), which overemphasizes its importance.

---

### Performance Metrics:

| Metric | Ridge Regression | Lasso Regression | Linear Regression |
|---|---|---|---|
| Training Mean Squared Error | 0.69 | 0.69 | 0.69 |
| Training $R^2$ Score | 0.31 | 0.31 | 0.31 |
| Validation Mean Squared Error | 0.49 | 0.49 | 0.49 |
| Validation $R^2$ Score | 0.33 | 0.33 | 0.33 |
| Testing Mean Squared Error | 0.57 | 0.57 | 0.56 |
| Testing $R^2$ Score | 0.34 | 0.33 | 0.34 |

---

### Analysis of Results:

**Performance Similarity:** The performance metrics across Ridge, Lasso, and Linear regression are nearly identical, indicating regularization had minimal impact on improving generalization. This suggests that the dataset may already be simple or well-conditioned, where regularization is unnecessary.**Lasso Feature Selection:** Lasso's feature selection does not significantly improve model performance. Features like `brand` and `country` were effectively removed, but their exclusion had little to no impact on error or $R^2$ scores.

**Linear Regression Advantage:** Linear Regression slightly outperforms Ridge and Lasso on testing data with a marginally lower Mean Squared Error (0.56 vs. 0.57) and a higher $R^2$ Score (0.34 vs. 0.33). This is unexpected, as regularization typically prevents overfitting, particularly when validation and testing scores differ significantly from training scores.

**Why Results Are Not as Expected:**

The dataset may not be complex or noisy enough for regularization to improve performance significantly. Features might already be orthogonal or well-scaled, reducing multicollinearity and mitigating the benefits of Ridge or Lasso.Regularization is most beneficial when overfitting is a concern (e.g., training performance is high but validation/testing is poor). Here, training, validation, and testing metrics are similar, indicating no overfitting. **Hyperparameter Range:**The α range for Ridge (or α for Lasso) might not have been broad enough to capture optimal regularization. Testing more extreme values of α (e.g., closer to 0 or very large) could reveal stronger effects.

---

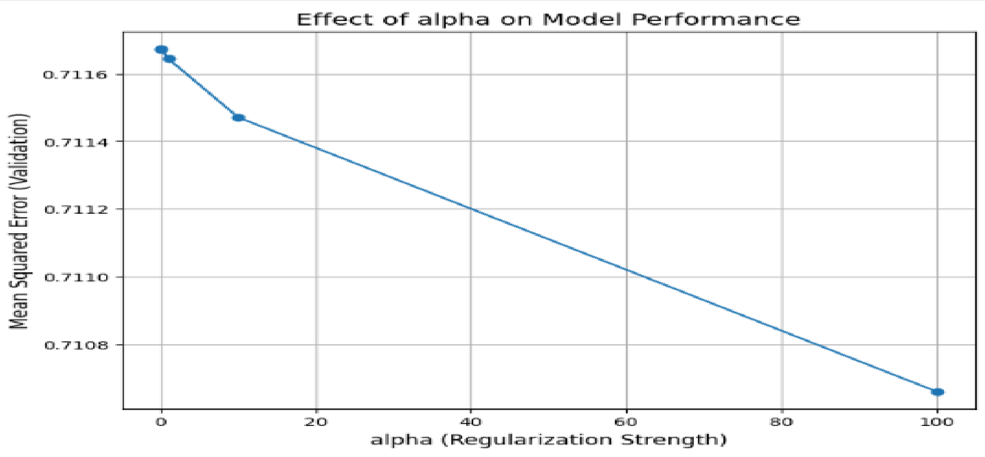
**Expected Results:**

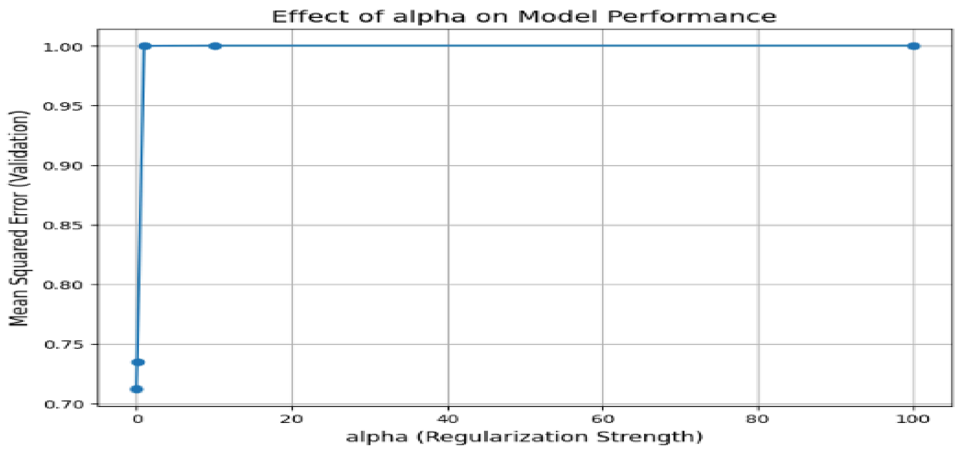
Ridge and Lasso should ideally show better generalization (higher R² scores and lower testing MSE) compared to Linear Regression due to controlled complexity and reduced overfitting. The removal of weakly contributing features (`brand` and `country`) should lead to a simpler and possibly more interpretable model, with slightly improved validation/testing performance. With a well-tuned α, regularized models should outperform Linear Regression, particularly on validation and testing datasets, indicating reduced overfitting and better robustness.

---

## Hyperparameter Tuning with Grid Search Apply grid search

to find the best hyperparameters for each model (e.g., λ for regularized models). This step ensures that each model is tuned for optimal performance For regularized models like **Ridge** and **Lasso**, we tested various values of λ (e.g., `[0.01, 0.1, 1, 10, 100]`) to minimize validation errors and improve **R²** scores.





```
Best alpha for Ridge: 100
Best alpha for Lasso: 0.01
```

**Linear Regression using closed form solution:**
This is a direct mathematical formula used to compute the optimal values of the model parameters (weights) without requiring iterative optimization methods like gradient descent. It is derived by minimizing the cost function (Mean Squared Error) analytically. Weights are computed using the following formula:

**weights = (X_transpose . X)^-1 . (X_transpose . y)**

- X is the matrix of input features, and X^T is its transpose.
- X^T * X combines the features to understand their relationships.
- (X^T * X)^-1 is the inverse of this matrix, which helps in solving for the best weights.
- X^T * y is the multiplication of the transposed input matrix and the target values.

The result, w, gives the optimal weights that minimize the error in predicting y.

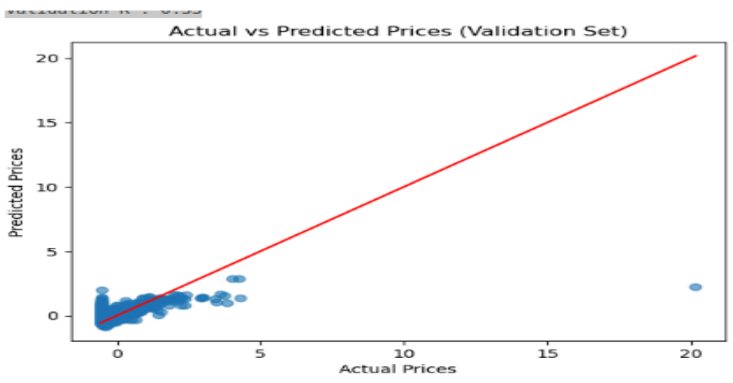
Using this method gave very close results to using the LinearRegression() API.

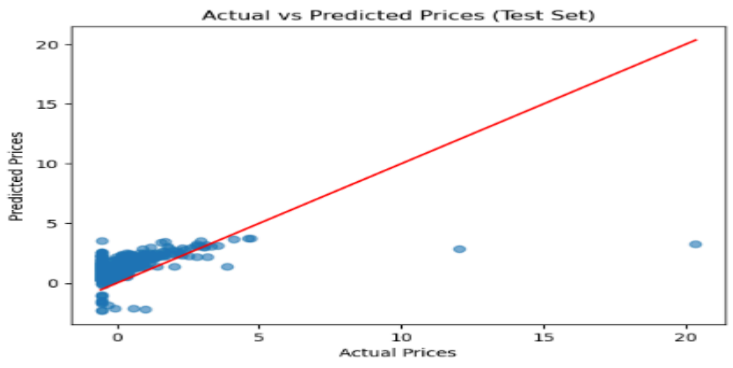
```
using closed form solution
Linear regression equation: y = (-0.61) * intercept + (0.20) * car name + (-0.05) * engine_capacity + (0.12) * cylinder + (0.21) * horse_power + (0.24) *
top_speed + (-0.00) * seats + (-0.17) * brand + (-0.03) * country
Validation Mean Squared Error: 0.49
Validation R² Score: 0.33
Testing Mean Squared Error: 0.56
Testing R² Score: 0.34
```

---

**Linear Regression using gradient descent:** this is another way of solving the linear regression problem. Gradient descent aims to find the value of parameters that minimize the mean squared error. Gradient descent updates the parameters iteratively depending on the learning rate and the gradient values.
Gradient descent gave almost identical results for the parameters as in using the closed form solution.

```
solution using gradient descent:
y = -0.00 + (0.20) * car name + (-0.05) * engine_capacity + (0.22) * cylinder + (0.21) * horse_power + (0.24) * top_speed + (-0.01) * seats + (-0.17) * b
rand + (-0.03) * country
Validation MSE: 0.49
Validation R²: 0.33
```





### Comparison of the Two Models:

**Coefficients Comparison**: Both models have very similar coefficients for features like car name, engine capacity, horse power, top speed, brand, and country. However, there are minor differences: The intercept in Model 1 is -0.61, while in Model 2, it is -0.00. The coefficient for cylinder in Model 1 is 0.12, whereas in Model 2, it is 0.22. The coefficient for seats is slightly different: Model 1 has -0.00, and Model 2 has -0.01.

Performance Metrics: The Validation MSE and $R^2$ scores for both models are identical, indicating that their performance on validation data is very similar: Validation MSE: Both models have a score of 0.49. Validation $R^2$: Both models have a score of 0.33. The Testing MSE and $R^2$ scores for the closed-form solution are slightly different than those for the gradient descent solution. For Model 1: Testing MSE: 0.56 and Testing $R^2$: 0.34.

**Conclusion:** Both models show similar performance, with very close MSE and $R^2$ values for validation. The coefficients are very similar, suggesting that both methods are finding a similar solution.The closed-form solution directly computes the coefficients using matrix operations, while gradient descent is an iterative optimization method that gradually adjusts the coefficients to minimize the error.In practice, both methods give similar results here, but gradient descent is more commonly used in larger datasets or when closed-form solutions are computationally expensive or not feasible.

---

## Polynomial Regression:

Polynomial regression is a non-linear regression where the relationship between the inputs (x) and the output (y) is modeled as an n-degree polynomial. It extends linear regression by introducing non-linearity to fit more complex datasets.
We varied the degrees of the polynomials, taking the degrees 2, 5, 7 and 10. Also, Feature selection with forward selection was applied to the polynomial models starting with an empty model then iteratively adding features that improve the performance of the model instead of taking all the features. This reduced the complexity of the model, and therefore reduced the overfitting in the model.

### Implementation :

The code performs the following steps to evaluate Polynomial Regression models with different degrees: Polynomial Transformation: For each degree in degrees, the code transforms the input features (X) using PolynomialFeatures(degree=degree). This adds polynomial terms (like X2X^2, X3X^3, etc.) to the features. Training the Model: A LinearRegression model is trained on the transformed training data (X_train_poly).

Model Evaluation: The model is tested on the transformed training set (X_train_poly), and the Mean Squared Error (MSE) is computed. The model is then tested on the validation set (X_val_poly), and the MSE for the validation set is calculated. Finally, the model is tested on the test set (X_test_poly), and the MSE for the test set is computed.

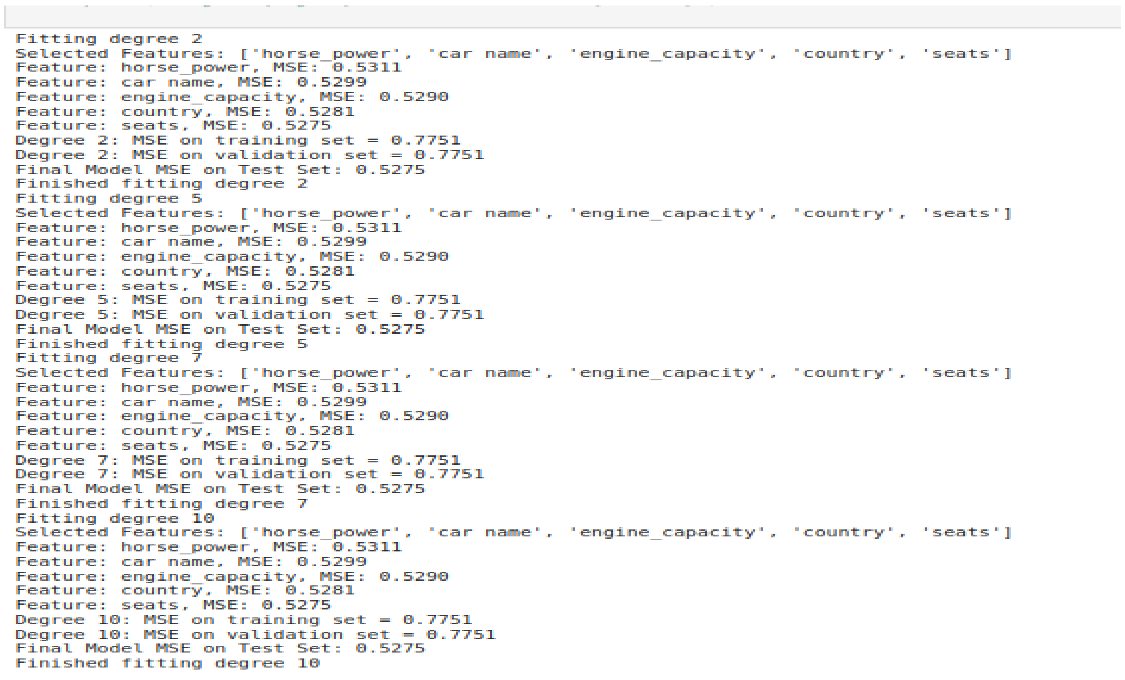
Output: The code prints the MSE for each degree on all three datasets: training, validation, and test.

This process helps in evaluating how the polynomial degree affects the model's performance and whether higher degrees result in overfitting or improved performance.

**Feature Selection with Forward Selection**

The Forward Selection function is used to iteratively improve a model by adding features one at a time, starting with an empty model. Initially, all features are available for selection. The process involves selecting the feature that, when added to the model, minimizes the mean squared error (MSE) on the validation set. The function works by training and testing the model with each available feature, calculating the MSE for each feature's addition, and selecting the feature that results in the lowest MSE. This process continues until no additional features improve the model's performance, or a specified maximum number of features is reached. The function returns the selected features and the history of the MSE values after each feature addition.

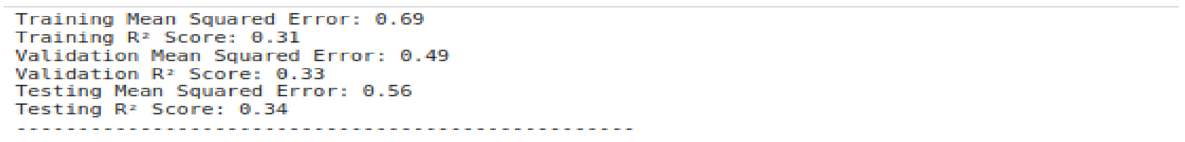
For each polynomial degree (2, 5, 7, and 10), the selected features were identical, and the final model's performance on the test set remained unchanged with an MSE of 0.5275. This suggests that the degree of the polynomial transformation did not significantly impact the model's performance, and the selected features contributed most to the model's predictive power.



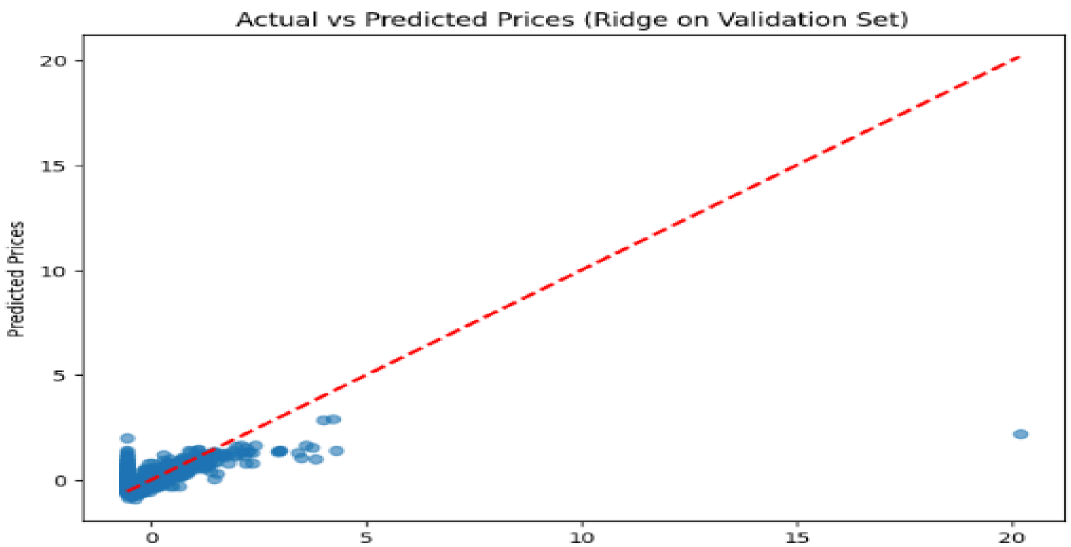
```
Fitting degree 2
Selected Features: ['horse_power', 'car name', 'engine_capacity', 'country', 'seats']
Feature: horse_power, MSE: 0.5311
Feature: car name, MSE: 0.5299
Feature: engine_capacity, MSE: 0.5290
Feature: country, MSE: 0.5281
Feature: seats, MSE: 0.5275
Degree 2: MSE on training set = 0.7751
Degree 2: MSE on validation set = 0.7751
Final Model MSE on Test Set: 0.5275
Finished fitting degree 2
Fitting degree 5
Selected Features: ['horse_power', 'car name', 'engine_capacity', 'country', 'seats']
Feature: horse_power, MSE: 0.5311
Feature: car name, MSE: 0.5299
Feature: engine_capacity, MSE: 0.5290
Feature: country, MSE: 0.5281
Feature: seats, MSE: 0.5275
Degree 5: MSE on training set = 0.7751
Degree 5: MSE on validation set = 0.7751
Final Model MSE on Test Set: 0.5275
Finished fitting degree 5
Fitting degree 7
Selected Features: ['horse_power', 'car name', 'engine_capacity', 'country', 'seats']
Feature: horse_power, MSE: 0.5311
Feature: car name, MSE: 0.5299
Feature: engine_capacity, MSE: 0.5290
Feature: country, MSE: 0.5281
Feature: seats, MSE: 0.5275
Degree 7: MSE on training set = 0.7751
Degree 7: MSE on validation set = 0.7751
Final Model MSE on Test Set: 0.5275
Finished fitting degree 7
Fitting degree 10
Selected Features: ['horse_power', 'car name', 'engine_capacity', 'country', 'seats']
Feature: horse_power, MSE: 0.5311
Feature: car name, MSE: 0.5299
Feature: engine_capacity, MSE: 0.5290
Feature: country, MSE: 0.5281
Feature: seats, MSE: 0.5275
Degree 10: MSE on training set = 0.7751
Degree 10: MSE on validation set = 0.7751
Final Model MSE on Test Set: 0.5275
Finished fitting degree 10
```

---

**Radial Basis Function (RBF):** RBF is a non-linear mapping technique that can capture complex relationships between features by mapping them to higher-dimensional spaces. It measures similarity between points in feature space based on their Euclidean distance, making it widely used in regression. However, to ensure good performance, RBF requires tuning of the hyperparameters alpha and gamma, to avoid overfitting and underfitting.



```
Training Mean Squared Error: 0.69
Training R² Score: 0.31
Validation Mean Squared Error: 0.49
Validation R² Score: 0.33
Testing Mean Squared Error: 0.56
Testing R² Score: 0.34
--------------------------------------------------
```



## Model Evaluation on Test Set

To determine which regression model is the best, we need to evaluate them based on several metrics, including:

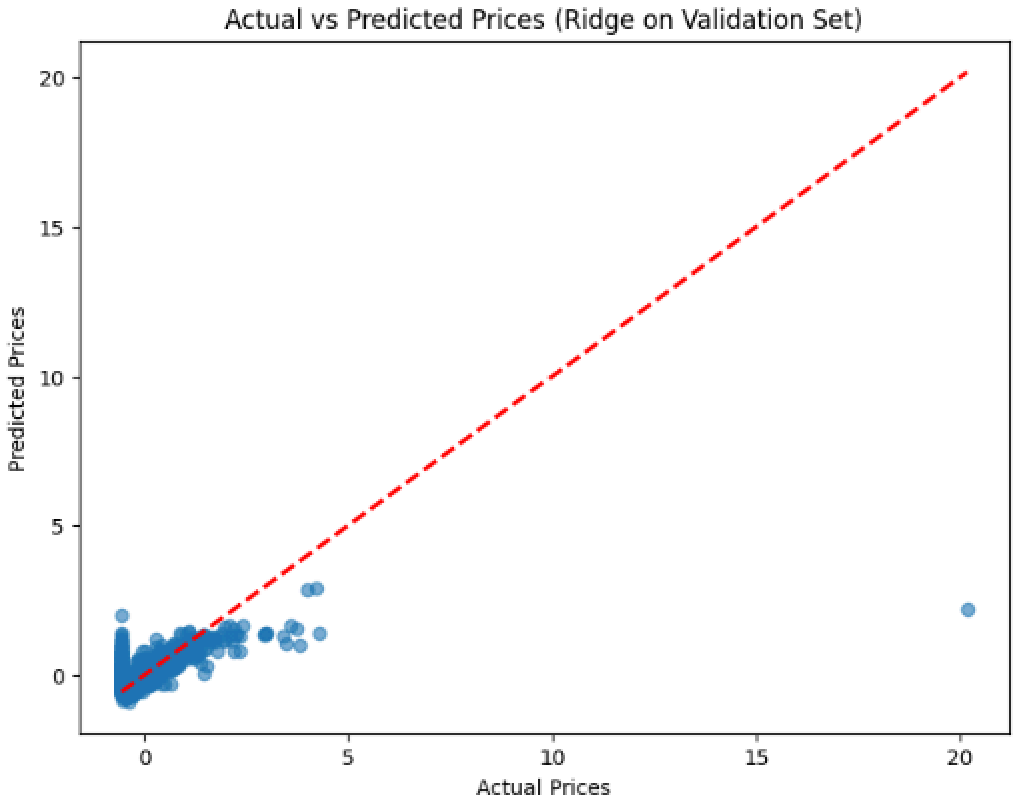
1. **Mean Squared Error (MSE)**: Lower MSE values indicate better model performance, as they reflect less error in predictions.
2. **R² Score**: The R² score measures the proportion of variance in the dependent variable that is explained by the model. A higher R² score indicates that the model is a better fit for the data.

RBF Kernel Regression (Radial Basis Function Kernel) performs well for our dataset, especially in terms of having the smallest validation error with 0.59

```python
# Predictions on testing data for Ridge Regression
y_pred_test = ridge_model.predict(X_test)
mse_test = mean_squared_error(y_test, y_pred_test)
r2_test = r2_score(y_test, y_pred_test)
```

And we get :

```
Testing Mean Squared Error: 0.56
Testing R² Score: 0.34
--------------------------------------------------
```


Actual vs Predicted Prices (Ridge on Validation Set)
Predicted Prices
Actual Prices