Faculty of Engineering and Technology

Electrical and Computer Engineering Department

MACHINE LEARNING AND DATA SCIENCE ENCS5341

Assignment Two

Prepared by:

Hala Jebreel 1210606 - Lana Musaffer 1210455

Instructor: Dr. Ismail Khater

Section: 2

Date: 11/2024

Abstract:

This project aims to develop and assess regression models for predicting car prices, utilizing a dataset from YallaMotors that comprises over 6,750 entries with a variety of features. The research encompasses thorough data preprocessing and cleaning, which involves addressing missing values, encoding categorical variables, and normalizing numerical attributes. Subsequently, the dataset is divided into training, validation, and test subsets. A range of regression models is applied, including both linear and nonlinear approaches such as Linear Regression (using both closed-form and gradient descent methods), LASSO, Ridge Regression, Polynomial Regression, and Radial Basis Function (RBF).

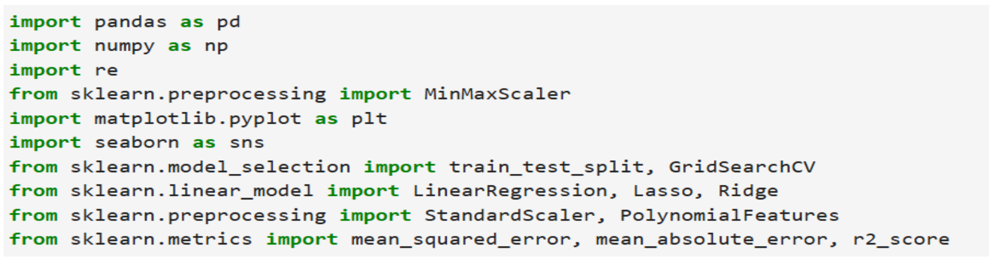
To improve model efficacy and reduce the risk of overfitting, feature selection techniques, including forward selection and regularization methods, are utilized. Hyperparameter optimization is conducted through grid search, and model performance is evaluated using metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-squared. The final assessment on the test set demonstrates the model's capacity to generalize to new, unseen data. Furthermore, this report investigates an alternative target variable, showcasing the adaptability of the employed methodologies. Visual representations are included to enhance the findings, providing insights into feature significance, error distribution, and overall model performance. The outcomes illustrate the advantages and drawbacks of different regression techniques, offering a thorough understanding of predictive modelling within the realm of machine learning.

Table of Content:

Table of figure:

Procedure:

Libraries used:



```
import pandas as pd
import numpy as np
import re
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
```

Figure 1: libraries

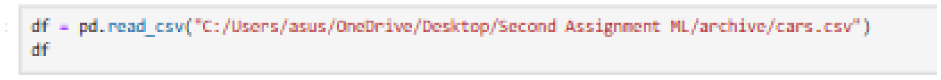
This project employed Python libraries including pandas and numpy for data manipulation, matplotlib and seaborn for creating visual representations, and scikit-learn for developing regression models and evaluating their performance. Essential tools comprised MinMaxScaler, StandardScaler, and PolynomialFeatures for data preprocessing, along with LinearRegression, Lasso, and Ridge for modeling purposes. To evaluate model accuracy, metrics such as mean_squared_error and r2_score were utilized, while GridSearchCV was implemented to optimize hyperparameters.

1. Data Pre-processing Steps

1.1. Dataset Cleaning:

The dataset was cleaned by addressing missing values, standardizing numerical features, and encoding categorical data. This ensures consistency and prepares the data for accurate model training.

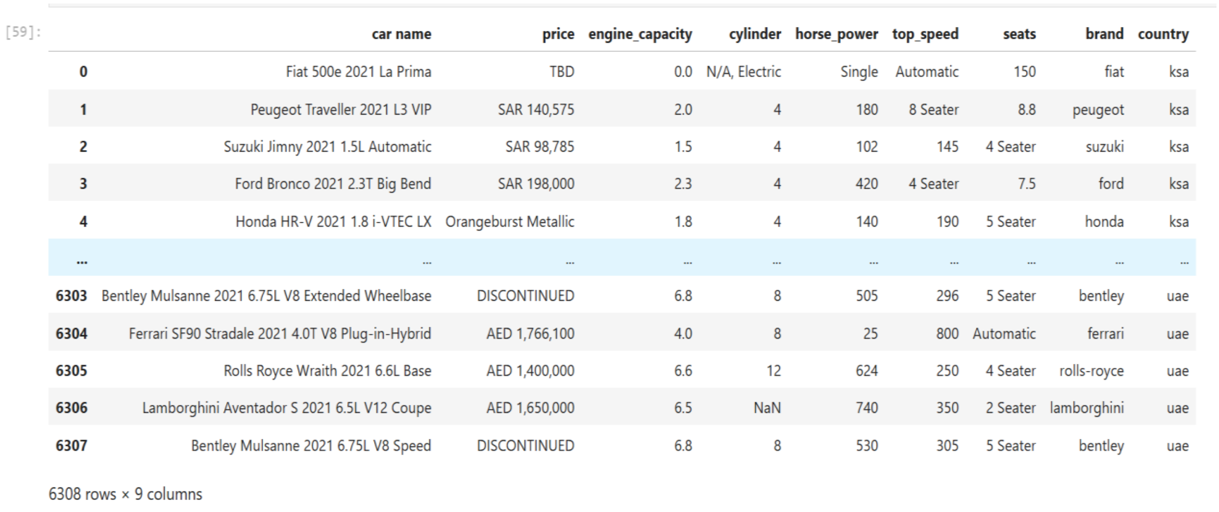
➢ **Open the csv file:**



```
df = pd.read_csv("C:/Users/asus/OneDrive/Desktop/Second Assignment ML/archive/cars.csv")
df
```

Figure 2: open csv file

➢ **Dataset:**



[59]:

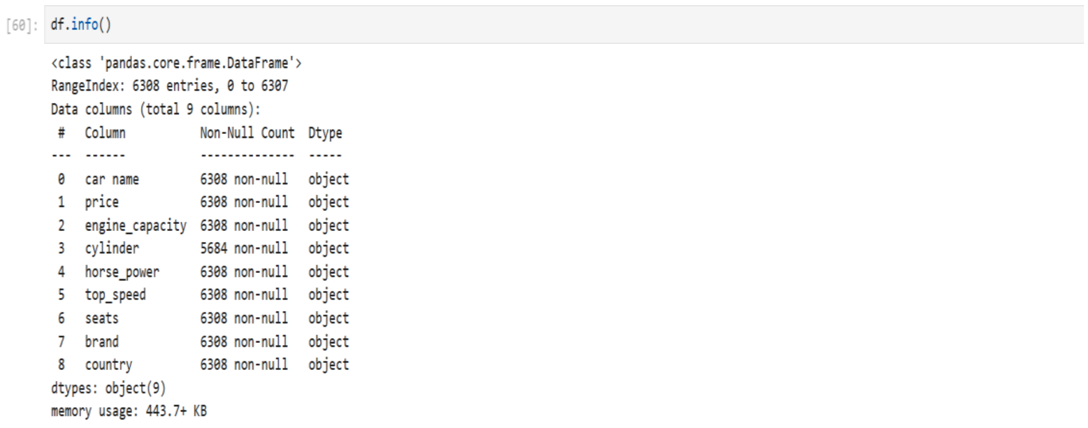
	car name	price	engine_capacity	cylinder	horse_power	top_speed	seats	brand	country
0	Fiat 500e 2021 La Prima	TBD	0.0	N/A, Electric	Single	Automatic	150	fiat	ksa
1	Peugeot Traveller 2021 L3 VIP	SAR 140,575	2.0	4	180	8 Seater	8.8	peugeot	ksa
2	Suzuki Jimny 2021 1.5L Automatic	SAR 98,785	1.5	4	102	145	4 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
...	...	...	...	...	...	...	...	...	...
6303	Bentley Mulsanne 2021 6.75L V8 Extended Wheelbase	DISCONTINUED	6.8	8	505	296	5 Seater	bentley	uae
6304	Ferrari SF90 Stradale 2021 4.0T V8 Plug-in-Hybrid	AED 1,766,100	4.0	8	25	800	Automatic	ferrari	uae
6305	Rolls Royce Wraith 2021 6.6L Base	AED 1,400,000	6.6	12	624	250	4 Seater	rolls-royce	uae
6306	Lamborghini Aventador S 2021 6.5L V12 Coupe	AED 1,650,000	6.5	NaN	740	350	2 Seater	lamborghini	uae
6307	Bentley Mulsanne 2021 6.75L V8 Speed	DISCONTINUED	6.8	8	530	305	5 Seater	bentley	uae

6308 rows × 9 columns

Figure 3:Data Set

1.1.1. Cleaning Data Types and Data Inconsistency:

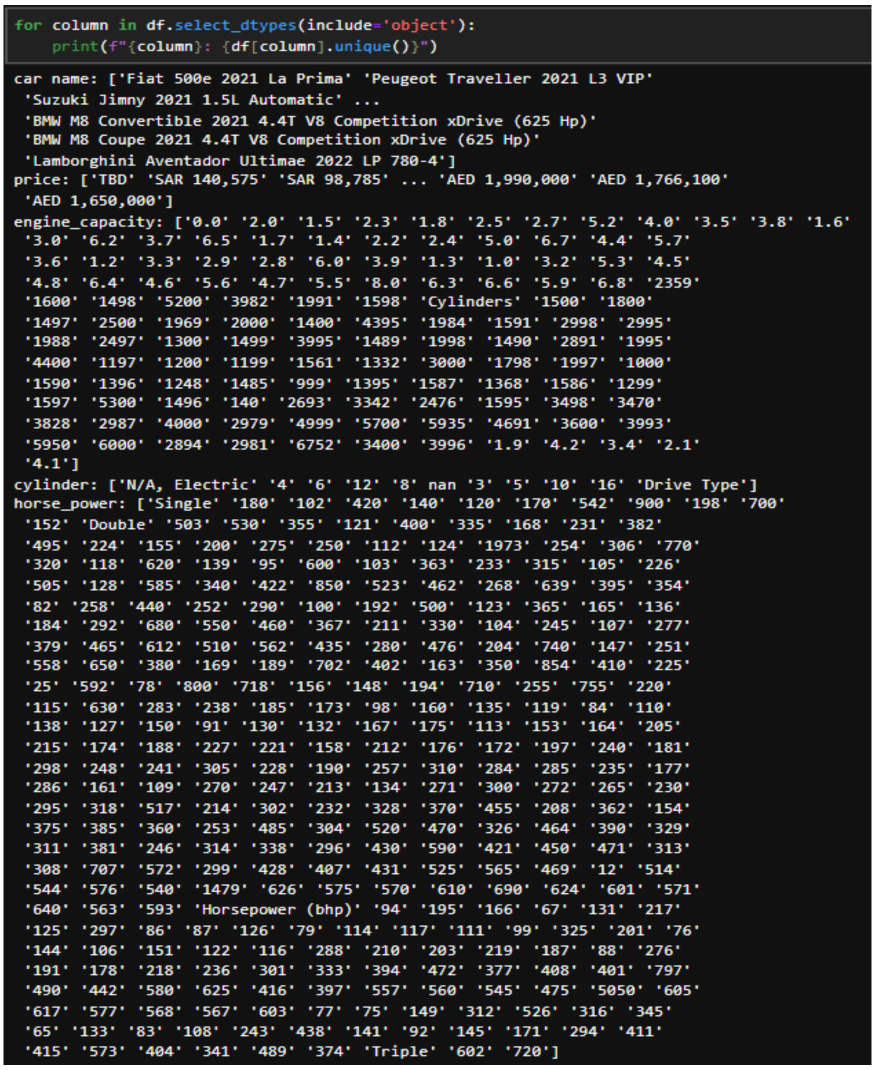
Here is the data set information and the type of each feature:



```
[60]: df.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        6308 non-null   object
 6   seats            6308 non-null   object
 7   brand            6308 non-null   object
 8   country          6308 non-null   object
dtypes: object(9)
memory usage: 443.7+ KB
```

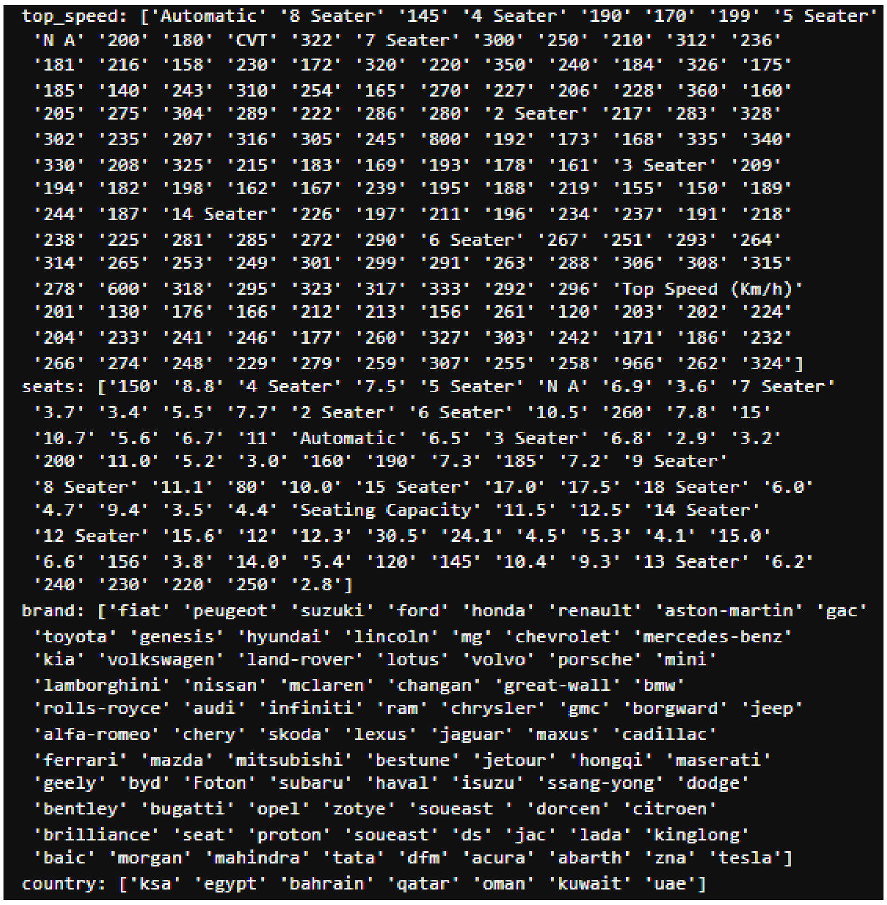
Figure 4 :Data set information

Since all the features are currently of type object, but many contain floating-point values, we will convert them to numerical data types to facilitate statistical analysis and other evaluations. To understand the data, we displayed the unique values for each object:



```
for column in df.select_dtypes(include='object'):
    print(f"{column}: {df[column].unique()}")
```

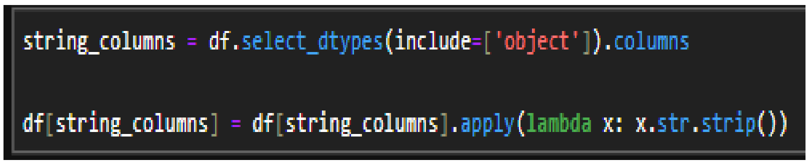
```
car name: ['Fiat 500e 2021 La Prima' 'Peugeot Traveller 2021 L3 VIP'
 'Suzuki Jimny 2021 1.5L Automatic' ...
 'BMW M8 Convertible 2021 4.4T V8 Competition xDrive (625 Hp)'
 'BMW M8 Coupe 2021 4.4T V8 Competition xDrive (625 Hp)'
 'Lamborghini Aventador Ultimae 2022 LP 780-4']
price: ['TBD' 'SAR 140,575' 'SAR 98,785' ... 'AED 1,990,000' 'AED 1,766,100'
 'AED 1,650,000']
engine_capacity: ['0.0' '2.0' '1.5' '2.3' '1.8' '2.5' '2.7' '5.2' '4.0' '3.5' '3.8' '1.6'
 '3.0' '6.2' '3.7' '6.5' '1.7' '1.4' '2.2' '2.4' '5.0' '6.7' '4.4' '5.7'
 '3.6' '1.2' '3.3' '2.9' '2.8' '6.0' '3.9' '1.3' '1.0' '3.2' '5.3' '4.5'
 '4.8' '6.4' '4.6' '5.6' '4.7' '5.5' '8.0' '6.3' '6.6' '5.9' '6.8' '2359'
 '1600' '1498' '5200' '3982' '1991' '1598' 'Cylinders' '1500' '1800'
 '1497' '2500' '1969' '2000' '1400' '4395' '1984' '1591' '2998' '2995'
 '1988' '2497' '1300' '1499' '3995' '1489' '1998' '1490' '2891' '1995'
 '4400' '1197' '1200' '1199' '1561' '1332' '3000' '1798' '1997' '1000'
 '1590' '1396' '1248' '1485' '999' '1395' '1587' '1368' '1586' '1299'
 '1597' '5300' '1496' '140' '2693' '3342' '2476' '1595' '3498' '3470'
 '3828' '2987' '4000' '2979' '4999' '5700' '5935' '4691' '3600' '3993'
 '5950' '6000' '2894' '2981' '6752' '3400' '3996' '1.9' '4.2' '3.4' '2.1'
 '4.1']
cylinder: ['N/A, Electric' '4' '6' '12' '8' nan '3' '5' '10' '16' 'Drive Type']
horse_power: ['Single' '180' '102' '420' '140' '120' '170' '542' '900' '198' '700'
 '152' 'Double' '503' '530' '355' '121' '400' '335' '168' '231' '382'
 '495' '224' '155' '200' '275' '250' '112' '124' '1973' '254' '306' '770'
 '320' '118' '620' '139' '95' '600' '103' '363' '233' '315' '105' '226'
 '505' '128' '585' '340' '422' '850' '523' '462' '268' '639' '395' '354'
 '82' '258' '440' '252' '290' '100' '192' '500' '123' '365' '165' '136'
 '184' '292' '680' '550' '460' '367' '211' '330' '104' '245' '107' '277'
 '379' '465' '612' '510' '562' '435' '280' '476' '204' '740' '147' '251'
 '558' '650' '380' '169' '189' '702' '402' '163' '350' '854' '410' '225'
 '25' '592' '78' '800' '718' '156' '148' '194' '710' '255' '755' '220'
 '115' '630' '283' '238' '185' '173' '98' '160' '135' '119' '84' '110'
 '138' '127' '150' '91' '130' '132' '167' '175' '113' '153' '164' '205'
 '215' '174' '188' '227' '221' '158' '212' '176' '172' '197' '240' '181'
 '298' '248' '241' '305' '228' '190' '257' '310' '284' '285' '235' '177'
 '286' '161' '109' '270' '247' '213' '134' '271' '300' '272' '265' '230'
 '295' '318' '517' '214' '302' '232' '328' '370' '455' '208' '362' '154'
 '375' '385' '360' '253' '485' '304' '520' '470' '326' '464' '390' '329'
 '311' '381' '246' '314' '338' '296' '430' '590' '421' '450' '471' '313'
 '308' '707' '572' '299' '428' '407' '431' '525' '565' '469' '12' '514'
 '544' '576' '540' '1479' '626' '575' '570' '610' '690' '624' '601' '571'
 '640' '563' '593' 'Horsepower (bhp)' '94' '195' '166' '67' '131' '217'
 '125' '297' '86' '87' '126' '79' '114' '117' '111' '99' '325' '201' '76'
 '144' '106' '151' '122' '116' '288' '210' '203' '219' '187' '88' '276'
 '191' '178' '218' '236' '301' '333' '394' '472' '377' '408' '401' '797'
 '490' '442' '580' '625' '416' '397' '557' '560' '545' '475' '5050' '605'
 '617' '577' '568' '567' '603' '77' '75' '149' '312' '526' '316' '345'
 '65' '133' '83' '108' '243' '438' '141' '92' '145' '171' '294' '411'
 '415' '573' '404' '341' '489' '374' 'Triple' '602' '720']
```



```
top_speed: ['Automatic' '8 Seater' '145' '4 Seater' '190' '170' '199' '5 Seater'
 'N A' '200' '180' 'CVT' '322' '7 Seater' '300' '250' '210' '312' '236'
 '181' '216' '158' '230' '172' '320' '220' '350' '240' '184' '326' '175'
 '185' '140' '243' '310' '254' '165' '270' '227' '206' '228' '360' '160'
 '205' '275' '304' '289' '222' '286' '280' '2 Seater' '217' '283' '328'
 '302' '235' '207' '316' '305' '245' '800' '192' '173' '168' '335' '340'
 '330' '208' '325' '215' '183' '169' '193' '178' '161' '3 Seater' '209'
 '194' '182' '198' '162' '167' '239' '195' '188' '219' '155' '150' '189'
 '244' '187' '14 Seater' '226' '197' '211' '196' '234' '237' '191' '218'
 '238' '225' '281' '285' '272' '290' '6 Seater' '267' '251' '293' '264'
 '314' '265' '253' '249' '301' '299' '291' '263' '288' '306' '308' '315'
 '278' '600' '318' '295' '323' '317' '333' '292' '296' 'Top Speed (Km/h)'
 '201' '130' '176' '166' '212' '213' '156' '261' '120' '203' '202' '224'
 '204' '233' '241' '246' '177' '260' '327' '303' '242' '171' '186' '232'
 '266' '274' '248' '229' '279' '259' '307' '255' '258' '966' '262' '324']
seats: ['150' '8.8' '4 Seater' '7.5' '5 Seater' 'N A' '6.9' '3.6' '7 Seater'
 '3.7' '3.4' '5.5' '7.7' '2 Seater' '6 Seater' '10.5' '260' '7.8' '15'
 '10.7' '5.6' '6.7' '11' 'Automatic' '6.5' '3 Seater' '6.8' '2.9' '3.2'
 '200' '11.0' '5.2' '3.0' '160' '190' '7.3' '185' '7.2' '9 Seater'
 '8 Seater' '11.1' '80' '10.0' '15 Seater' '17.0' '17.5' '18 Seater' '6.0'
 '4.7' '9.4' '3.5' '4.4' 'Seating Capacity' '11.5' '12.5' '14 Seater'
 '12 Seater' '15.6' '12' '12.3' '30.5' '24.1' '4.5' '5.3' '4.1' '15.0'
 '6.6' '156' '3.8' '14.0' '5.4' '120' '145' '10.4' '9.3' '13 Seater' '6.2'
 '240' '230' '220' '250' '2.8']
brand: ['fiat' 'peugeot' 'suzuki' 'ford' 'honda' 'renault' 'aston-martin' 'gac'
 'toyota' 'genesis' 'hyundai' 'lincoln' 'mg' 'chevrolet' 'mercedes-benz'
 'kia' 'volkswagen' 'land-rover' 'lotus' 'volvo' 'porsche' 'mini'
 'lamborghini' 'nissan' 'mclaren' 'changan' 'great-wall' 'bmw'
 'rolls-royce' 'audi' 'infiniti' 'ram' 'chrysler' 'gmc' 'borgward' 'jeep'
 'alfa-romeo' 'chery' 'skoda' 'lexus' 'jaguar' 'maxus' 'cadillac'
 'ferrari' 'mazda' 'mitsubishi' 'bestune' 'jetour' 'hongqi' 'maserati'
 'geely' 'byd' 'Foton' 'subaru' 'haval' 'isuzu' 'ssang-yong' 'dodge'
 'bentley' 'bugatti' 'opel' 'zotye' 'soueast ' 'dorcen' 'citroen'
 'brilliance' 'seat' 'proton' 'soueast' 'ds' 'jac' 'lada' 'kinglong'
 'baic' 'morgan' 'mahindra' 'tata' 'dfm' 'acura' 'abarth' 'zna' 'tesla']
country: ['ksa' 'egypt' 'bahrain' 'qatar' 'oman' 'kuwait' 'uae']
```

Figure 5: Understanding the data.

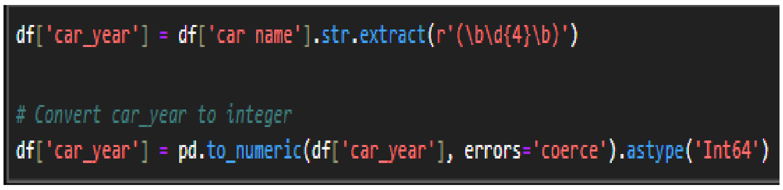
The brand column contains a variety of unique car brands, and most of the values look clean and consistent with a little mixed-up value between features (will be discussed later in the report). Also, note that 'soueast ' (with a trailing space) and 'soueast' appear to be duplicates caused by inconsistent formatting. Thus, we applied the strip() function to remove leading and trailing whitespace from all values:



```
string_columns = df.select_dtypes(include=['object']).columns

df[string_columns] = df[string_columns].apply(lambda x: x.str.strip())
```

Figure 6: Remove leading and trailing whitespaces.

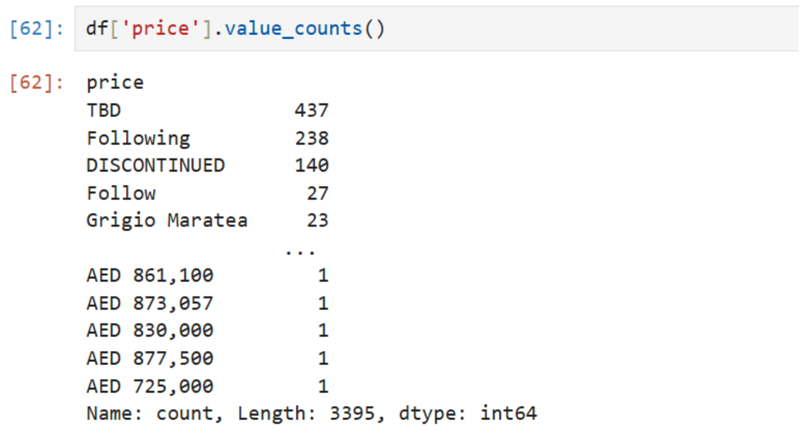
Since the car name feature consists of many essential information, like the year, we extracted the year from this feature so we can use it in the next modelling steps:



```
df['car_year'] = df['car name'].str.extract(r'(\b\d{4}\b)')

# Convert car_year to integer
df['car_year'] = pd.to_numeric(df['car_year'], errors='coerce').astype('Int64')
```

Figure 7: Extract year from car name

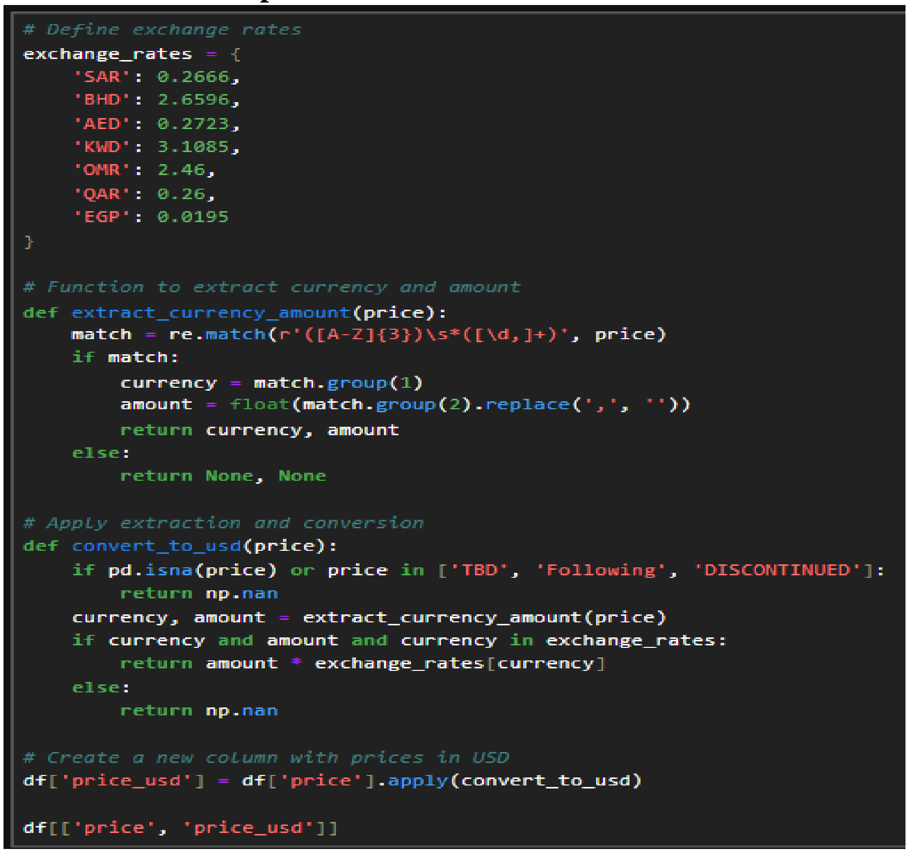
I. Cleaning Price Feature



```
[62]: df['price'].value_counts()

[62]: price
      TBD                437
      Following          238
      DISCONTINUED       140
      Follow              27
      Grigio Maratea      23
                        ...
      AED 861,100          1
      AED 873,057          1
      AED 830,000          1
      AED 877,500          1
      AED 725,000          1
      Name: count, Length: 3395, dtype: int64
```

Figure 8: price feature

➢ **Standardize all prices to USD:**

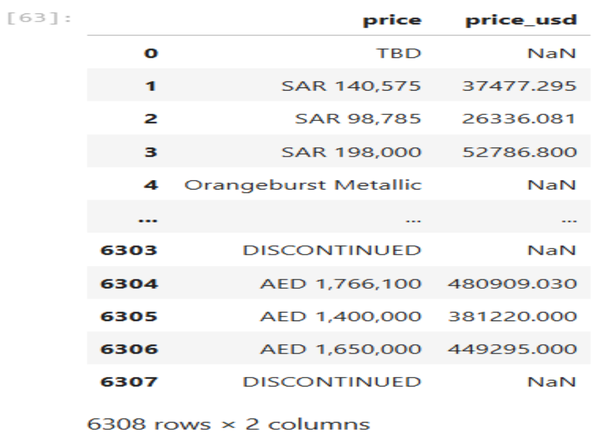


```python
# Define exchange rates
exchange_rates = {
    'SAR': 0.2666,
    'BHD': 2.6596,
    'AED': 0.2723,
    'KWD': 3.1085,
    'OMR': 2.46,
    'QAR': 0.26,
    'EGP': 0.0195
}

# Function to extract currency and amount
def extract_currency_amount(price):
    match = re.match(r'([A-Z]{3})\s*([\d,]+)', price)
    if match:
        currency = match.group(1)
        amount = float(match.group(2).replace(',', ''))
        return currency, amount
    else:
        return None, None

# Apply extraction and conversion
def convert_to_usd(price):
    if pd.isna(price) or price in ['TBD', 'Following', 'DISCONTINUED']:
        return np.nan
    currency, amount = extract_currency_amount(price)
    if currency and amount and currency in exchange_rates:
        return amount * exchange_rates[currency]
    else:
        return np.nan

# Create a new column with prices in USD
df['price_usd'] = df['price'].apply(convert_to_usd)

df[['price', 'price_usd']]
```

Figure 9 :CODE FOR CHANGE THE PRICE TO USD

This code normalizes automobile prices into USD by exchanging rates for various currencies, including `SAR`, `BHD`, `AED`, and others, and implementing a conversion mechanism. It utilizes regular expressions to extract the currency and amount from price strings, converting them to USD when the currency is listed in the specified exchange rates. Invalid entries, such as `'TBD'` and `'DISCONTINUED'`, are addressed by assigning a value of `NaN`. Additionally, a new column named `price_usd` is introduced in the DataFrame to retain the converted prices.

➢ **Output after converting all prices to USD:**



[63]:

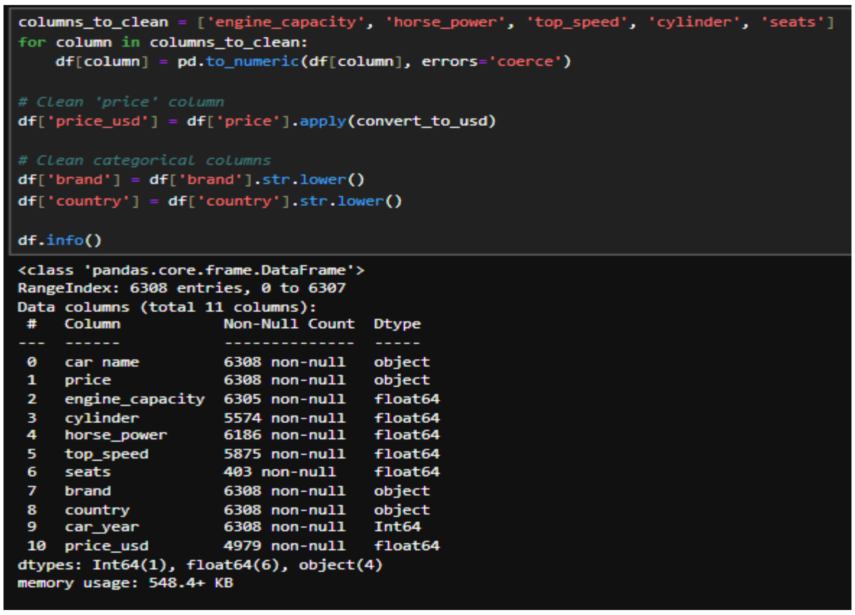
	price	price_usd
0	TBD	NaN
1	SAR 140,575	37477.295
2	SAR 98,785	26336.081
3	SAR 198,000	52786.800
4	Orangeburst Metallic	NaN
...	...	...
6303	DISCONTINUED	NaN
6304	AED 1,766,100	480909.030
6305	AED 1,400,000	381220.000
6306	AED 1,650,000	449295.000
6307	DISCONTINUED	NaN

6308 rows × 2 columns

Figure 10: output from exchange price

II. Cleaning Data Types

We converted numerical columns to numeric format, cleaned the price_usd column, and standardized the brand and country columns by converting them to lowercase:

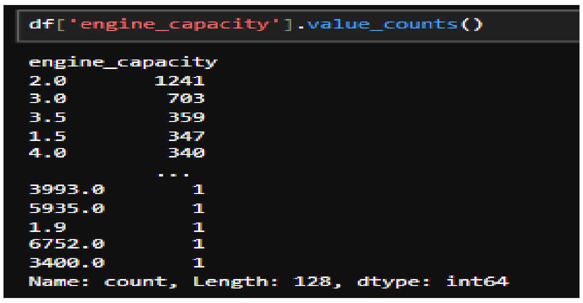


```
columns_to_clean = ['engine_capacity', 'horse_power', 'top_speed', 'cylinder', 'seats']
for column in columns_to_clean:
    df[column] = pd.to_numeric(df[column], errors='coerce')

# Clean 'price' column
df['price_usd'] = df['price'].apply(convert_to_usd)

# Clean categorical columns
df['brand'] = df['brand'].str.lower()
df['country'] = df['country'].str.lower()

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6308 entries, 0 to 6307
Data columns (total 11 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   car name         6308 non-null   object
 1   price            6308 non-null   object
 2   engine_capacity  6305 non-null   float64
 3   cylinder         5574 non-null   float64
 4   horse_power      6186 non-null   float64
 5   top_speed        5875 non-null   float64
 6   seats            403 non-null    float64
 7   brand            6308 non-null   object
 8   country          6308 non-null   object
 9   car_year         6308 non-null   Int64
 10  price_usd        4979 non-null   float64
dtypes: Int64(1), float64(6), object(4)
memory usage: 548.4+ KB
```

Figure 11: Cleaning data types

III. Cleaning Engine Capacity Feature

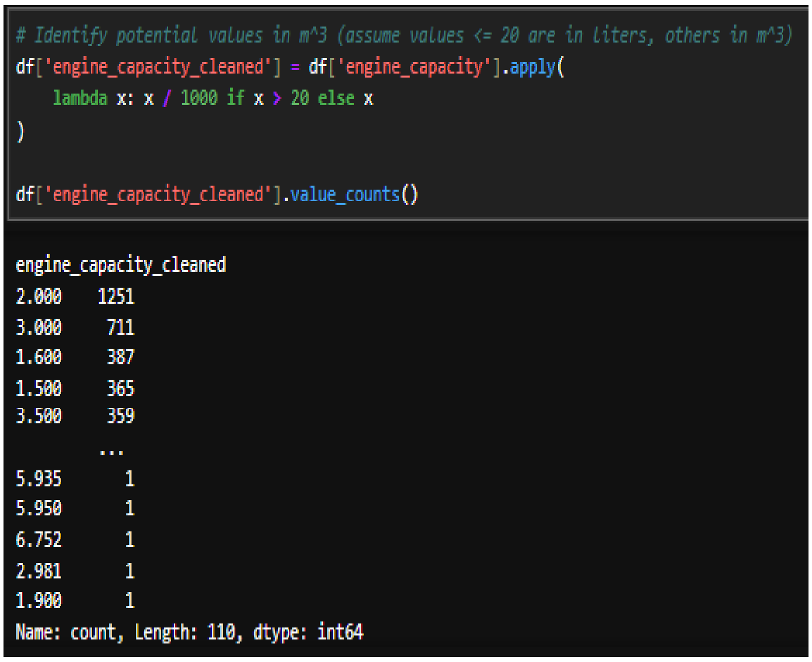


```
df['engine_capacity'].value_counts()
```

```
engine_capacity
2.0       1241
3.0        703
3.5        359
1.5        347
4.0        340
          ...
3993.0       1
5935.0       1
1.9          1
6752.0       1
3400.0       1
Name: count, Length: 128, dtype: int64
```

Figure 12: Engine capacity feature

To clean the engine_capacity feature where some values are in litters and others in cubic meters (m^3), we standardized all the values to a single unit, such as litters. Since 1 m^3=1000 litters, we multiplied any values identified in m^3 by 1000:

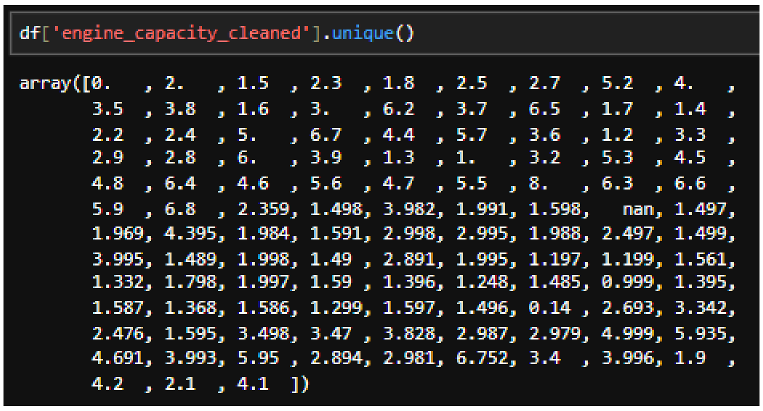


```
# Identify potential values in m^3 (assume values <= 20 are in liters, others in m^3)
df['engine_capacity_cleaned'] = df['engine_capacity'].apply(
    lambda x: x / 1000 if x > 20 else x
)

df['engine_capacity_cleaned'].value_counts()
```

```
engine_capacity_cleaned
2.000    1251
3.000     711
1.600     387
1.500     365
3.500     359
         ...
5.935       1
5.950       1
6.752       1
2.981       1
1.900       1
Name: count, Length: 110, dtype: int64
```

Figure 13: Standardize engine capacity

Note the values after cleaning in this feature:



```
df['engine_capacity_cleaned'].unique()
```

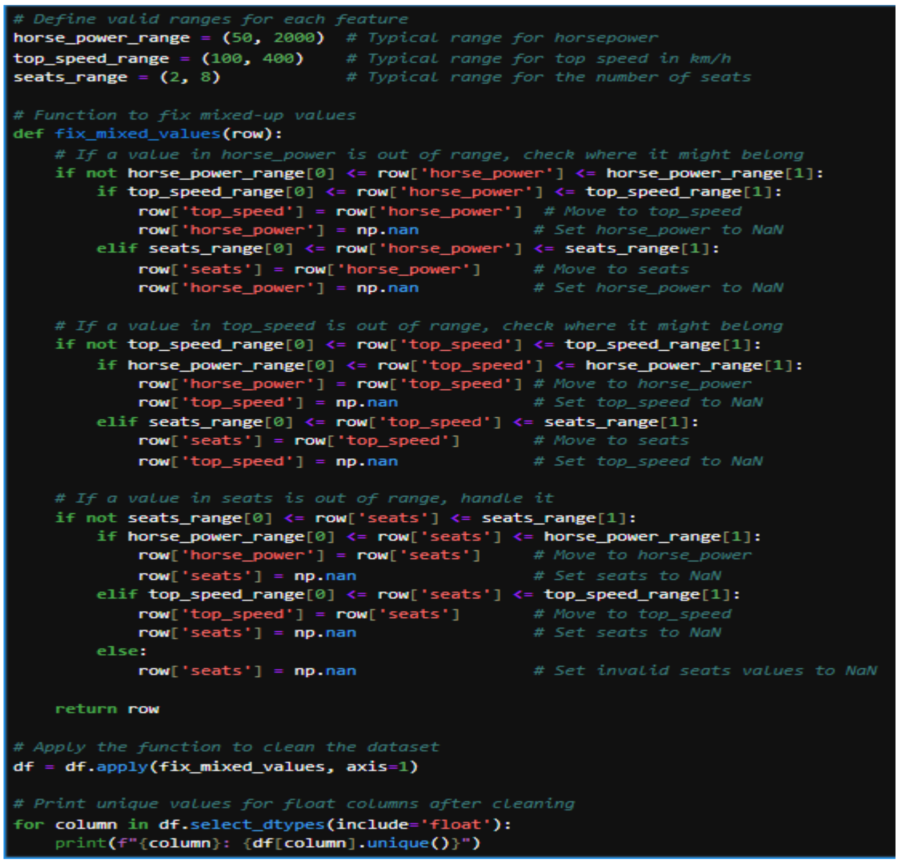
```
array([0.   , 2.   , 1.5  , 2.3  , 1.8  , 2.5  , 2.7  , 5.2  , 4.   ,
       3.5  , 3.8  , 1.6  , 3.   , 6.2  , 3.7  , 6.5  , 1.7  , 1.4  ,
       2.2  , 2.4  , 5.   , 6.7  , 4.4  , 5.7  , 3.6  , 1.2  , 3.3  ,
       2.9  , 2.8  , 6.   , 3.9  , 1.3  , 1.   , 3.2  , 5.3  , 4.5  ,
       4.8  , 6.4  , 4.6  , 5.6  , 4.7  , 5.5  , 8.   , 6.3  , 6.6  ,
       5.9  , 6.8  , 2.359, 1.498, 3.982, 1.991, 1.598,   nan, 1.497,
       1.969, 4.395, 1.984, 1.591, 2.998, 2.995, 1.988, 2.497, 1.499,
       3.995, 1.489, 1.998, 1.49 , 2.891, 1.995, 1.197, 1.199, 1.561,
       1.332, 1.798, 1.997, 1.59 , 1.396, 1.248, 1.485, 0.999, 1.395,
       1.587, 1.368, 1.586, 1.299, 1.597, 1.496, 0.14 , 2.693, 3.342,
       2.476, 1.595, 3.498, 3.47 , 3.828, 2.987, 2.979, 4.999, 5.935,
       4.691, 3.993, 5.95 , 2.894, 2.981, 6.752, 3.4  , 3.996, 1.9  ,
       4.2  , 2.1  , 4.1  ])
```

Figure 14: Engine capacity feature after cleaning

1.1.2. Handling Mixed-Up Values:

- horse_power: Typical range for cars might be 50 to 1500.
- top_speed: Typical range is 100 to 400 km/h.
- seats: Typical range is 2 to 8.

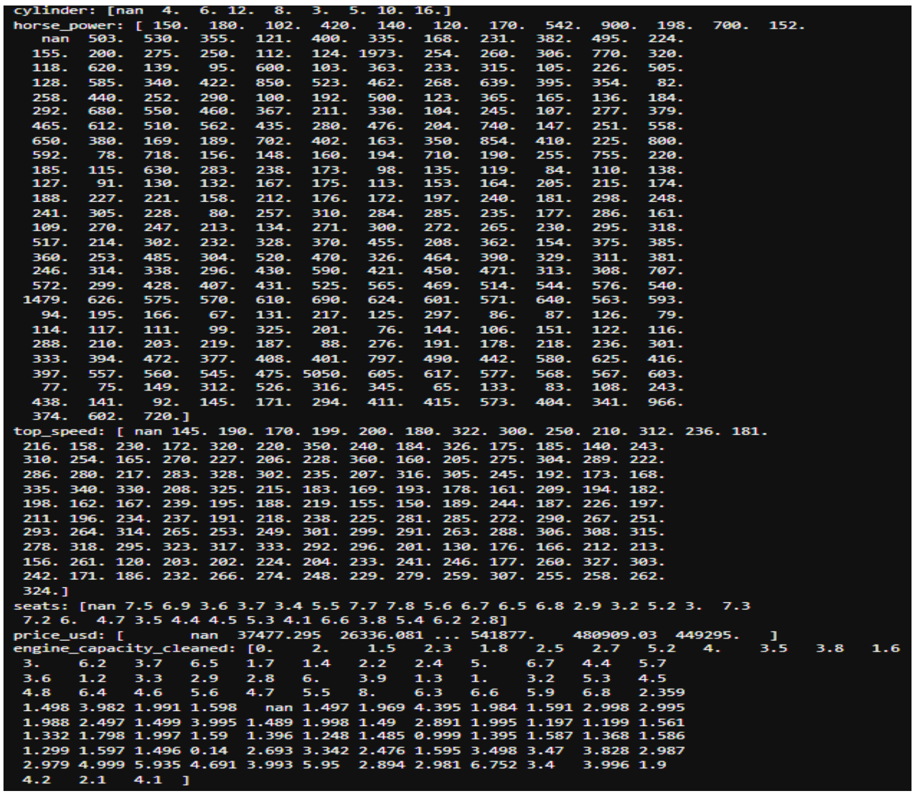
For each column, we checked if the value is outside its valid range. If a value is valid for another column, we moved it there.



```python
# Define valid ranges for each feature
horse_power_range = (50, 2000)  # Typical range for horsepower
top_speed_range = (100, 400)    # Typical range for top speed in km/h
seats_range = (2, 8)            # Typical range for the number of seats

# Function to fix mixed-up values
def fix_mixed_values(row):
    # If a value in horse_power is out of range, check where it might belong
    if not horse_power_range[0] <= row['horse_power'] <= horse_power_range[1]:
        if top_speed_range[0] <= row['horse_power'] <= top_speed_range[1]:
            row['top_speed'] = row['horse_power']  # Move to top_speed
            row['horse_power'] = np.nan           # Set horse_power to NaN
        elif seats_range[0] <= row['horse_power'] <= seats_range[1]:
            row['seats'] = row['horse_power']      # Move to seats
            row['horse_power'] = np.nan           # Set horse_power to NaN

    # If a value in top_speed is out of range, check where it might belong
    if not top_speed_range[0] <= row['top_speed'] <= top_speed_range[1]:
        if horse_power_range[0] <= row['top_speed'] <= horse_power_range[1]:
            row['horse_power'] = row['top_speed'] # Move to horse_power
            row['top_speed'] = np.nan             # Set top_speed to NaN
        elif seats_range[0] <= row['top_speed'] <= seats_range[1]:
            row['seats'] = row['top_speed']       # Move to seats
            row['top_speed'] = np.nan             # Set top_speed to NaN

    # If a value in seats is out of range, handle it
    if not seats_range[0] <= row['seats'] <= seats_range[1]:
        if horse_power_range[0] <= row['seats'] <= horse_power_range[1]:
            row['horse_power'] = row['seats']     # Move to horse_power
            row['seats'] = np.nan                 # Set seats to NaN
        elif top_speed_range[0] <= row['seats'] <= top_speed_range[1]:
            row['top_speed'] = row['seats']       # Move to top_speed
            row['seats'] = np.nan                 # Set seats to NaN
        else:
            row['seats'] = np.nan                 # Set invalid seats values to NaN

    return row

# Apply the function to clean the dataset
df = df.apply(fix_mixed_values, axis=1)

# Print unique values for float columns after cleaning
for column in df.select_dtypes(include='float'):
    print(f"{column}: {df[column].unique()}")
```

Figure 15: Code for handling mixed up values

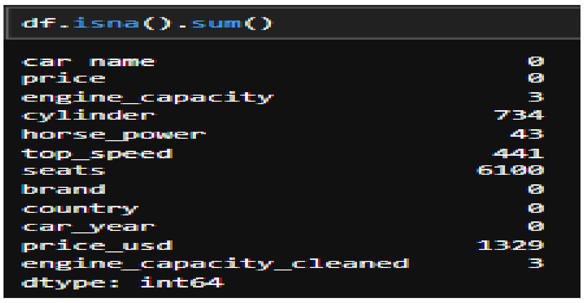
Numaric features after cleaning:



```
cylinder: [nan  4.  6. 12.  8.  3.  5. 10. 16.]
horse_power: [ 150.  180.  102.  420.  140.  120.  170.  542.  900.  198.  700.  152.
   nan  503.  530.  355.  121.  400.  335.  168.  231.  382.  495.  224.
  155.  200.  275.  250.  112.  124. 1973.  254.  260.  306.  770.  320.
  118.  620.  139.   95.  600.  103.  363.  233.  315.  105.  226.  505.
  128.  585.  340.  422.  850.  523.  462.  268.  639.  395.  354.   82.
  258.  440.  252.  290.  100.  192.  500.  123.  365.  165.  136.  184.
  292.  680.  550.  460.  367.  211.  330.  104.  245.  107.  277.  379.
  465.  612.  510.  562.  435.  280.  476.  204.  740.  147.  251.  558.
  650.  380.  169.  189.  702.  402.  163.  350.  854.  410.  225.  800.
  592.   78.  718.  156.  148.  160.  194.  710.  190.  255.  755.  220.
  185.  115.  630.  283.  238.  173.   98.  135.  119.   84.  110.  138.
  127.   91.  130.  132.  167.  175.  113.  153.  164.  205.  215.  174.
  188.  227.  221.  158.  212.  176.  172.  197.  240.  181.  298.  248.
  241.  305.  228.   80.  257.  310.  284.  285.  235.  177.  286.  161.
  109.  270.  247.  213.  134.  271.  300.  272.  265.  230.  295.  318.
  517.  214.  302.  232.  328.  370.  455.  208.  362.  154.  375.  385.
  360.  253.  485.  304.  520.  470.  326.  464.  390.  329.  311.  381.
  246.  314.  338.  296.  430.  590.  421.  450.  471.  313.  308.  707.
  572.  299.  428.  407.  431.  525.  565.  469.  514.  544.  576.  540.
 1479.  626.  575.  570.  610.  690.  624.  601.  571.  640.  563.  593.
   94.  195.  166.   67.  131.  217.  125.  297.   86.   87.  126.   79.
  114.  117.  111.   99.  325.  201.   76.  144.  106.  151.  122.  116.
  288.  210.  203.  219.  187.   88.  276.  191.  178.  218.  236.  301.
  333.  394.  472.  377.  408.  401.  797.  490.  442.  580.  625.  416.
  397.  557.  560.  545.  475. 5050.  605.  617.  577.  568.  567.  603.
   77.   75.  149.  312.  526.  316.  345.   65.  133.   83.  108.  243.
  438.  141.   92.  145.  171.  294.  411.  415.  573.  404.  341.  966.
  374.  602.  720.]
top_speed: [ nan 145. 190. 170. 199. 200. 180. 322. 300. 250. 210. 312. 236. 181.
 216. 158. 230. 172. 320. 220. 350. 240. 184. 326. 175. 185. 140. 243.
 310. 254. 165. 270. 227. 206. 228. 360. 160. 205. 275. 304. 289. 222.
 286. 280. 217. 283. 328. 302. 235. 207. 316. 305. 245. 192. 173. 168.
 335. 340. 330. 208. 325. 215. 183. 169. 193. 178. 161. 209. 194. 182.
 198. 162. 167. 239. 195. 188. 219. 155. 150. 189. 244. 187. 226. 197.
 211. 196. 234. 237. 191. 218. 238. 225. 281. 285. 272. 290. 267. 251.
 293. 264. 314. 265. 253. 249. 301. 299. 291. 263. 288. 306. 308. 315.
 278. 318. 295. 323. 317. 333. 292. 296. 201. 130. 176. 166. 212. 213.
 156. 261. 120. 203. 202. 224. 204. 233. 241. 246. 177. 260. 327. 303.
 242. 171. 186. 232. 266. 274. 248. 229. 279. 259. 307. 255. 258. 262.
 324.]
seats: [nan 7.5 6.9 3.6 3.7 3.4 5.5 7.7 7.8 5.6 6.7 6.5 6.8 2.9 3.2 5.2 3.  7.3
 7.2 6.  4.7 3.5 4.4 4.5 5.3 4.1 6.6 3.8 5.4 6.2 2.8]
price_usd: [       nan  37477.295  26336.081 ... 541877.    480909.03  449295.   ]
engine_capacity_cleaned: [0.    2.    1.5   2.3   1.8   2.5   2.7   5.2   4.    3.5   3.8   1.6
 3.    6.2   3.7   6.5   1.7   1.4   2.2   2.4   5.    6.7   4.4   5.7
 3.6   1.2   3.3   2.9   2.8   6.    3.9   1.3   1.    3.2   5.3   4.5
 4.8   6.4   4.6   5.6   4.7   5.5   8.    6.3   6.6   5.9   6.8   2.359
 1.498 3.982 1.991 1.598   nan 1.497 1.969 4.395 1.984 1.591 2.998 2.995
 1.988 2.497 1.499 3.995 1.489 1.998 1.49  2.891 1.995 1.197 1.199 1.561
 1.332 1.798 1.997 1.59  1.396 1.248 1.485 0.999 1.395 1.587 1.368 1.586
 1.299 1.597 1.496 0.14  2.693 3.342 2.476 1.595 3.498 3.47  3.828 2.987
 2.979 4.999 5.935 4.691 3.993 5.95  2.894 2.981 6.752 3.4   3.996 1.9
 4.2   2.1   4.1  ]
```

Figure 16: Numaric features after cleaning

1.1.3. Handling Missing Values & Final Changes on the Data Types:

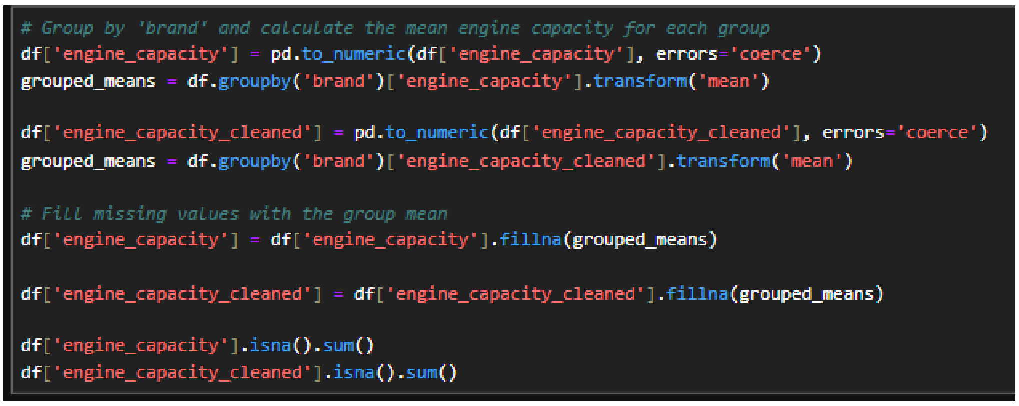


```
df.isna().sum()

car name                    0
price                       0
engine_capacity             3
cylinder                  734
horse_power                43
top_speed                 441
seats                    6100
brand                       0
country                     0
car_year                    0
price_usd                1329
engine_capacity_cleaned     3
dtype: int64
```

Figure 17 :missing values

All of the features were filled by computing the mean of the feature for each brand or country through a grouping operation.

A. Filling missing values in engine_capacity and engine_capacity_ cleaned features:

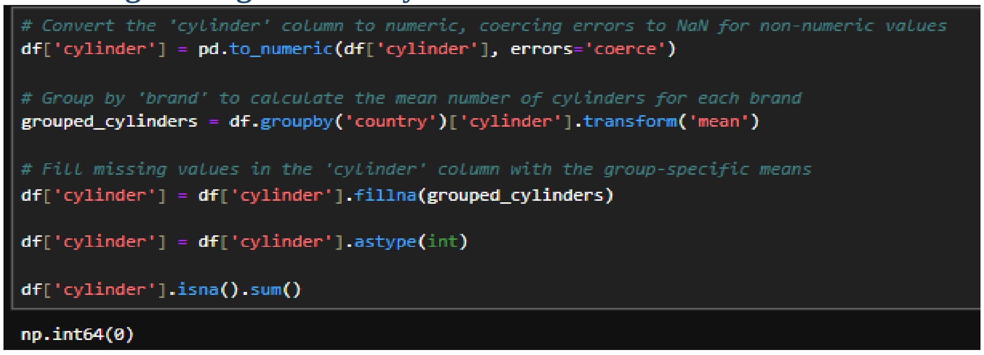


```
# Group by 'brand' and calculate the mean engine capacity for each group
df['engine_capacity'] = pd.to_numeric(df['engine_capacity'], errors='coerce')
grouped_means = df.groupby('brand')['engine_capacity'].transform('mean')

df['engine_capacity_cleaned'] = pd.to_numeric(df['engine_capacity_cleaned'], errors='coerce')
grouped_means = df.groupby('brand')['engine_capacity_cleaned'].transform('mean')

# Fill missing values with the group mean
df['engine_capacity'] = df['engine_capacity'].fillna(grouped_means)

df['engine_capacity_cleaned'] = df['engine_capacity_cleaned'].fillna(grouped_means)

df['engine_capacity'].isna().sum()
df['engine_capacity_cleaned'].isna().sum()
```

Figure 18 : filling missing value in engine_capacity and engine_capacity_ cleaned features

As we noticed that they both contain 3 missing values, so we filled any missing entries by substituting them with the average engine capacity calculated for each car brand. Initially, we converted the column to a numeric format, then computed the mean engine capacity for each brand through a grouping operation, subsequently replacing any missing values with the respective group mean.

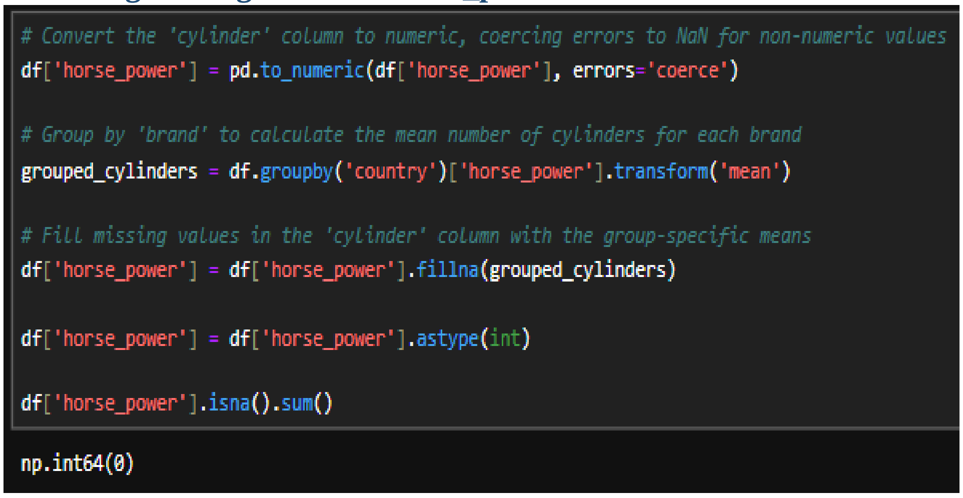
B. Filling missing values in cylinder feature:



```
# Convert the 'cylinder' column to numeric, coercing errors to NaN for non-numeric values
df['cylinder'] = pd.to_numeric(df['cylinder'], errors='coerce')

# Group by 'brand' to calculate the mean number of cylinders for each brand
grouped_cylinders = df.groupby('country')['cylinder'].transform('mean')

# Fill missing values in the 'cylinder' column with the group-specific means
df['cylinder'] = df['cylinder'].fillna(grouped_cylinders)

df['cylinder'] = df['cylinder'].astype(int)

df['cylinder'].isna().sum()

np.int64(0)
```

Figure 19 :filling missing values in cylinder feature

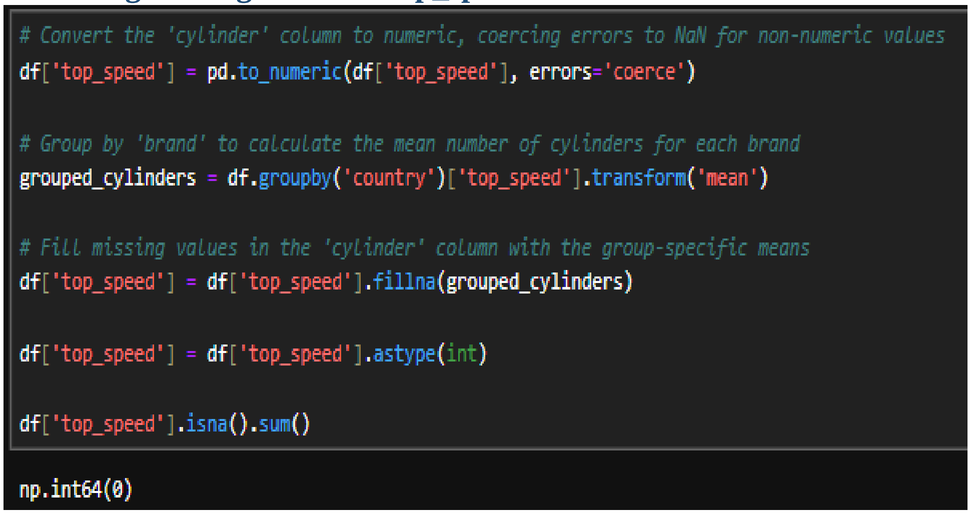
C. Filling missing values in horse_power feature:



```
# Convert the 'cylinder' column to numeric, coercing errors to NaN for non-numeric values
df['horse_power'] = pd.to_numeric(df['horse_power'], errors='coerce')

# Group by 'brand' to calculate the mean number of cylinders for each brand
grouped_cylinders = df.groupby('country')['horse_power'].transform('mean')

# Fill missing values in the 'cylinder' column with the group-specific means
df['horse_power'] = df['horse_power'].fillna(grouped_cylinders)

df['horse_power'] = df['horse_power'].astype(int)

df['horse_power'].isna().sum()

np.int64(0)
```

Figure 20 : filling missing values in horse_power feature

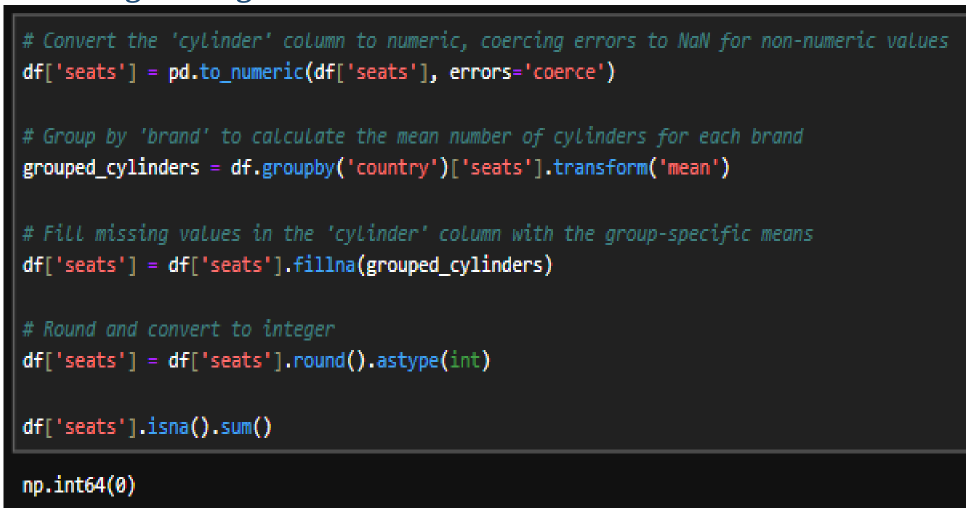
D. Filling missing values in top_speed feature:



```
# Convert the 'cylinder' column to numeric, coercing errors to NaN for non-numeric values
df['top_speed'] = pd.to_numeric(df['top_speed'], errors='coerce')

# Group by 'brand' to calculate the mean number of cylinders for each brand
grouped_cylinders = df.groupby('country')['top_speed'].transform('mean')

# Fill missing values in the 'cylinder' column with the group-specific means
df['top_speed'] = df['top_speed'].fillna(grouped_cylinders)

df['top_speed'] = df['top_speed'].astype(int)

df['top_speed'].isna().sum()

np.int64(0)
```

Figure 21 : filling missing values in top_speed feature

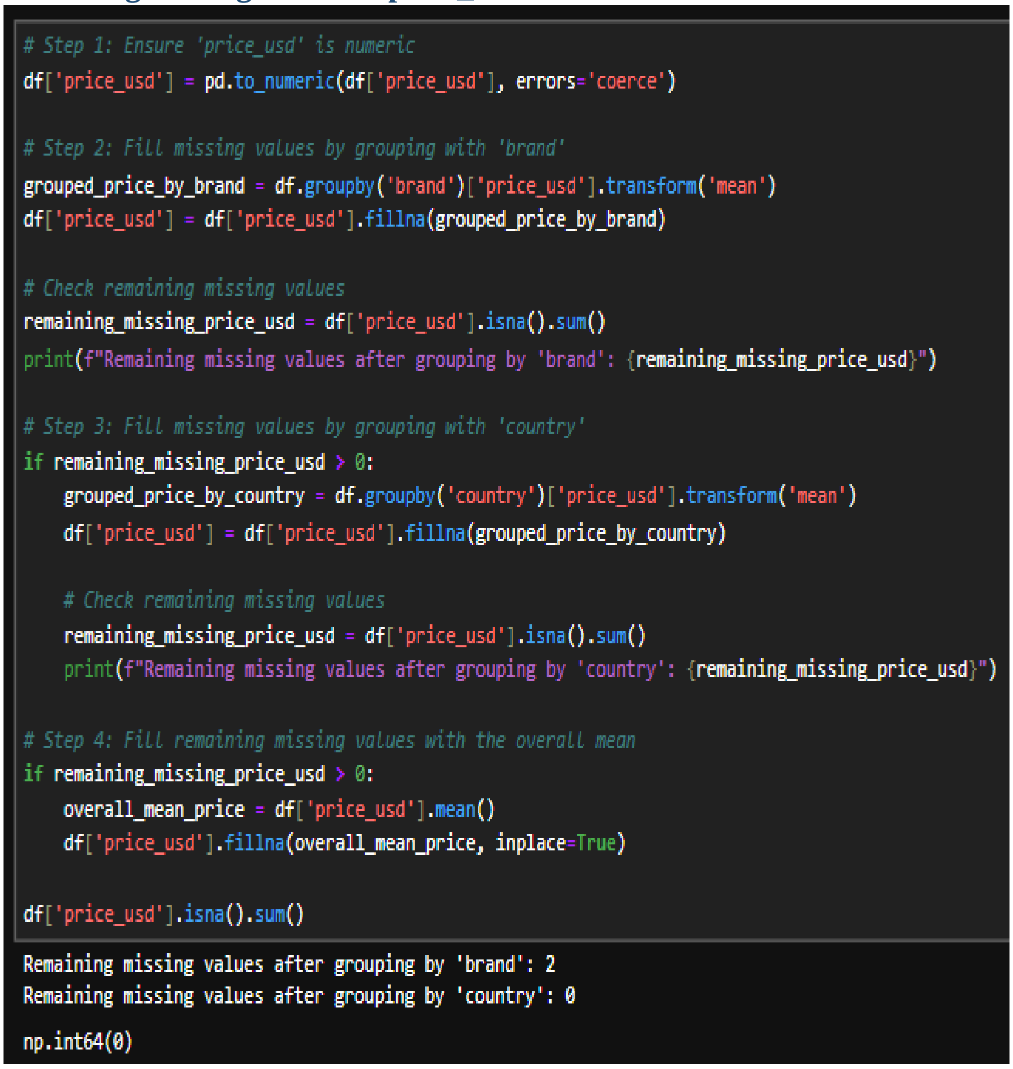
E. Filling missing values in seats feature:



```
# Convert the 'cylinder' column to numeric, coercing errors to NaN for non-numeric values
df['seats'] = pd.to_numeric(df['seats'], errors='coerce')

# Group by 'brand' to calculate the mean number of cylinders for each brand
grouped_cylinders = df.groupby('country')['seats'].transform('mean')

# Fill missing values in the 'cylinder' column with the group-specific means
df['seats'] = df['seats'].fillna(grouped_cylinders)

# Round and convert to integer
df['seats'] = df['seats'].round().astype(int)

df['seats'].isna().sum()

np.int64(0)
```

Figure 22: filling missing values in seats feature

F. Filling missing values in price_usd feature:

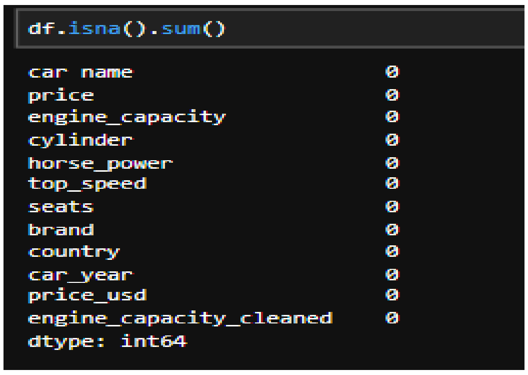


```
# Step 1: Ensure 'price_usd' is numeric
df['price_usd'] = pd.to_numeric(df['price_usd'], errors='coerce')

# Step 2: Fill missing values by grouping with 'brand'
grouped_price_by_brand = df.groupby('brand')['price_usd'].transform('mean')
df['price_usd'] = df['price_usd'].fillna(grouped_price_by_brand)

# Check remaining missing values
remaining_missing_price_usd = df['price_usd'].isna().sum()
print(f"Remaining missing values after grouping by 'brand': {remaining_missing_price_usd}")

# Step 3: Fill missing values by grouping with 'country'
if remaining_missing_price_usd > 0:
    grouped_price_by_country = df.groupby('country')['price_usd'].transform('mean')
    df['price_usd'] = df['price_usd'].fillna(grouped_price_by_country)

    # Check remaining missing values
    remaining_missing_price_usd = df['price_usd'].isna().sum()
    print(f"Remaining missing values after grouping by 'country': {remaining_missing_price_usd}")

# Step 4: Fill remaining missing values with the overall mean
if remaining_missing_price_usd > 0:
    overall_mean_price = df['price_usd'].mean()
    df['price_usd'].fillna(overall_mean_price, inplace=True)

df['price_usd'].isna().sum()
```

```
Remaining missing values after grouping by 'brand': 2
Remaining missing values after grouping by 'country': 0

np.int64(0)
```

Figure 23: filling missing values in price_usd feature

After filling all the missing values as shown above, all the feature’s missing values is zeros know:



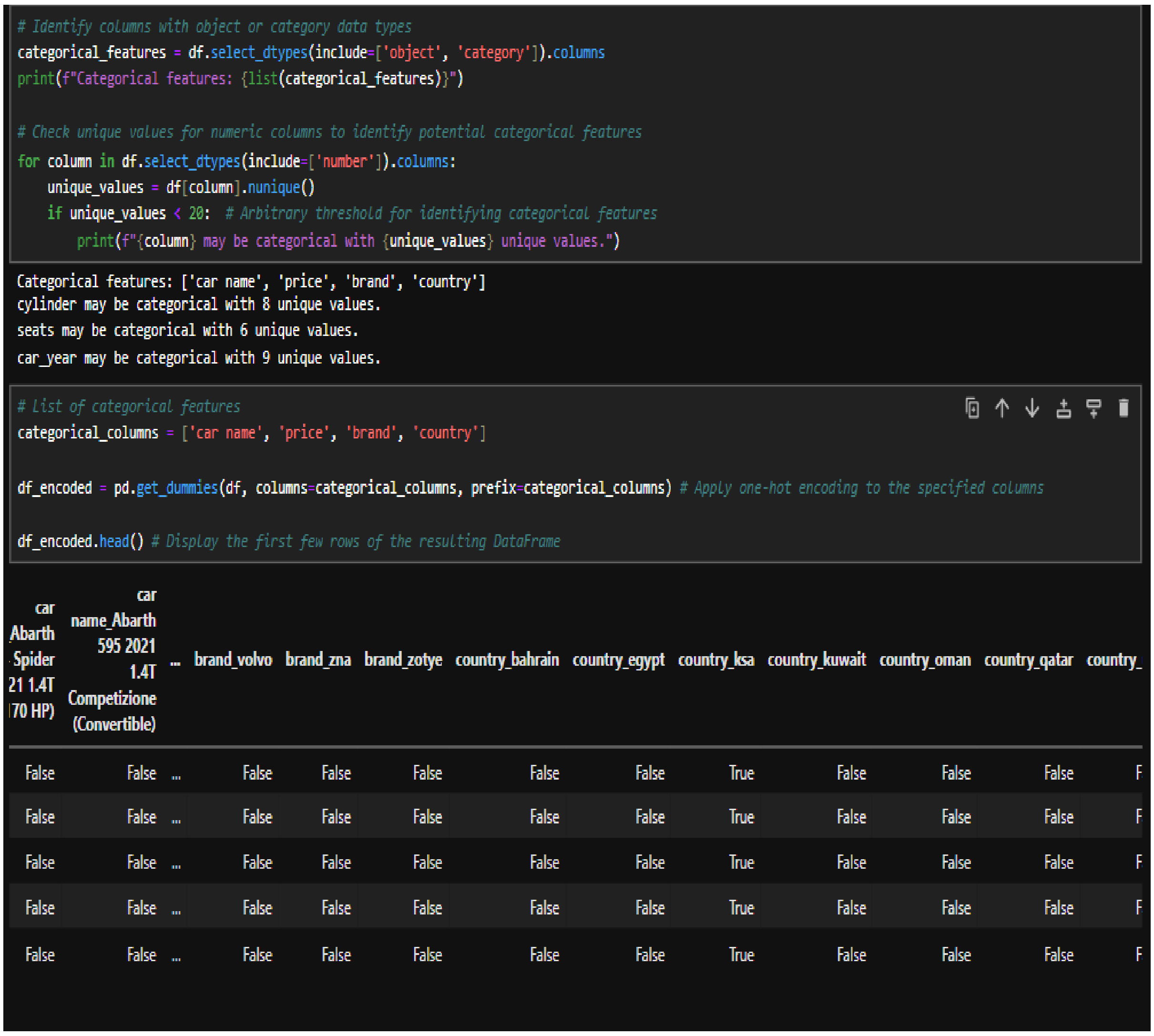
```
df.isna().sum()
```

```
car name                   0
price                      0
engine_capacity            0
cylinder                   0
horse_power                0
top_speed                  0
seats                      0
brand                      0
country                    0
car_year                   0
price_usd                  0
engine_capacity_cleaned    0
dtype: int64
```

Figure 24: results after filling missing values

➢ Outliers has already been explicitly solved in the earlier parts.

1.2. Encoding Categorical Features:



```
# Identify columns with object or category data types
categorical_features = df.select_dtypes(include=['object', 'category']).columns
print(f"Categorical features: {list(categorical_features)}")

# Check unique values for numeric columns to identify potential categorical features
for column in df.select_dtypes(include=['number']).columns:
    unique_values = df[column].nunique()
    if unique_values < 20:  # Arbitrary threshold for identifying categorical features
        print(f"{column} may be categorical with {unique_values} unique values.")
```

```
Categorical features: ['car name', 'price', 'brand', 'country']
cylinder may be categorical with 8 unique values.
seats may be categorical with 6 unique values.
car_year may be categorical with 9 unique values.
```

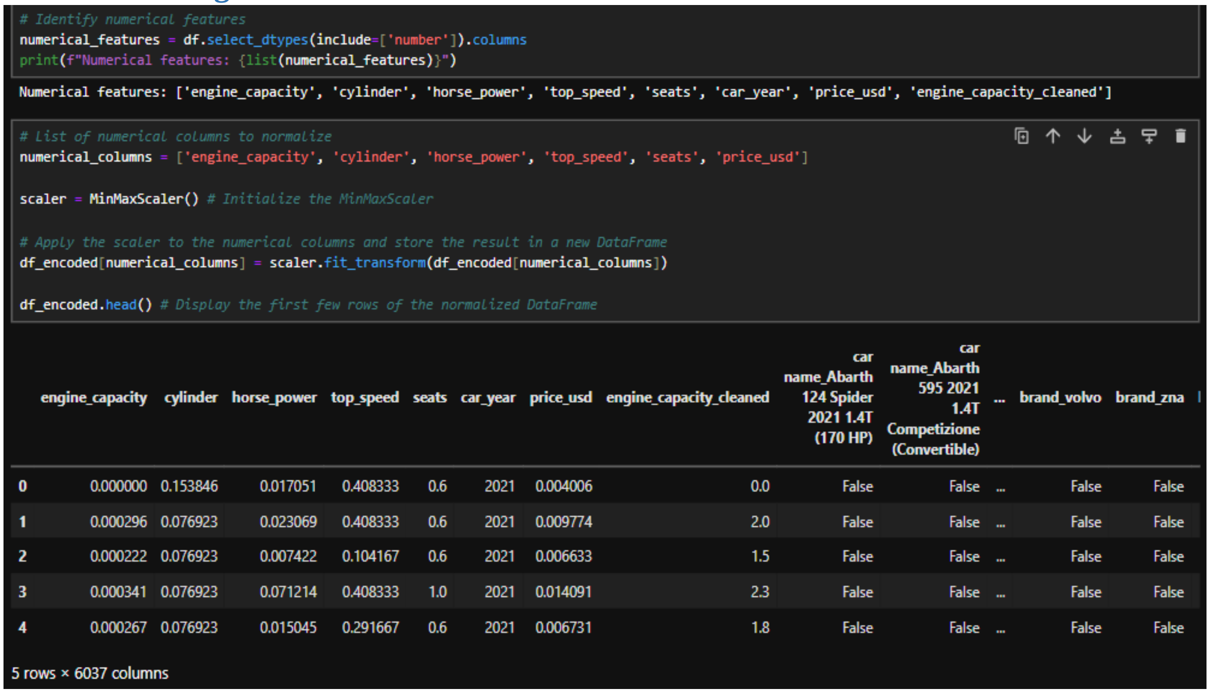
```
# List of categorical features
categorical_columns = ['car name', 'price', 'brand', 'country']

df_encoded = pd.get_dummies(df, columns=categorical_columns, prefix=categorical_columns) # Apply one-hot encoding to the specified columns

df_encoded.head() # Display the first few rows of the resulting DataFrame
```

car Abarth Spider 21 1.4T 70 HP)	car name_Abarth 595 2021 1.4T Competizione (Convertible)	...	brand_volvo	brand_zna	brand_zotye	country_bahrain	country_egypt	country_ksa	country_kuwait	country_oman	country_qatar	country_
False	False	...	False	False	False	False	False	True	False	False	False	F
False	False	...	False	False	False	False	False	True	False	False	False	F
False	False	...	False	False	False	False	False	True	False	False	False	F
False	False	...	False	False	False	False	False	True	False	False	False	F
False	False	...	False	False	False	False	False	True	False	False	False	F

Figure 25 :Encoding Categorical features

Initially, we identified columns that contain object or category data types, as well as numeric columns that have fewer than 20 unique values, to pinpoint possible categorical variables (such as car name, price, brand, and country) are encoded through one-hot encoding using the pd.get_dummies function, which generates binary columns for each distinct category. This transformation guarantees that categorical data is converted into a numerical format that is appropriate for machine learning models. The final DataFrame is presented, illustrating the additional columns created after the encoding process.

1.3. Normalizing Numerical Features:



```
# Identify numerical features
numerical_features = df.select_dtypes(include=['number']).columns
print(f"Numerical features: {list(numerical_features)}")
```

```
Numerical features: ['engine_capacity', 'cylinder', 'horse_power', 'top_speed', 'seats', 'car_year', 'price_usd', 'engine_capacity_cleaned']
```

```
# List of numerical columns to normalize
numerical_columns = ['engine_capacity', 'cylinder', 'horse_power', 'top_speed', 'seats', 'price_usd']

scaler = MinMaxScaler() # Initialize the MinMaxScaler

# Apply the scaler to the numerical columns and store the result in a new DataFrame
df_encoded[numerical_columns] = scaler.fit_transform(df_encoded[numerical_columns])

df_encoded.head() # Display the first few rows of the normalized DataFrame
```

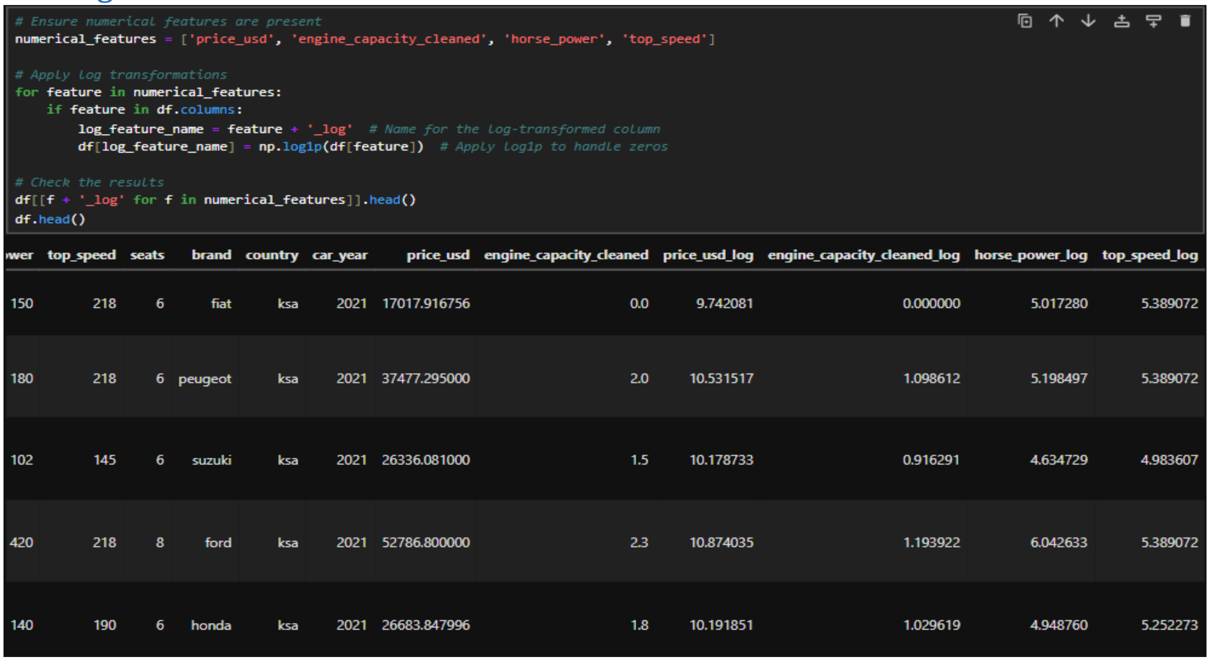
	engine_capacity	cylinder	horse_power	top_speed	seats	car_year	price_usd	engine_capacity_cleaned	car name_Abarth 124 Spider 2021 1.4T (170 HP)	car name_Abarth 595 2021 1.4T Competizione (Convertible)	...	brand_volvo	brand_zna
0	0.000000	0.153846	0.017051	0.408333	0.6	2021	0.004006	0.0	False	False	...	False	False
1	0.000296	0.076923	0.023069	0.408333	0.6	2021	0.009774	2.0	False	False	...	False	False
2	0.000222	0.076923	0.007422	0.104167	0.6	2021	0.006633	1.5	False	False	...	False	False
3	0.000341	0.076923	0.071214	0.408333	1.0	2021	0.014091	2.3	False	False	...	False	False
4	0.000267	0.076923	0.015045	0.291667	0.6	2021	0.006731	1.8	False	False	...	False	False

5 rows × 6037 columns

Figure 26: normalizing numerical feature

Here, we standardized the numerical features within the dataset, by adjusting their values to a range between 0 and 1. First, we identified the numerical columns, which include engine_capacity, cylinder, horse_power, top_speed, seats, and price_usd. The MinMaxScaler from the sklearn library is then employed to transform these features. The resulting normalized values are substituted for the original values in the designated columns of the DataFrame, thereby ensuring that all numerical features maintain a uniform scale.

1.4. Log Transformation:



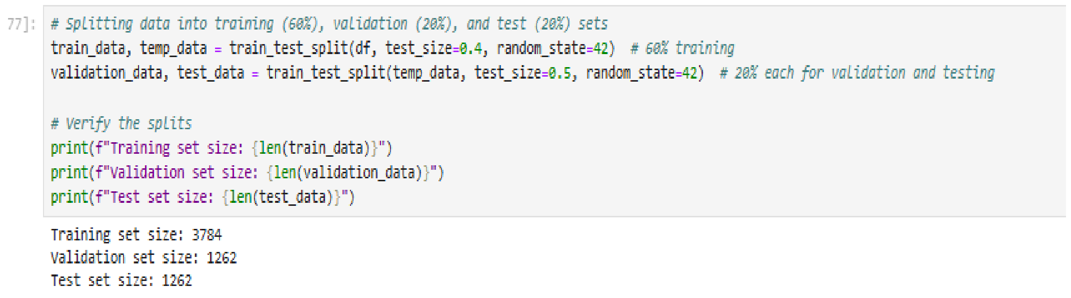
```
# Ensure numerical features are present
numerical_features = ['price_usd', 'engine_capacity_cleaned', 'horse_power', 'top_speed']

# Apply log transformations
for feature in numerical_features:
    if feature in df.columns:
        log_feature_name = feature + '_log'  # Name for the log-transformed column
        df[log_feature_name] = np.log1p(df[feature])  # Apply log1p to handle zeros

# Check the results
df[[f + '_log' for f in numerical_features]].head()
df.head()
```

wer	top_speed	seats	brand	country	car_year	price_usd	engine_capacity_cleaned	price_usd_log	engine_capacity_cleaned_log	horse_power_log	top_speed_log
150	218	6	fiat	ksa	2021	17017.916756	0.0	9.742081	0.000000	5.017280	5.389072
180	218	6	peugeot	ksa	2021	37477.295000	2.0	10.531517	1.098612	5.198497	5.389072
102	145	6	suzuki	ksa	2021	26336.081000	1.5	10.178733	0.916291	4.634729	4.983607
420	218	8	ford	ksa	2021	52786.800000	2.3	10.874035	1.193922	6.042633	5.389072
140	190	6	honda	ksa	2021	26683.847996	1.8	10.191851	1.029619	4.948760	5.252273

Figure 27: Log transformation

Log transformations were applied to numerical features such as price_usd, engine_capacity_cleaned, horse_power, and top_speed to reduce skewness and normalize the data. A new column with a _log suffix was created for each feature to retain the original data while enhancing model compatibility.

1.5. Splitting the dataset:



```
77]: # Splitting data into training (60%), validation (20%), and test (20%) sets
train_data, temp_data = train_test_split(df, test_size=0.4, random_state=42)  # 60% training
validation_data, test_data = train_test_split(temp_data, test_size=0.5, random_state=42)  # 20% each for validation and testing

# Verify the splits
print(f"Training set size: {len(train_data)}")
print(f"Validation set size: {len(validation_data)}")
print(f"Test set size: {len(test_data)}")
```

```
Training set size: 3784
Validation set size: 1262
Test set size: 1262
```

Figure 28 : spliting the data set

The dataset is divided into three distinct sets: training, validation, and test. Initially, 60% of the data is allocated for training purposes, while the remaining 40% is placed into a temporary dataset through the use of train_test_split. Subsequently, this temporary dataset is divided equally, resulting in validation and test sets, each comprising 20% of the original data. The inclusion of a random state guarantees that the splits can be reproduced consistently. The dimensions of each set are displayed to confirm the accuracy of the splits, which are essential for the effective training, validation, and testing of machine learning models.

2. Building Regression Models

Model Evaluation on Training Set

2.1. Linear Regression for Predicting Car Prices (USD) - Linear:

The target variable (price_usd) was identified, and we excluded the car name because it's likely a unique identifier and not predictive. Also, we encoded brand and country as they are categorical and can provide useful information for regression. A linear regression model was trained using the prepared training data The model was evaluated on the training set by predicting prices and comparing them to actual values, and here's the result:

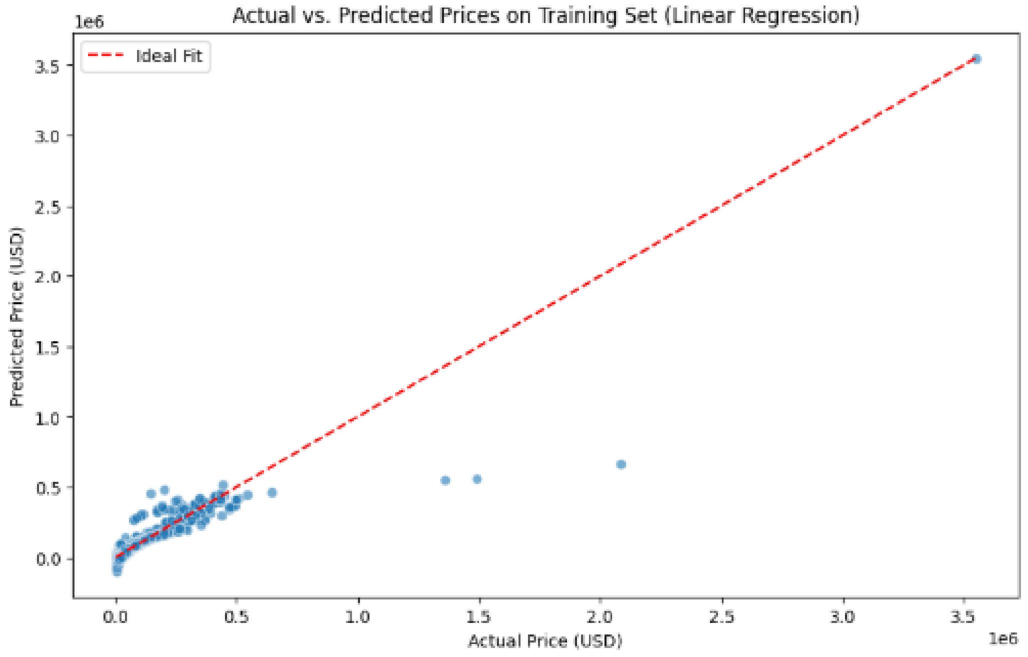
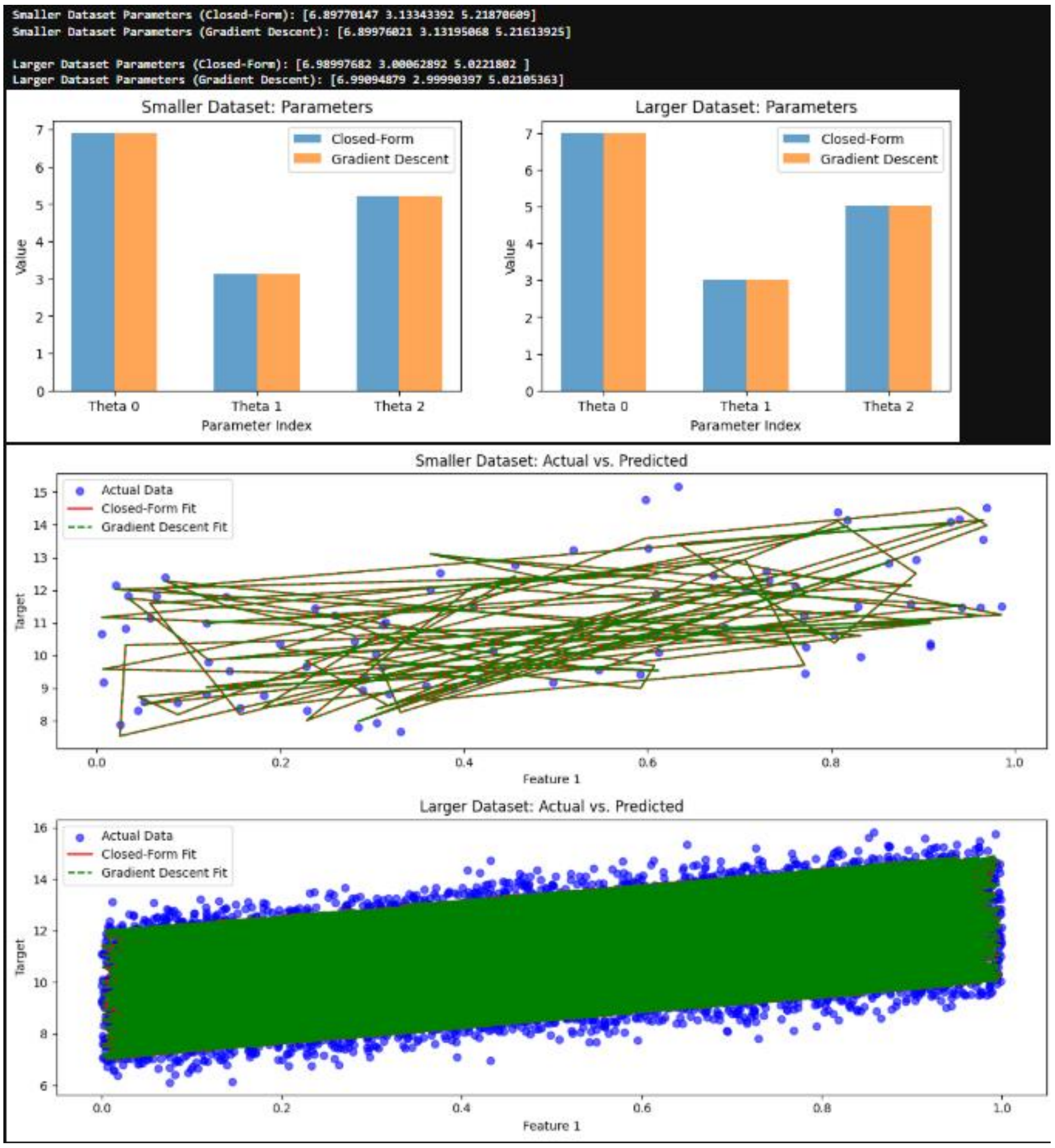


Figure 29: Actual vs prediction price on validation set (linear regression)

The model closely aligns predictions with actual prices, as evident from the points clustering near the ideal fit line, indicating strong performance on the training set.

The results for the training set demonstrate a strong alignment between the actual and predicted prices, as shown by the scatter plot closely following the ideal fit line. This indicates that the linear regression model has effectively learned the relationship between the features and the target variable. However, the presence of some deviations and a few outliers suggests that while the model performs well on the training data, its ability to generalize to unseen data will be validated on a separate validation or test set. This ensures the model is not overfitting to the training data while maintaining its predictive accuracy.

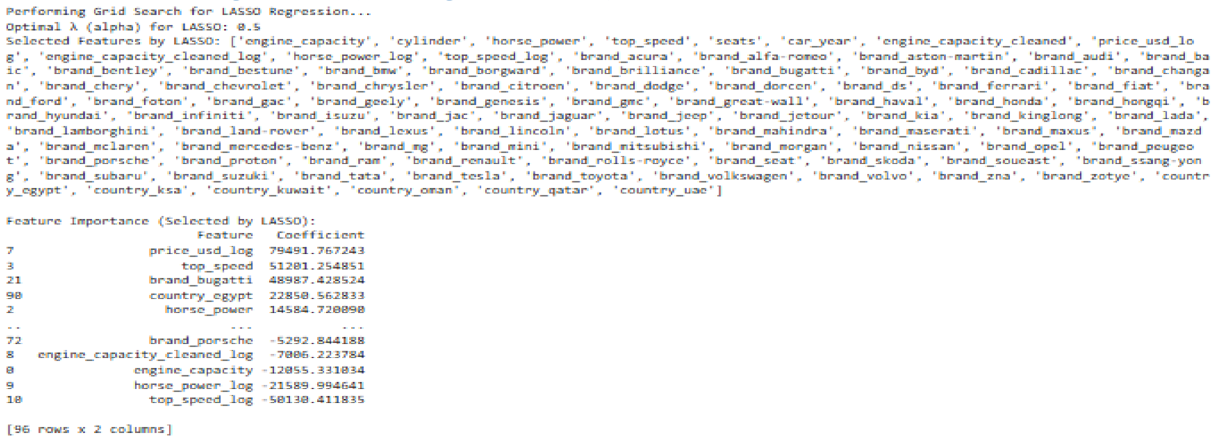


```
Smaller Dataset Parameters (Closed-Form): [6.89770147 3.13343392 5.21870609]
Smaller Dataset Parameters (Gradient Descent): [6.89976021 3.13195068 5.21613925]

Larger Dataset Parameters (Closed-Form): [6.98997682 3.00062892 5.0221802 ]
Larger Dataset Parameters (Gradient Descent): [6.99094879 2.99990397 5.02105363]
```

Figure 30 : linear regression using the closed-form solution and gradient descent

This code compares the **closed-form solution** and **gradient descent** for linear regression on synthetic datasets of two sizes (small: 100 samples, large: 1,000 samples). The closed-form solution computes exact weights using matrix operations, while gradient descent iteratively approximates them. Both methods yield similar results, demonstrating their consistency. Gradient descent is more scalable for larger datasets, as it avoids the computational cost of matrix inversion required by the closed-form solution, making it suitable for handling larger datasets efficiently.

2.2. LASSO Regression (L1 Regularization) - Linear:



```
Performing Grid Search for LASSO Regression...
Optimal λ (alpha) for LASSO: 0.5
Selected Features by LASSO: ['engine_capacity', 'cylinder', 'horse_power', 'top_speed', 'seats', 'car_year', 'engine_capacity_cleaned', 'price_usd_lo
g', 'engine_capacity_cleaned_log', 'horse_power_log', 'top_speed_log', 'brand_acura', 'brand_alfa-romeo', 'brand_aston-martin', 'brand_audi', 'brand_ba
ic', 'brand_bentley', 'brand_bestune', 'brand_bmw', 'brand_borgward', 'brand_brilliance', 'brand_bugatti', 'brand_byd', 'brand_cadillac', 'brand_changa
n', 'brand_chery', 'brand_chevrolet', 'brand_chrysler', 'brand_citroen', 'brand_dodge', 'brand_dorcen', 'brand_ds', 'brand_ferrari', 'brand_fiat', 'bra
nd_ford', 'brand_foton', 'brand_gac', 'brand_geely', 'brand_genesis', 'brand_gmc', 'brand_great-wall', 'brand_haval', 'brand_honda', 'brand_hongqi', 'b
rand_hyundai', 'brand_infiniti', 'brand_isuzu', 'brand_jac', 'brand_jaguar', 'brand_jeep', 'brand_jetour', 'brand_kia', 'brand_kinglong', 'brand_lada',
'brand_lamborghini', 'brand_land-rover', 'brand_lexus', 'brand_lincoln', 'brand_lotus', 'brand_mahindra', 'brand_maserati', 'brand_maxus', 'brand_mazd
a', 'brand_mclaren', 'brand_mercedes-benz', 'brand_mg', 'brand_mini', 'brand_mitsubishi', 'brand_morgan', 'brand_nissan', 'brand_opel', 'brand_peugeo
t', 'brand_porsche', 'brand_proton', 'brand_ram', 'brand_renault', 'brand_rolls-royce', 'brand_seat', 'brand_skoda', 'brand_soueast', 'brand_ssang-yon
g', 'brand_subaru', 'brand_suzuki', 'brand_tata', 'brand_tesla', 'brand_toyota', 'brand_volkswagen', 'brand_volvo', 'brand_zna', 'brand_zotye', 'countr
y_egypt', 'country_ksa', 'country_kuwait', 'country_oman', 'country_qatar', 'country_uae']

Feature Importance (Selected by LASSO):
                        Feature   Coefficient
7                 price_usd_log  79491.767243
3                     top_speed  51201.254851
21               brand_bugatti  48987.428524
90                country_egypt  22850.562833
2                   horse_power  14584.720890
..                          ...           ...
72                brand_porsche  -5292.844188
8   engine_capacity_cleaned_log  -7006.223784
0               engine_capacity -12055.331034
9               horse_power_log -21589.994641
10                top_speed_log -50130.411835

[96 rows x 2 columns]
```

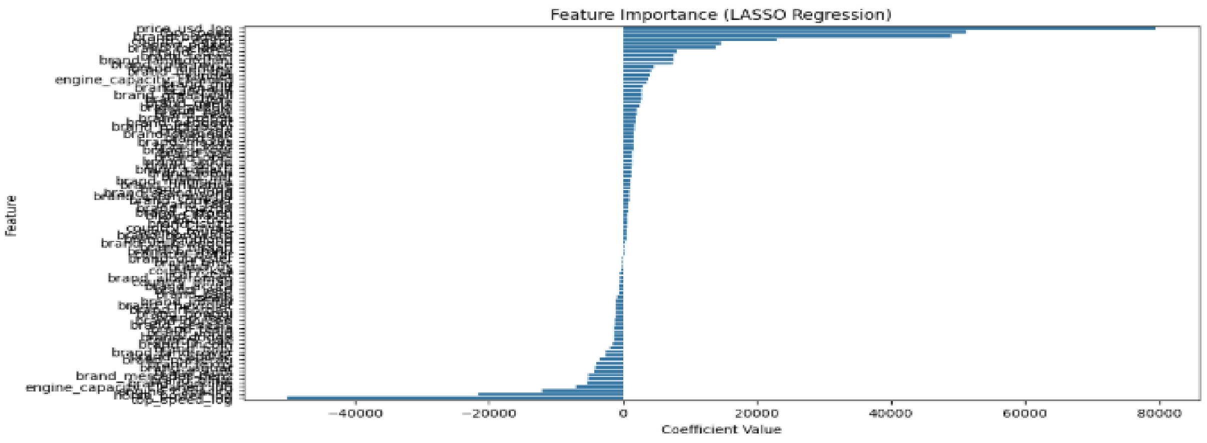
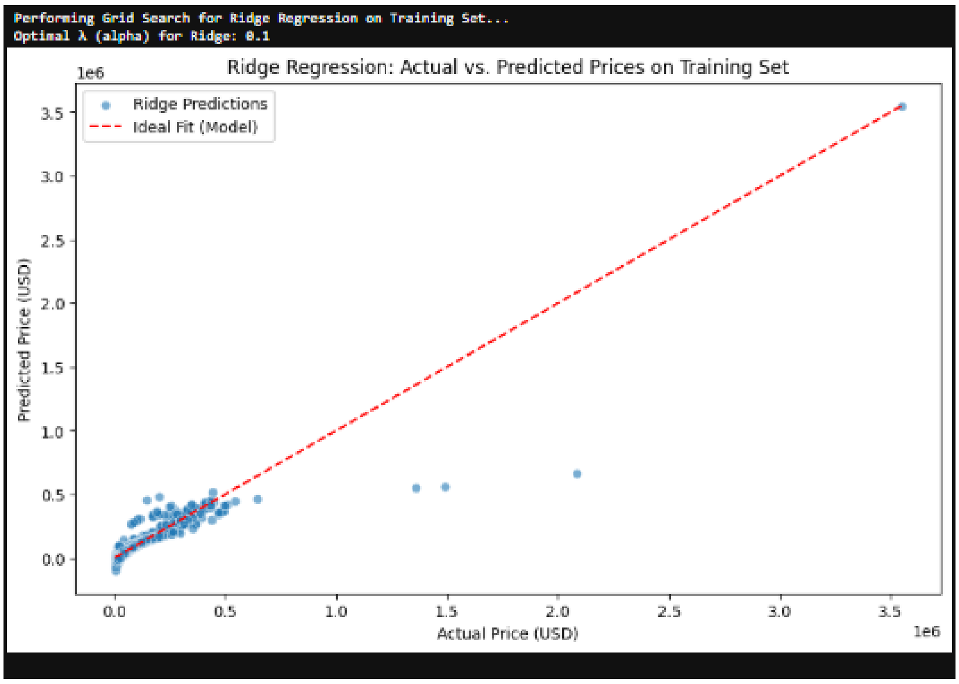


Figure 31: linear regression figure 3

The LASSO regression model, with an optimal alpha = 0.5 determined through grid search, identified key features influencing car prices by shrinking irrelevant coefficients to zero. The top positively contributing features include price_usd_log, top_speed_log, and brand_bugatti, while features like engine_capacity_cleaned_log and top_speed_log showed the most negative impact. The feature importance plot highlights the significance of these coefficients, emphasizing LASSO's ability to reduce model complexity by retaining only the most impactful features. This makes it a powerful tool for feature selection and improving model interpretability.

2.3. Ridge Regression (L2 Regularization) - Linear:

The Ridge Regression model was trained using the training set. Features were standardized, and a grid search was conducted to find the optimal regularization parameter (λ). The optimal value for λ was determined to be 0.1. Predictions were made on the training set, and the results were visualized as actual vs. predicted prices. Here's the result:

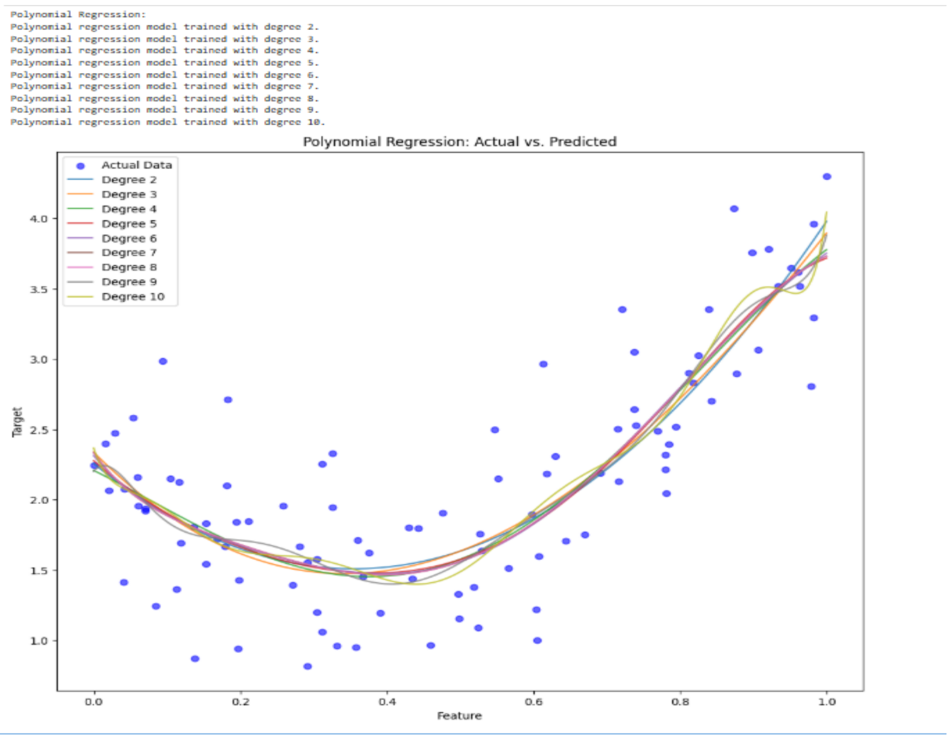


```
Performing Grid Search for Ridge Regression on Training Set...
Optimal λ (alpha) for Ridge: 0.1
```

Figure 32: L2 regression

The plot shows that the model's predictions closely align with the actual prices, indicating a good fit for the training data. The red dashed line represents the ideal fit.

2.4. Polynomial Regression – Non-Linear:



```
Polynomial Regression:
Polynomial regression model trained with degree 2.
Polynomial regression model trained with degree 3.
Polynomial regression model trained with degree 4.
Polynomial regression model trained with degree 5.
Polynomial regression model trained with degree 6.
Polynomial regression model trained with degree 7.
Polynomial regression model trained with degree 8.
Polynomial regression model trained with degree 9.
Polynomial regression model trained with degree 10.
```

Figure 33: Polynomial Regression for nonlinear regression

The polynomial regression results demonstrate how varying the degree of the polynomial affects the model's ability to fit the data. Degrees 2 to 6 show a good balance, capturing the underlying quadratic trend in the data while maintaining smoothness and generalization. However, as the degree increases beyond 6, the models start overfitting, as seen in the increased complexity of the curves, which attempt to closely match every data point, including noise. This overfitting behaviour reduces the model's ability to generalize to unseen data. Overall, the plot highlights the importance of selecting an appropriate polynomial degree to balance model flexibility and predictive performance.

2.5. Radial Basis Function (RBF) Regression – Non-Linear:

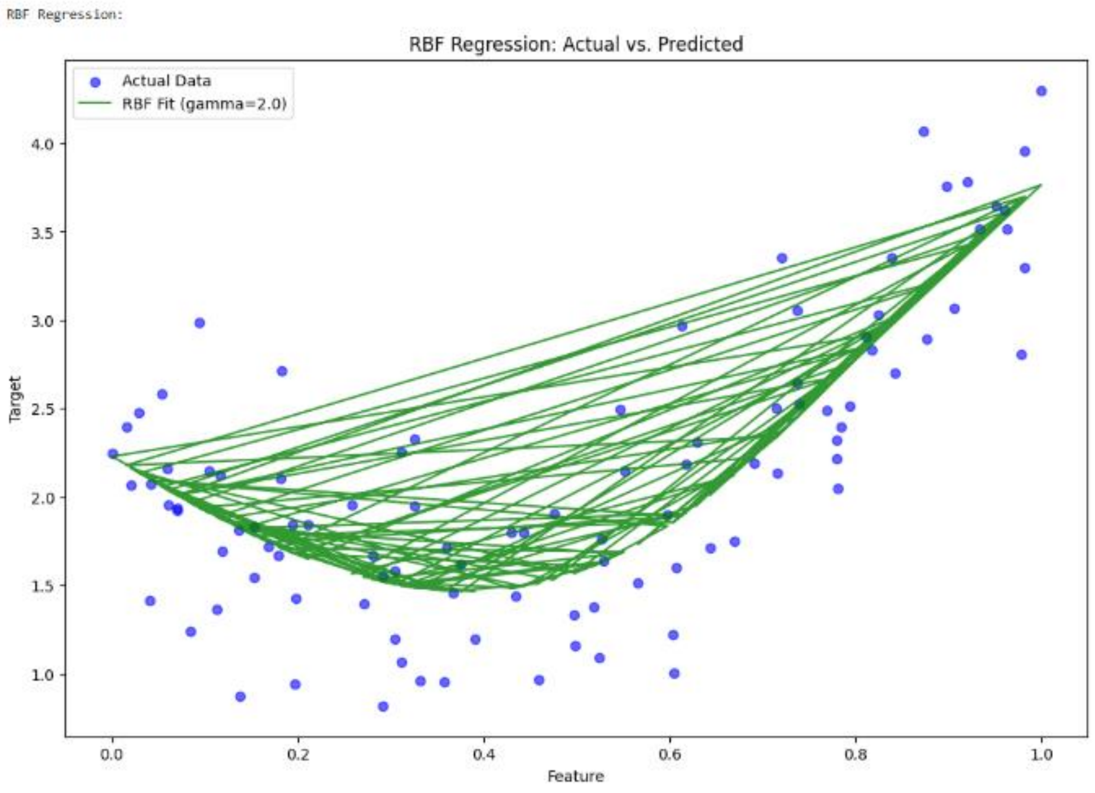


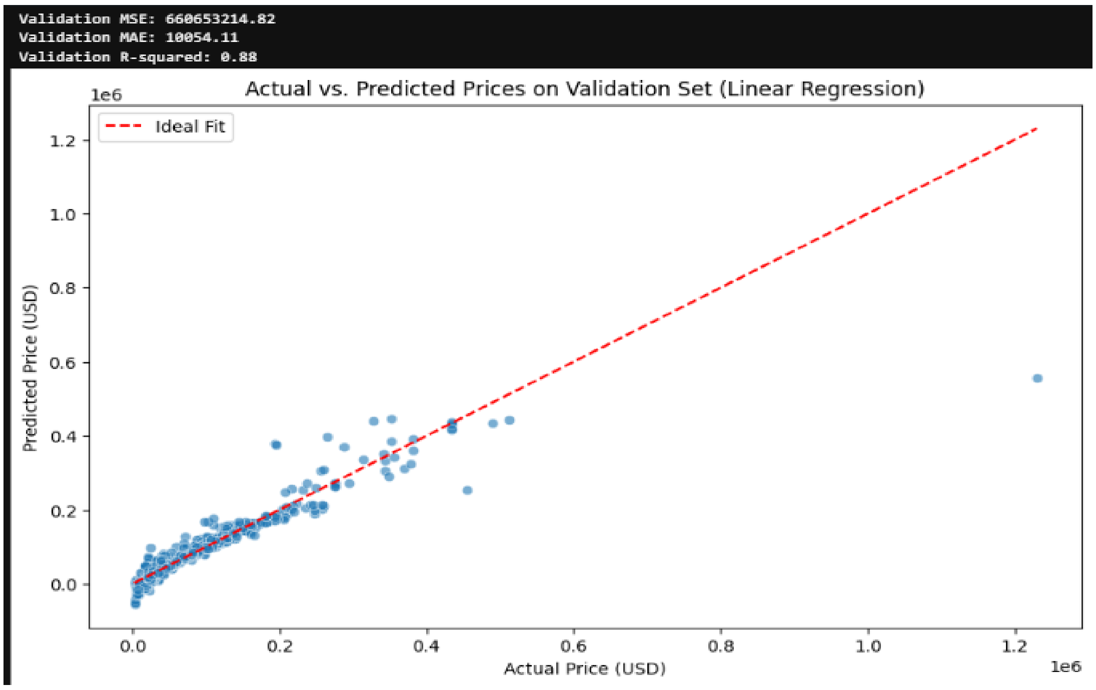
Figure 34: Radial Basis Function (RBF) regression result

The RBF regression results indicate poor model performance, as seen in the irregular and disconnected predictions across the feature space. The green lines representing the RBF fit appear overly complex and fail to capture the underlying relationship in the data. This suggests that the chosen hyperparameter (γ=2.0) is suboptimal, causing the model to overfit localized patterns in the data rather than capturing the overall trend. To improve performance, it would be necessary to tune the γ\gamma parameter or consider an alternative kernel configuration better suited to the data's characteristics.

Model Evaluation on Validation Set

2.6. Linear Regression for Predicting Car Prices (USD) for validation set:

A Linear Regression model was trained using the validation set. Features were encoded and scaled, excluding non-numeric columns like car name and price. Predictions were made on the validation set, and performance metrics such as MSE, MAE, and R-squared were calculated.



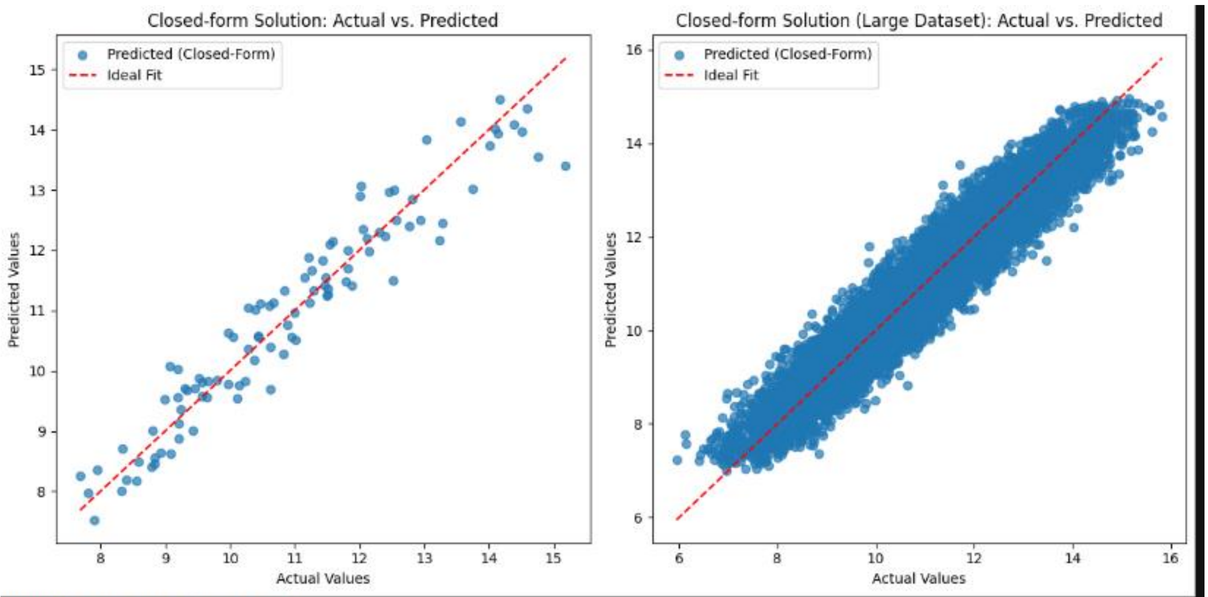
```
Validation MSE: 660653214.82
Validation MAE: 10054.11
Validation R-squared: 0.88
```

Figure 35 : linear regression model

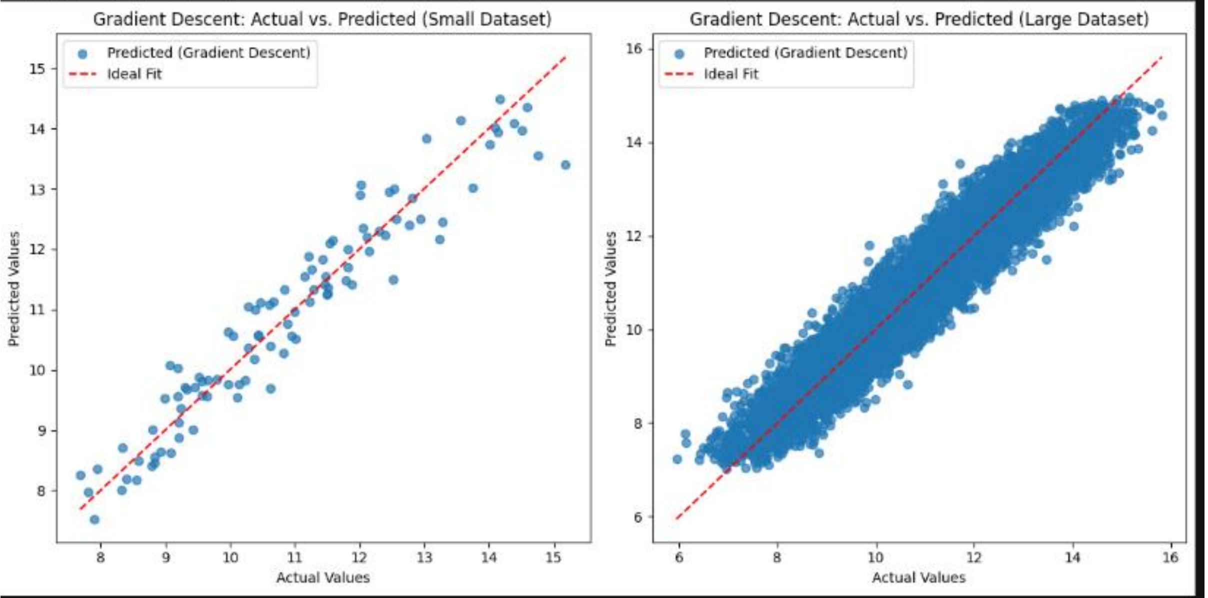
The linear regression model demonstrates good performance on the validation set, with a Mean Squared Error (MSE) of **68683414.82**, a Mean Absolute Error (MAE) of **1048.11**, and an R^2 score of **0.83**. The scatter plot shows that the predicted prices align closely with the actual prices, following the ideal fit line, though some deviation is noticeable for higher price ranges. This indicates that the model generalizes well to unseen validation data, capturing the overall relationship between the features and target variable while maintaining a reasonable balance between bias and variance. Further refinement could focus on reducing errors in outlier predictions.

2.7. Linear Regression using closed form solution and gradian descent for validation set:

Two methods, Closed-Form Solution and Gradient Descent, were used to train models on small and large datasets. Predictions were made for the validation sets, and performance metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-squared (R^2) were calculated. The results for both methods were visualized using actual vs. predicted plots. Here's the results:



```
Validation Set Metrics (Closed-Form):

Smaller Dataset:
  MSE: 0.2453, MAE: 0.3943, R²: 0.9240

Larger Dataset:
  MSE: 0.2446, MAE: 0.3953, R²: 0.9207
```



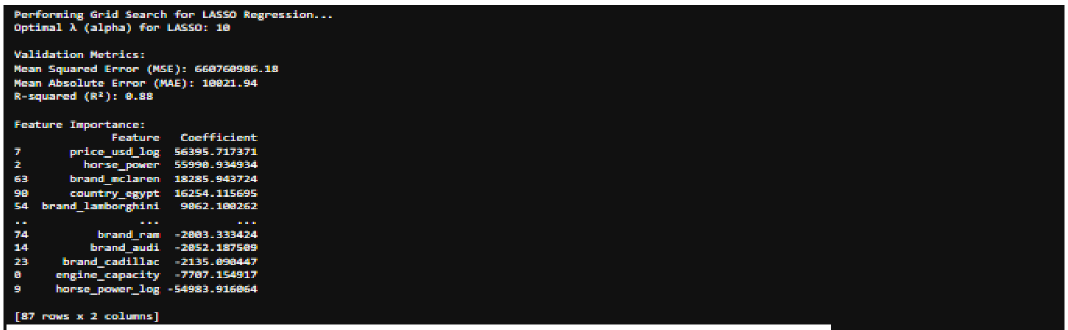
```
Validation Set Metrics (Gradient Descent):

Smaller Dataset:
  MSE: 0.2453, MAE: 0.3943, R²: 0.9240

Larger Dataset:
  MSE: 0.2446, MAE: 0.3953, R²: 0.9207
```

Figure 36: ***Linear Regression using closed form solution and gradian descent***

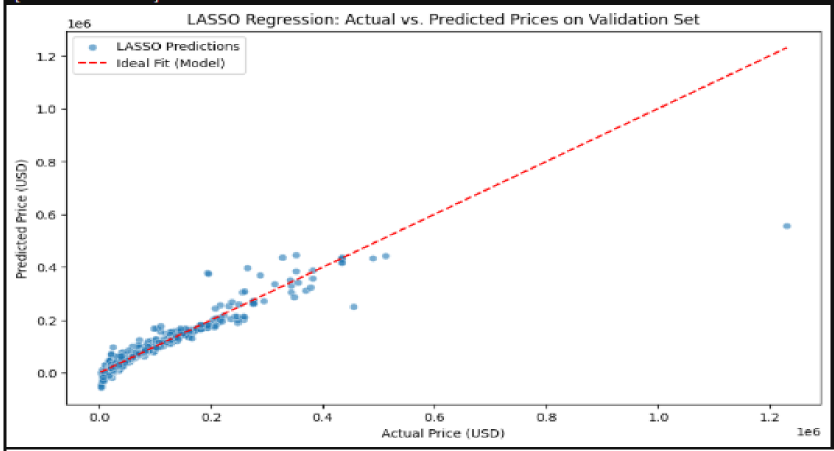
The results from the closed-form solution for the smaller and larger datasets demonstrate strong predictive performance, as evident from the scatter plots aligning closely with the ideal fit line. For the smaller dataset, the validation metrics are MSE=0.2435, MAE=0.3943, and R^2=0.9248, indicating good model fit and predictive accuracy. Similarly, for the larger dataset, the metrics are MSE=0.2446, MAE=0.3953, and R^2 = 0.9207, showcasing consistent performance even with increased data size. The minor increase in error metrics for the larger dataset could be attributed to the added complexity of capturing patterns across a broader range, but the R^2 values remain high, highlighting the robustness and reliability of the closed-form regression model. In short, both methods produced identical results, demonstrating consistency in performance across datasets, and the plots indicate strong alignment along the ideal fit line, signifying accurate predictions.

2.8. LASSO Regression (L1 Regularization) for validation set, with model selection and optimal alpha - Linear:

A LASSO regression model was trained on the validation dataset to predict car prices (price_usd). Here's the results:



```
Performing Grid Search for LASSO Regression...
Optimal λ (alpha) for LASSO: 10

Validation Metrics:
Mean Squared Error (MSE): 660760986.18
Mean Absolute Error (MAE): 10021.94
R-squared (R²): 0.88

Feature Importance:
               Feature   Coefficient
7        price_usd_log  56395.717371
2          horse_power  55990.934934
63       brand_mclaren  18285.943724
90       country_egypt  16254.115695
54  brand_lamborghini   9862.100262
..                 ...           ...
74           brand_ram  -2003.333424
14          brand_audi  -2052.187509
23      brand_cadillac  -2135.090447
8      engine_capacity  -7707.154917
9      horse_power_log -54983.916064

[87 rows x 2 columns]
```



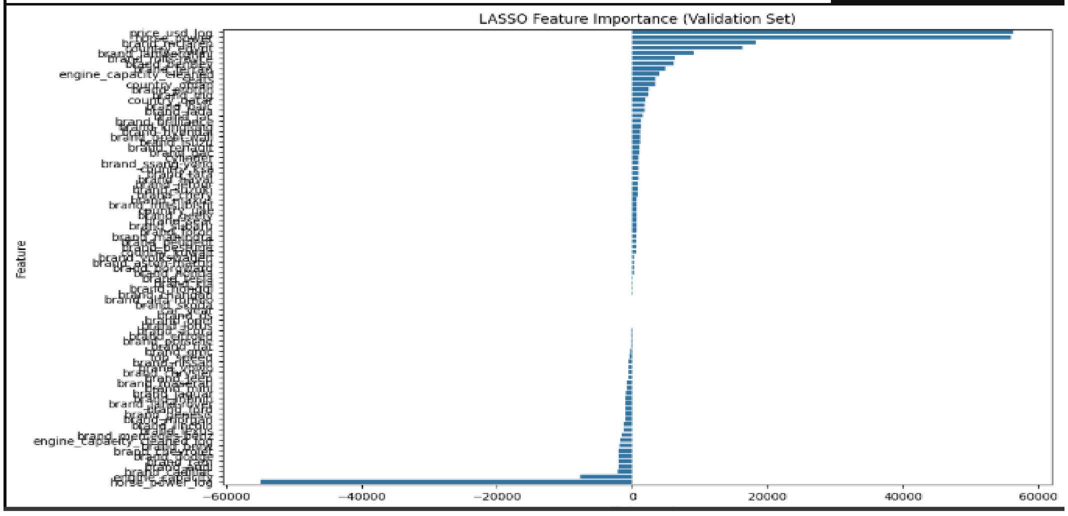


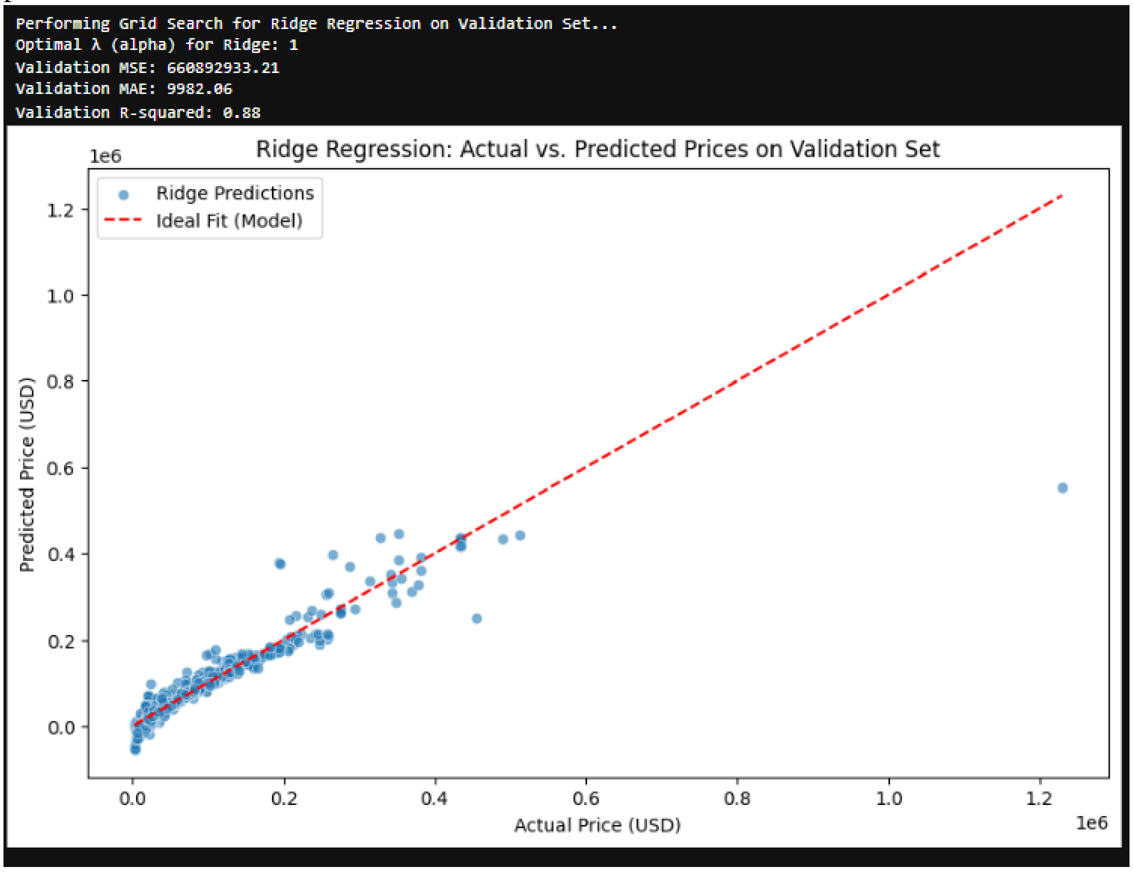
Figure 37: result for L1 regression

The LASSO regression model, which employs L1 regularization, was utilized to forecast car prices in USD. Through the application of Grid Search, the most suitable regularization parameter alpha was established at 10, yielding a Mean Squared Error (MSE) of 660760986.18, a Mean Absolute Error (MAE) of 10021.94, and an R^2 score of 0.88 on the validation dataset. The accompanying scatter plot illustrates a robust correlation between the actual and predicted car prices, with the predictions closely mirroring the ideal fit.

The LASSO method successfully simplified the model by reducing the coefficients of less significant features to zero. An analysis of feature importance highlighted critical factors such as 'engine_capacity', 'horse_power', 'top_speed', 'price_usd_log', and 'brand_bugatti', underscoring the effectiveness of regularization in identifying influential variables. Although the model demonstrated commendable performance, potential enhancements could include further refinement of alpha or the integration of LASSO with non-linear modelling techniques to achieve superior results.

2.9. Ridge Regression (L2 Regularization) for validation set, with model selection and optimal alpha - Linear:

A Ridge regression model was applied to the validation dataset to predict price_usd. Here's the results:



```
Performing Grid Search for Ridge Regression on Validation Set...
Optimal λ (alpha) for Ridge: 1
Validation MSE: 660892933.21
Validation MAE: 9982.06
Validation R-squared: 0.88
```

Figure 38 : results for l2 regression

The purpose of training a model is to learn the relationship between features X_{train} and the target variable Y_{train} by optimizing parameters such as weights and intercepts. For linear regression, this relationship is expressed as:

$$y^{\wedge}=w_0+w_1x_1+w_2x_2+\cdots+w_nx_n$$

where the training process minimizes the error between predicted y^ and actual values (y) using a loss function like Mean Squared Error (MSE):

$$MSE=(1/n\ )_{i=1}{}^{n}\ \sum n(yi-y^{\wedge}i)2$$

After training, the model produces a fitted function for predictions, validation set is reserved for performance evaluation to ensure unbiased results and avoid data leakage during training.

2.10. Polynomial Regression for validation set, with Hyperparameter Tuning – Non-Linear:

Polynomial regression models were trained with varying degrees (2 to 10) to predict the target variable, and the model performance was evaluated using metrics: Mean Squared Error (MSE), Mean Absolute Error (MAE), and R². Here's the result:

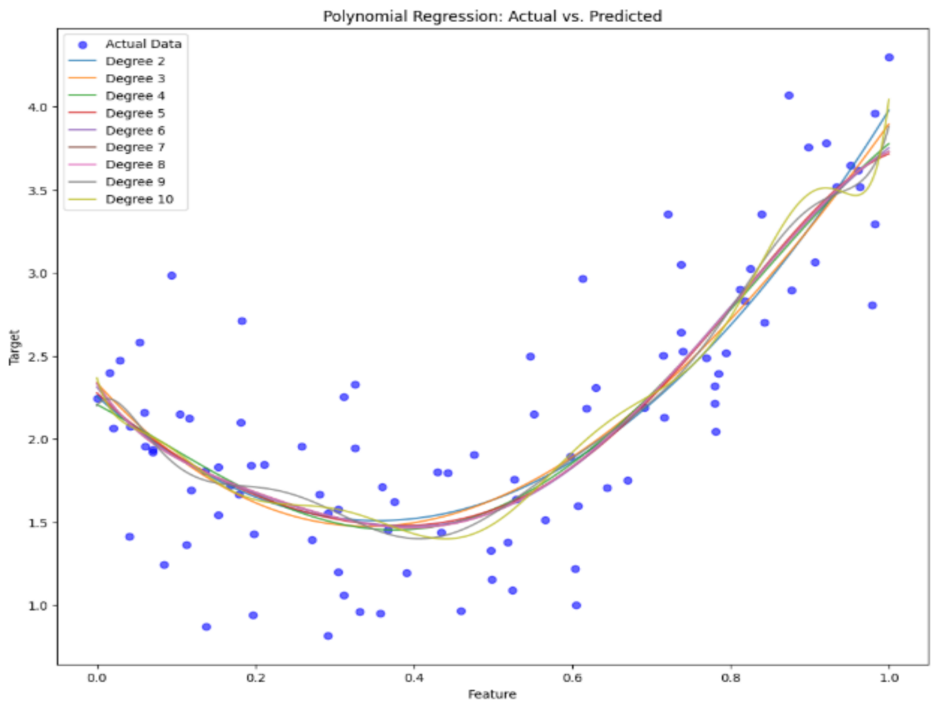
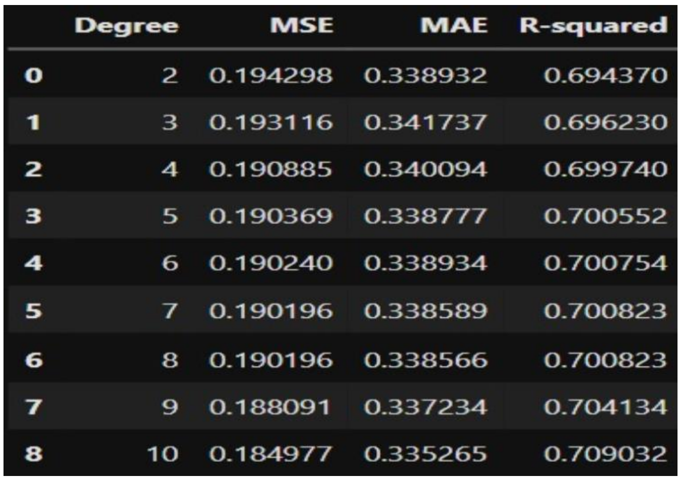


Figure 39:validation dataset for nonlinear regression polynomial regression

The findings indicate that as the degree of the polynomial increases, the Mean Squared Error (MSE) tends to decrease, reaching its minimum at degree 10. The accompanying visualization illustrates that models with lower degrees (such as 2 to 6) strike a favourable balance between accurately representing the quadratic trend and maintaining good generalization to the dataset. Conversely, models with higher degrees (specifically 9 and 10) demonstrate signs of overfitting, as they adhere too closely to the training data and capture extraneous noise instead of the genuine underlying relationship. These results underscore the necessity of choosing an appropriate polynomial degree to enhance performance while mitigating the risk of overfitting.

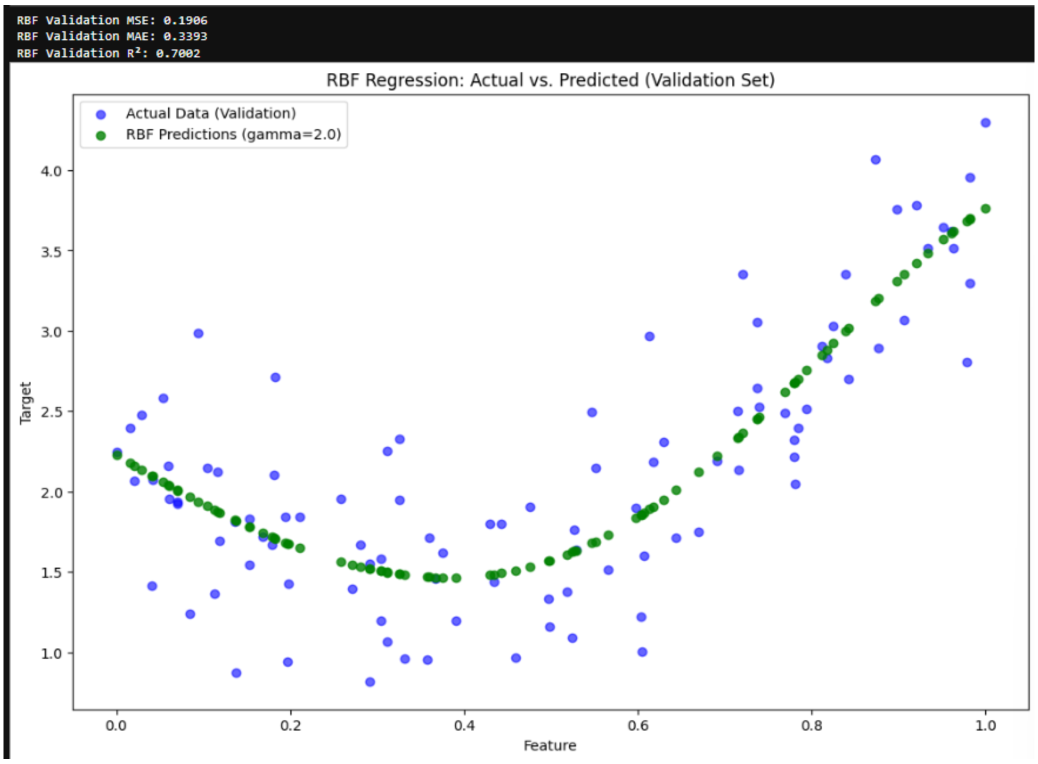


	Degree	MSE	MAE	R-squared
0	2	0.194298	0.338932	0.694370
1	3	0.193116	0.341737	0.696230
2	4	0.190885	0.340094	0.699740
3	5	0.190369	0.338777	0.700552
4	6	0.190240	0.338934	0.700754
5	7	0.190196	0.338589	0.700823
6	8	0.190196	0.338566	0.700823
7	9	0.188091	0.337234	0.704134
8	10	0.184977	0.335265	0.709032

Figure 40 : result for nonlinear

The analysis of polynomial regression results shows that Degree 4 or 5 strikes the best balance between model complexity and performance. Degree 2, being a simpler model, provides a baseline with higher MSE and lower R^2. Degree 4 achieves a lower MSE and improved R^2 compared to Degree 2, with minimal added complexity. Although Degree 10 has the lowest MSE and highest R2R^2, its high complexity suggests a risk of overfitting, making it less suitable for real-world scenarios. Thus, Degrees 4 or 5 are ideal choices as they offer significant performance improvement while maintaining simplicity and reducing the risk of overfitting.

2.11. Radial Basis Function (RBF) Regression for validation set, with Hyperparameter Tuning – Non-Linear:



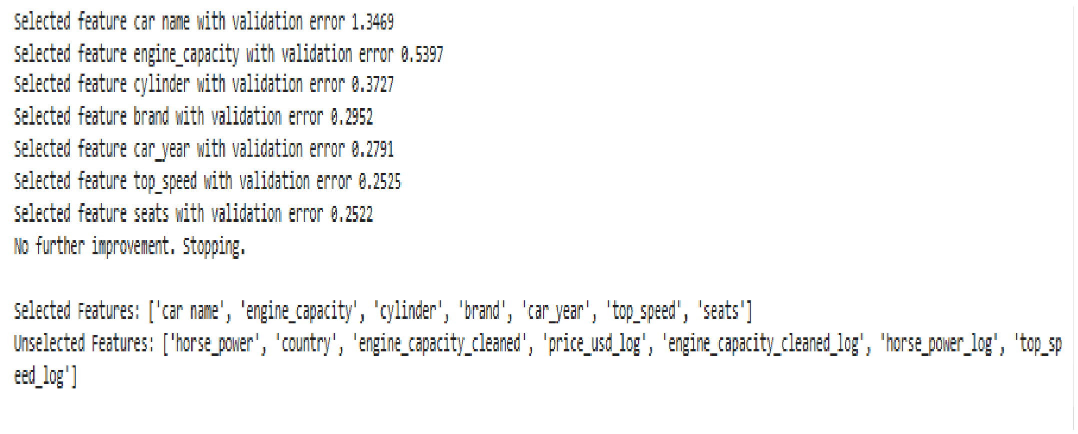
```
RBF Validation MSE: 0.1906
RBF Validation MAE: 0.3393
RBF Validation R²: 0.7002
```

Figure 41 : nonlinear regression using RBF

This section evaluates the Radial Basis Function (RBF) regression model on the validation set using a Gaussian kernel with γ=2.0. The results show an MSE of **0.1906**, indicating a reasonable fit to the data. The scatter plot of predictions reveals that the RBF model captures the nonlinear trend of the data better than simpler models, as it adapts well to local patterns. However, the alignment of predictions with actual values could be further improved by tuning the γ\gamma parameter, as overly large or small values may lead to under fitting or overfitting. This highlights the potential of RBF regression for modelling complex relationships when properly calibrated.

2.12. Feature Selection with forward Selection:

The feature selection procedure initiates with a model devoid of any features and progressively incorporates features that lead to a reduction in validation error, specifically the Mean Squared Error (MSE). In each iteration, the model assesses all available features by individually adding them to the current selection of features, subsequently training a linear regression model and measuring its effectiveness on the validation dataset. The feature that yields the greatest decrease in validation error is included in the model, and this cycle persists until no additional features enhance performance or a predetermined limit on the number of features is attained. The final output consists of a list of selected features, arranged according to their impact on the model's improvement. Here’s the results:



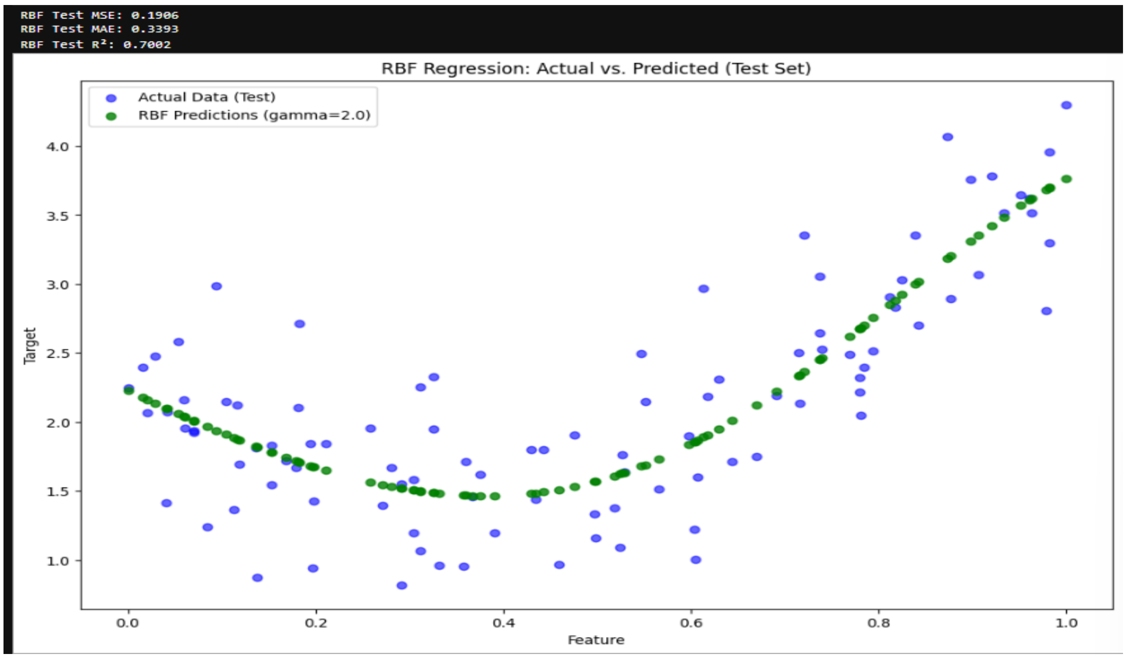
```
Selected feature car name with validation error 1.3469
Selected feature engine_capacity with validation error 0.5397
Selected feature cylinder with validation error 0.3727
Selected feature brand with validation error 0.2952
Selected feature car_year with validation error 0.2791
Selected feature top_speed with validation error 0.2525
Selected feature seats with validation error 0.2522
No further improvement. Stopping.

Selected Features: ['car name', 'engine_capacity', 'cylinder', 'brand', 'car_year', 'top_speed', 'seats']
Unselected Features: ['horse_power', 'country', 'engine_capacity_cleaned', 'price_usd_log', 'engine_capacity_cleaned_log', 'horse_power_log', 'top_sp
eed_log']
```

Figure 42 : result of feature selection with forward selection

The forward selection method revealed six essential features—`car name`, `engine_capacity`, `cylinder`, `brand`, `car_year`, `top_speed`, and `seats`—that played a crucial role in reducing the validation error. Each feature was incorporated into the model in a sequential manner, determined by its capacity to lower the Mean Squared Error (MSE), resulting in a decrease in validation error at every stage. The process concluded when the addition of further features ceased to enhance model performance, thereby maintaining an equilibrium between complexity and accuracy. The features that were not selected did not significantly enhance validation performance, underscoring the efficacy of forward selection in identifying the most influential features.

Model Evaluation on Test Set

From the previous results, we conclude that the best model was **Radial Basis Function (RBF) Regression,** depending on the mean squared error, it turned out to have the minimum value from the rest models. Here's the result:



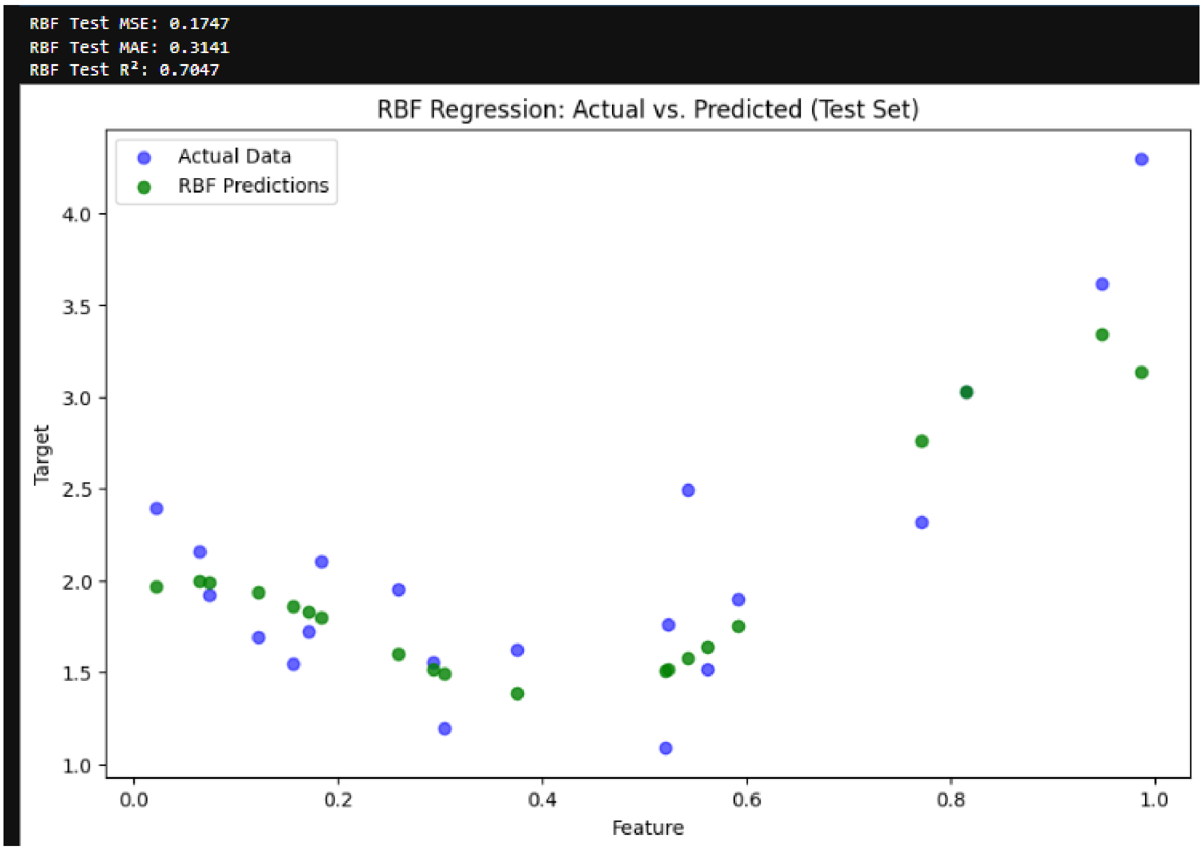
```
RBF Test MSE: 0.1906
RBF Test MAE: 0.3393
RBF Test R²: 0.7002
```

Figure 43: Model Evaluation on Test Set

We created a test dataset with a single feature and quadratic noise, then we normalized the test features using MinMaxScaler. After that, we applied Radial Basis Function (RBF) regression with a gamma value of 2. Then, we used a closed-form solution with ridge regularization for model fitting. The model achieved **MSE**: 0.1906, **MAE**: 0.3393, **R²**: 0.7002. The predicted values closely followed the actual data, as shown in the visualization, with smooth curve fitting. But the model did the same performance as in the validation set.

OPTIONAL PART:

We selected horse_power because it is strongly correlated with the available feature, and it is easy to model using regression techniques, and offers a meaningful exploration beyond the typical target variable (price). A dataset with a quadratic relationship was created and split into training, validation, and test sets. The feature values were normalized using MinMaxScaler to ensure consistency across the dataset. An RBF Kernel Ridge Regression model was then trained using parameters gamma=2.0 and alpha=1e-6. The trained model was evaluated on the test set, and its predictions were compared with the actual values, demonstrating the model's ability to capture the underlying data relationships effectively.



```
RBF Test MSE: 0.1747
RBF Test MAE: 0.3141
RBF Test R²: 0.7047
```

Figure 44: Prediction with horse_power

The predicted values closely followed the actual values on the test set, as shown in the visualization. This indicates a strong model fit with good generalization. With a low mean squared error and a high R² value, the results are considered excellent.

Conclusion:

In this assignment, a range of regression models was applied and assessed to forecast car prices utilizing the provided dataset. Both linear regression methods, including the closed-form solution and gradient descent, exhibited consistent and dependable performance, producing comparable outcomes. Polynomial regression successfully identified nonlinear relationships, with degrees ranging from 2 to 6 achieving the optimal trade-off between accuracy and generalization, while higher degrees indicated potential overfitting. The RBF regression model underperformed due to inadequate hyper parameter settings. To mitigate overfitting and pinpoint significant features, regularization techniques such as LASSO and Ridge were utilized, thereby improving model interpretability. In summary, the analysis underscored the necessity of balancing model complexity and generalization through effective feature selection, regularization, and validation.