Report: RR Car Price Prediction

Include your visualisations, analysis, results, insights, and outcomes. Explain your methodology and approach to the tasks. Add your conclusions to the sections.

1. Data Understanding

```
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.preprocessing import PowerTransformer
import warnings
warnings.filterwarnings('ignore')
from scipy import stats
from collections import Counter
import re
from google.colab import drive
```

Before loading the data, I have imported all the necessary libraries such as Pandas, Numpy, Matplotlib, Sci-Kit Learn's sub-packages, etc.

1.1. Data Loading

1.1.1. Load the dataset

To load the dataset into the file, I have done the following:

1) Mounted Google drive to access the files.
2) Used the *pd.read_csv()* function to read the dataset (which is in CSV format).
3) Displayed some necessary information about the dataset using the *.head()* function - to display the first 5 rows of the dataset, and *.info()* function - to display details of the columns in the dataset. The *display()* function encasing the 2 functions was used instead of *print()* to show the outputs in a readable and user-friendly way.

```
# Load the data
drive.mount('/content/drive')

fileLoc = "/content/drive/MyDrive/Assignment 3/Car_Price_data.csv"
df = pd.read_csv(fileLoc)

display(df.head())
display(df.info())
```

	make_model	body_type	price	vat	km	Type	Fuel	Gears	Comfort_Convenience	Entertainment_Media	...	Previous_Owners	hp_kW
0	Audi A1	Sedans	15770	VAT deductible	56013.0	Used	Diesel	7.0	Air conditioning,Armrest,Automatic climate con...	Bluetooth,Hands-free equipment,On-board comput...	...	2.0	66.0
1	Audi A1	Sedans	14500	Price negotiable	80000.0	Used	Benzine	7.0	Air conditioning,Automatic climate control,Hil...	Bluetooth,Hands-free equipment,On-board comput...	...	1.0	141.0
2	Audi A1	Sedans	14640	VAT deductible	83450.0	Used	Diesel	7.0	Air conditioning,Cruise control,Electrical sid...	MP3,On-board computer	...	1.0	85.0
3	Audi A1	Sedans	14500	VAT deductible	73000.0	Used	Diesel	6.0	Air suspension,Armrest,Auxiliary heating,Elect...	Bluetooth,CD player,Hands-free equipment,MP3,O...	...	1.0	66.0
4	Audi A1	Sedans	16790	VAT deductible	16200.0	Used	Diesel	7.0	Air conditioning,Armrest,Automatic climate con...	Bluetooth,CD player,Hands-free equipment,MP3,O...	...	1.0	66.0

5 rows × 23 columns

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15915 entries, 0 to 15914
Data columns (total 23 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   make_model           15915 non-null  object
 1   body_type            15915 non-null  object
 2   price                15915 non-null  int64
 3   vat                  15915 non-null  object
 4   km                   15915 non-null  float64
 5   Type                 15915 non-null  object
 6   Fuel                 15915 non-null  object
 7   Gears                15915 non-null  float64
 8   Comfort_Convenience  15915 non-null  object
 9   Entertainment_Media  15915 non-null  object
 10  Extras               15915 non-null  object
 11  Safety_Security      15915 non-null  object
 12  age                  15915 non-null  float64
 13  Previous_Owners      15915 non-null  float64
 14  hp_kW                15915 non-null  float64
 15  Inspection_new       15915 non-null  int64
 16  Paint_Type           15915 non-null  object
 17  Upholstery_type      15915 non-null  object
 18  Gearing_Type         15915 non-null  object
 19  Displacement_cc      15915 non-null  float64
 20  Weight_kg            15915 non-null  float64
 21  Drive_chain          15915 non-null  object
 22  cons_comb            15915 non-null  float64
dtypes: float64(8), int64(2), object(13)
memory usage: 2.8+ MB
```

From the outputs above, we see there are 23 columns in the dataset with no null values present for each column. The datatypes of the columns revolve around object (13 columns), float (8 columns) and integer (2 columns).

2. Analysis and Feature Engineering

2.1. Preliminary Analysis and Frequency Distributions

2.1.1. Check and fix missing values

To check and fix missing values in the dataset, I have followed the steps below:

1) The "proportionMissing" variable contains the proportion of missing values per column by using the formula:
   (Sum of values/Length of DataFrame) * 100
   This was done by using the *.sum()* and *len()* in-built functions.
2) Filtering the columns that have missing values helps in keeping only columns where the proportion of missing values is greater than 0. The *.sort_values(ascending=False)* sorts the columns in descending order such that the highest number of missing values in a column will appear first.
3) The rows with missing values (NaN) have been removed using *.dropna()*. After dropping these rows, the shape of the dataset is shown using the *.shape* method.

```
# Find the proportion of missing values in each column and handle if found
proportionMissing = df.isnull().sum() / len(df) * 100
pmValues = proportionMissing[proportionMissing > 0].sort_values(ascending=False)
print("Proportion = \n", pmValues)

df = df.dropna()
print(f"Shape of dataset after handling missing values: {df.shape}")
```

```
Proportion =
 Series([], dtype: float64)
Shape of dataset after handling missing values: (15915, 23)
```

Based on the output above, we see that there are no missing values to handle as the proportion of them are NULL. Hence, there are also no missing values after handling. The shape of the dataset is (15915,23), indicating that there is no change.

Question: From the features, identify the target feature and numerical and categorical predictors. Select the numerical and categorical features carefully as they will be used in analysis.

Answer:

- The Target feature is the *'price'* column.
- The Numerical features columns are: *'km', 'age', 'Previous_Owners', 'hp_kW', 'Displacement_cc', 'Weight_kg', 'cons_comb'*.

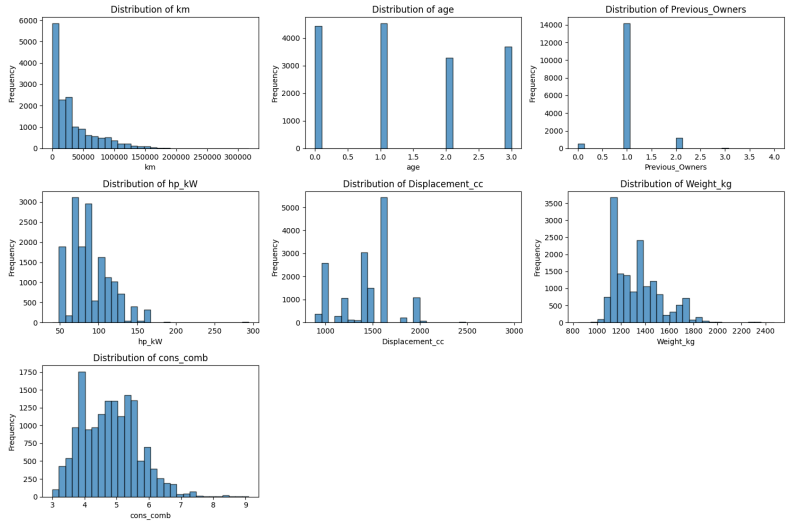
- The Categorical features columns are: *'make_model', 'body_type', 'vat', 'Type', 'Fuel', 'Gears', 'Paint_Type', 'Upholstery_type', 'Gearing_Type', 'Drive_chain'*.

2.1.2. Identify numerical distributions and plot their frequency distributions

The code below shows how to select numerical features and plot their distributions. The steps below are as followed:

1) A list of the numerical features have been established as they contain the features for the histogram plots.
2) A canvas of size 15x10 is created to show each plot in the numerical features list. The for loop iterates through each feature and returns the plot for each feature, where “i” is the index and “feature” is the name of the column. Each plot is given in a 3x3 grid by using the *.subplot(3,3,i)* function.
3) The *plt.hist()* function ends up plotting these graphs with attributes such as “bins=30” for 30 equal intervals, “edgecolor=’black’” for outlining the bars for easy understanding and “alpha=0.7” to make the bars in the plot transparent.
4) The *.tight_layout()* adjusts the layout such that the labels do not overlap one another.

```
# Identify numerical features and plot histograms
numerical_features = ['km', 'age', 'Previous_Owners', 'hp_kW', 'Displacement_cc', 'Weight_kg', 'cons_comb']
plt.figure(figsize=(15, 10))
for i, feature in enumerate(numerical_features, 1):
    plt.subplot(3, 3, i)
    plt.hist(df[feature].dropna(), bins=30, edgecolor='black', alpha=0.7)
    plt.title(f'Distribution of {feature}')
    plt.xlabel(feature)
    plt.ylabel('Frequency')
plt.tight_layout()
plt.show()
```



Details about each plot are given below.

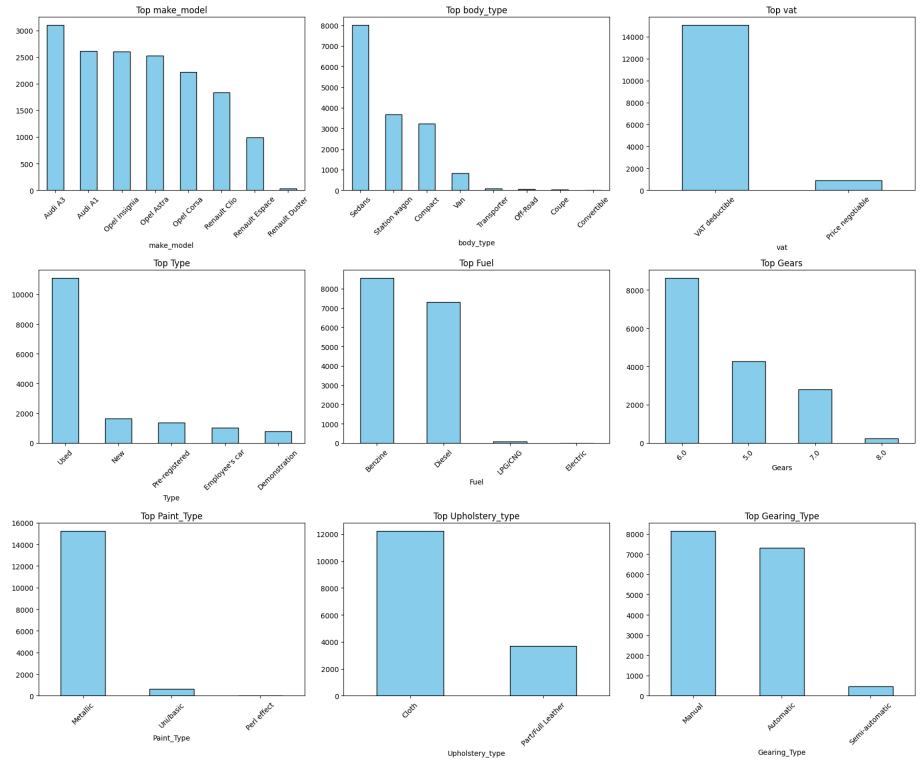
1) **“km” histogram plot**
   The distribution is strongly skewed to the right meaning that most cars have mileage between 0-50,000 km and some of the cars are considered as outliers since their mileage goes above 300,000km.
2) **“age” histogram plot**
   The distribution for the age distribution appears evenly distributed amongst the categories and having a small spread would likely mean that the dataset mostly contains newer cars.
3) **“Previous_Owners” histogram plot**
   There is a very sharp peak at the value 1, with very few cars that have 0-2 owners and almost no cars having 3-4 previous owners. This means that most cars are common in the user-car markets with having only one previous owner.
4) **“hp_kW” histogram plot**
   The histogram looks bell-shaped but is mostly right-skewed meaning that most cars fall between 60-140kW and there are a few outliers that reach 250-300kW which are considered as high-performance cars.
   This suggests that most cars have a mid-range horsepower which is usually the typical distribution for horsepower and some outliers suggesting that they could be premium cars.
5) **“Displacement_cc” histogram plot**
   The histogram here is a multi-modal distribution with several peaks at different sizes of engines. There are clusters being formed around 1.0L, 1.2L,1.4L and 2.0L. This means that there are different car types and that the engine displacement correlates strongly with the output/target feature - price.
6) **“Weight_kg” histogram plot**
   The distribution is right-skewed with most vehicles weighing between 1000-1500kg. There are some heavy vehicles, within the range of 2000-2400kg, considering them as outliers. This could mean that the cars in the dataset have a typical car weight distribution across their segments and the outliers could be large cars like SUVs or commercial vans.
7) **“cons_comb” histogram plot**
   Most of the data points are between 4 to 6 which suggests that this was the typical fuel consumption range of the cars in the dataset. However, a few vehicles have high consumption values but are quite rare.

2.1.3. Identify categorical distributions and plot their frequency distributions

The code below shows the different categorical features that were identified in the dataset and their distributions have been plotted.

1) Similar to the previous section, a list of categorical features have been created so that their distributions can be visualised.
2) A grid of size 3x3 with 9 subplots has been created and the *axes.ravel()* function helps in flattening the array into one dimension which is easier to work with loops in the next step.
3) The for loop will iterate these categorical features and the *.value_counts().head(8)* function counts the number of values in each category and selects the top 8 categories to be displayed.
4) The categories are then plotted using the *.plot()* function with specific attributes like “kind=’bar’” for the type of chart, “color=’skyblue’” for the colour of the bars in the chart.
5) The *.set_title()* and *.tick_params()* functions set the title for each plot and rotate the labels on the x-axis for better understanding and readability, respectively.

```
# Identify categorical columns and check their frequency distributions
categorical_features = ['make_model', 'body_type', 'vat', 'Type', 'Fuel', 'Gears', 'Paint_Type', 'Upholstery_type', 'Gearing_Type', 'Drive_chain']

fig, axes = plt.subplots(3, 3, figsize=(18, 15))
axes = axes.ravel()
for i, feature in enumerate(categorical_features[:9]):
    top_categories = df[feature].value_counts().head(8)
    top_categories.plot(kind='bar', ax=axes[i], color='skyblue', edgecolor='black')
    axes[i].set_title(f'Top {feature}')
    axes[i].tick_params(axis='x', rotation=45)

plt.tight_layout()
plt.show()
```



The details of the categorical features are explained below.

1) **“Make_model” histogram plot**
   This shows the models of the cars that are in the dataset and a few models such as Audi dominate it.
2) **“body_type” histogram plot**
   Body type refers to the type of car such as Wagons, Ertigas, etc. From the plot, we see that Sedans are the most common followed by Wagons and Compacts.
3) **“vat” histogram plot**
   The status is for the vehicle price. Most of the listings in the dataset have deductible VAT which indicates business/dealer listings.
4) **“Type” histogram plot**
   There are a few types of cars that have been recorded in the data set such as “employee’s car” or “new”. From the plot, we see that the majority of the cars have been or are used.
5) **“Fuel” histogram plot**
   There are a few types of fuels that are used, according to the dataset. However, Petrol and Diesel dominate compared to the other types of fuels present.
6) **“Gears” histogram plot**
   There are cars with different types of gears in the dataset but the common ones are 6 gears, which is the most common, then 5 and 7 gears.
7) **“Paint_type” histogram plot**
   There are different types of paint jobs that have been done on the cars in the dataset and the Metallic paint is the most common one amongst them.
8) **“Upholstery_type” histogram plot**
   The cloth material used for upholstery, referring to the interior design of the car, is more common than leather.
9) **“Gearing_type” histogram plot**
   There are different types of gears - manual, semi-automatic and automatic. However from the dataset, manual and automatic gears are popular.

Question: Look carefully at the values stored in columns `["Comfort_Convenience", "Entertainment_Media", "Extras", "Safety_Security"]`.

a) **Should they be considered categorical?**
   Yes, these columns should be considered categorical as they have strings representing features and they are not numerical quantities with an order.

b) **Should they be dropped or handled any other way?**
They should not be dropped as they contain useful information about the car's features, helping in model prediction. They can be handled in other ways such as: Cleaning categorical labels through label encoding or one-hot encoding, Converting multi-label text entries into multi-hot encoded features, Categorise entries/dropping columns (with no meaningful input) if they have high cardinality.

2.1.4. Fix columns with low frequency values and class imbalances

This step was done to improve the quality of data and performance of the model since there might be redundancy and imbalance in categorical features. The steps below show how the columns were fixed:

1) A dictionary is created such that the similar types of categories can be merged into a single category called "Used".
2) The values in the "Type" column are replaced using the .replace() function with the dictionary created in the previous step. Other values that were not present in the dictionary are left unchanged.
3) The for loop will iterate through the categorical features and count the values in the columns and calculate its proportions. Those with a proportion less than 0.01 are considered as rare categories and hence a Boolean mask is created to make it easier.
4) The highlighted categories are replaced using the bitwise NOT operator where True becomes False and False becomes True. This is done since .where() will not change the values where the condition has been satisfied (True).

```
# Fix columns as needed
type_mapping = {
    'Pre-registered': 'Used',
    "Employee's car": 'Used',
    'Demonstration': 'Used'
}
df['Type'] = df['Type'].replace(type_mapping)
for col in ['body_type', 'vat', 'Fuel', 'Gears', 'Paint_Type', 'Upholstery_type', 'Gearing_Type', 'Drive_chain']:
    counts = df[col].value_counts(normalize=True)
    rare_mask = counts < 0.01
    df[col] = df[col].where(~df[col].isin(counts[rare_mask].index), 'Other')

print("Fixed low frequency categories in categorical columns")
print(df['Type'].value_counts())
```

```
Fixed low frequency categories in categorical columns
Type
Used    14266
New      1649
Name: count, dtype: int64
```

From the above output, we see that the original values in the column, “Pre-registered”, “Employee’s Car” and “Demonstration” have been changed into “Used”. Hence, the change in the value counts from 11095, 1364, 1011, 796 into 14266, respectively. Similarly, there have been no modifications made to the “New” column, hence its value remains the same.

2.1.5. Identify target variable and plot the frequency distributions. Apply necessary transformations.

From the previous steps, we have identified that the target variable is the “Price” column. The log transformation is done so as to reduce the skewness of the distribution and tries to compress extreme values. Hence, the steps done below are to plot the frequency distributions.

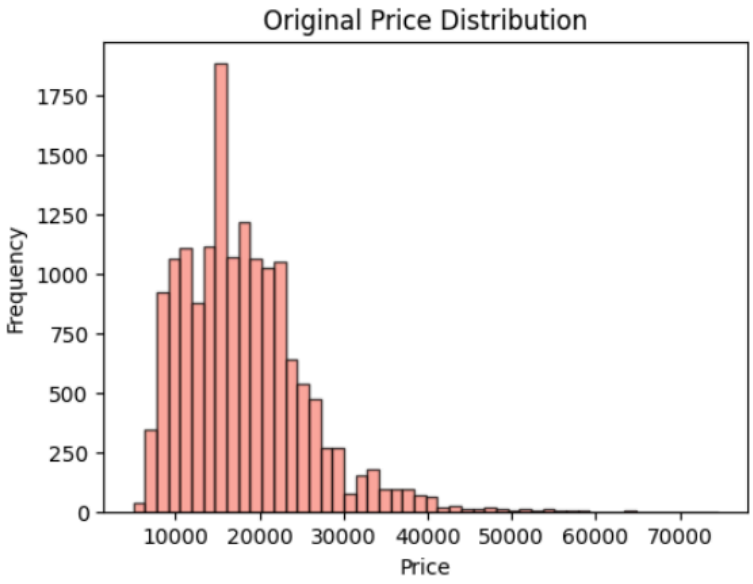
1) In code 1, a canvas of size 12x4 was created with a subplot with 1 row and 2 columns.
2) The *.hist()* function creates a histogram of the values in the “price” columns. The attributes included in the function are “bins=50” for creating 50 equal intervals, “edgecolor=’black’” to outline the bars to make it easier to read, “alpha=0.7” to make the bars in the plots transparent.
3) The title, x-label and y-label of the graphs are set using the *.set_title()*, *.set_xlabel()* and *.set_ylabel()* functions.
4) In code 2, the 2nd subplot with the same layout is created.
5) A log transformation is applied on the “price” column using the *.log1p()* function, following the formula: log(x+1). Finally, a histogram is plotted using the *.hist()* function with similar attributes as in code 1.

```
# Plot histograms for target feature
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].hist(df['price'], bins=50, edgecolor='black', alpha=0.7, color='salmon')
axes[0].set_title('Original Price Distribution')
axes[0].set_xlabel('Price')
axes[0].set_ylabel('Frequency')
```

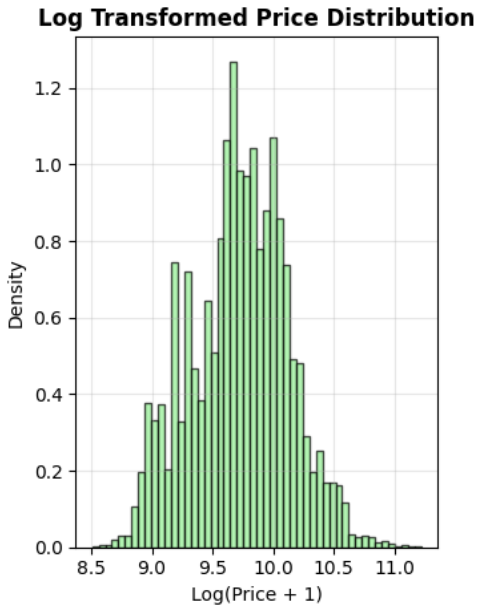
Code 1

```python
# Transform the target feature
plt.subplot(1, 2, 2)
price_log = np.log1p(df['price'])
plt.hist(price_log, bins=50, edgecolor='black', alpha=0.7, color='lightgreen', density=True)
plt.title('Log Transformed Price Distribution', fontsize=12, fontweight='bold')
plt.xlabel('Log(Price + 1)')
plt.ylabel('Density')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
df['log_price'] = np.log1p(df['price'])
```

Code 2



Output 1



Output 2

From output 1, which is the original price distribution, we see that the target variable distribution is positively skewed. This suggests that most car prices are cheaper, being on the lower end and a relatively small number of cars being expensive, being on the other side. This shows that there is more influence of the expensive cars in the dataset.

From output 2, which is the log-transformed price distribution, we see that it is more symmetric and bell-shaped with no extreme high prices that dominate the scale. This means that the transformation has brought the high-priced cars closer to the center of distribution. The outliers are controlled wherein expensive cars exist but their influence on the dataset is reduced, compared to the same in output 1.

From the above outputs, we can summarize that the log transformation was done to reduce skewness and make the distribution more symmetric. The “log1p” formula handles zero values safely, if present in the dataset, as log(0) is undefined.

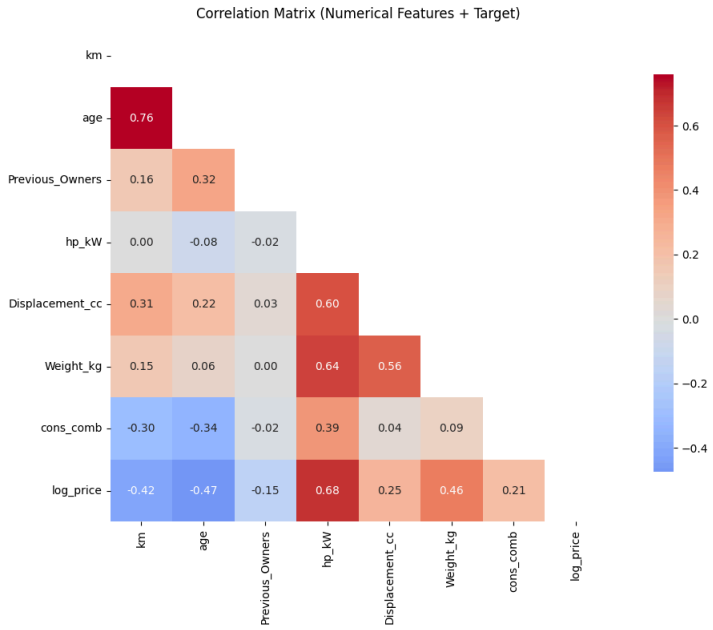
2.2. Correlation Analysis

2.2.1. Plot the correlation map between features and target variable

A correlation map helps to identify which features are strongly related to the target feature and shows where multi-colinearity may exist among the features. The following steps show how the visualisation was created.

1) The numerical features and target feature have been “joined” together for the correlation analysis to include both.
2) The .corr() function computes the correlation matrix using Pearson correlation coefficients and only uses specified numerical columns using df[numericalAndTarget]. The values of a correlation can range between -1 and 1 where,
   +1 = strong positive correlation
   0 = no linear correlation
   -1 = strong negative correlation
3) A mask has been created for the upper triangle of the correlation matrix using the *np.triu()* function to hide any duplicate correlations and improve the readability of the visualisation. The *np.ones_like()* function creates a matrix of all the True values present.

```
# Visualise correlation
numericalAndTarget = numerical_features + ['log_price']
corr_matrix = df[numericalAndTarget].corr()

plt.figure(figsize=(12, 8))
mask = np.triu(np.ones_like(corr_matrix, dtype=bool))
sns.heatmap(corr_matrix, mask=mask, annot=True, cmap='coolwarm', center=0,
            square=True, fmt='.2f', cbar_kws={"shrink": .8})
plt.title('Correlation Matrix (Numerical Features + Target)')
plt.tight_layout()
plt.show()
```



From the output above, we see that there are different types of relationships present between the numerical features and the target variable.

The numerical features with strong positive correlations are:

- “hp_kw” with a correlation of 0.68. This means that the higher the power of the engine, the higher the price of the car.
- “Gears” has a correlation of 0.59, suggesting that as the transmission becomes advanced, the price of the car increases.
- “Weight_kg” also has a strong correlation of 0.46 which shows that heavier cars tend to be more expensive.

The numerical features with strong negative correlations are:

- “age” has a correlation of -0.47, implying that the older the age of the car, the cheaper it is.
- Similarly, “km” has a correlation of -0.42 suggesting that if a car has a higher mileage, it has a lower price tag to it.

And, the numerical features with weak correlations, which can be considered as negligible since they do not affect the data, are:

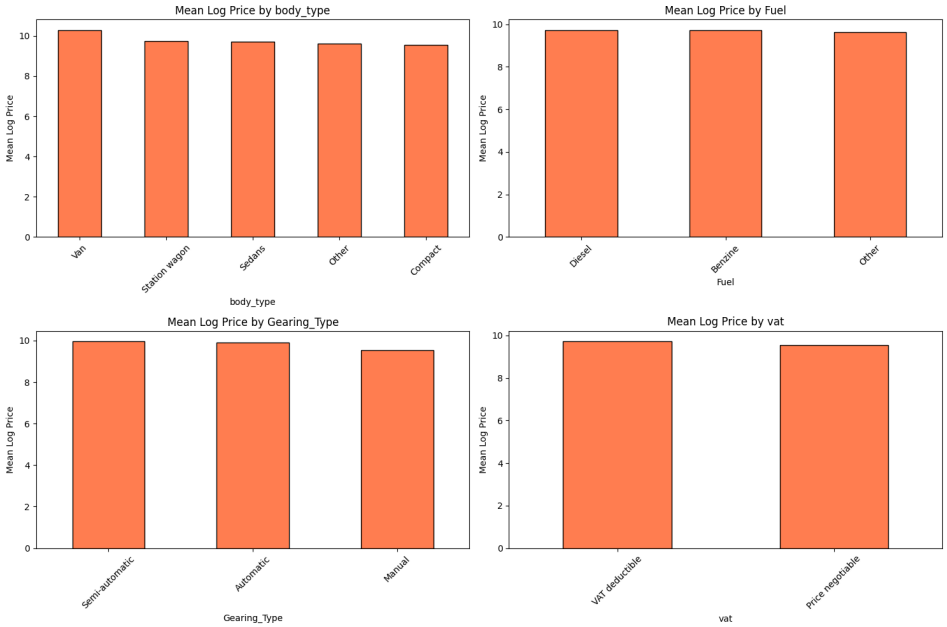
- “Insepection_new” with a very weak positive correlation of 0.03
- “Previous_Owners” with a weak negative correlation of -0.15

2.2.2. Analyse correlation between categorical features and target variable

This analysis was done to identify the categories that are associated with higher or lower prices based on their correlation. It usually helps in feature selection to ensure that meaningful features are used in further model building.

1) The important features, as given in the list, are “body_type”, “Fuel’, “Gearing_Type” and “vat”. These have been selected as they have the strongest influence on the target feature.
2) Within a subplot layout of 2x2, a canvas of size 15x10 is created. Axes[row, column] help in the placement of each plot in the grid.
3) The for loop iterates through the selected categorical features. To determine the subplot position, we use “row = i // 2” and “col = i % 2” to convert the index into a 2D grid position.
4) Then, the mean transformed price is computed by firstly grouping the dataset using the .groupby() function. Then, the mean is computed using the .mean() function. Finally, these values are sorted in descending order using the .sort_values() function and stored in the “cat_means” variable.
5) The bar charts are plotted using the .plot() function to show the mean transformed price per category, with the bars in the plots representing how expensive each category is on average.

```
# Comparing average values of target for different categories
categories = ['body_type', 'Fuel', 'Gearing_Type', 'vat']
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

for i, feature in enumerate(categories):
    row = i // 2
    col = i % 2
    cat_means = df.groupby(feature)['log_price'].mean().sort_values(ascending=False)
    cat_means.plot(kind='bar', ax=axes[row, col], color='coral', edgecolor='black')
    axes[row, col].set_title(f'Mean Log Price by {feature}')
    axes[row, col].tick_params(axis='x', rotation=45)
    axes[row, col].set_ylabel('Mean Log Price')

plt.tight_layout()
plt.show()
```



From the output above, we can interpret the following:

- In the “Mean Log Price by Body Type” bar chart, vans have the highest average price meaning that they are the most expensive compared to the rest while compact cars are the least expensive. This suggests that the larger vehicles tend to have higher prices than smaller vehicles
- In the “Mean Log Price by Fuel Type” bar chart, diesel and petrol vehicles have similar log prices, as compared to the “Other” category. This suggests that fuel type, especially diesel and petrol, have some influence on the pricing of a car.
- In the “Mean log Price by Gearing Type” plot, we see that the vehicles with manual gear type are relatively cheaper than those with semi-automatic and fully automatic gear type, suggesting that gear type and systems will increase the price of the car.

- In the “Mean Log Price by VAT Type” plot, the price-negotiable vehicles are cheaper than the vehicles with VAT deductible status. This suggests that the VAT-deductible vehicles are usually dealer or business listed hence having a higher price.

2.3. Outlier Analysis

2.3.1. Identify potential outliers in the data

Identifying outliers is important as it will result in less bias in model training and lead to accurate predictions. It also helps in deciding where or if extreme values need to be removed or transformed. The following steps are done to identify the potential outliers.

1) A function “detect_outliers_iqr” is created and uses the InterQuartile Range (IQR) to identify the outliers in the data.
2) The 1st (25th percentile) and 3rd quartiles (75th percentile) are calculated using the .quantile(0.25) and .quantile(0.75) function. This helps in defining the middle of the data at 50%.
3) IQR is computed using the formula IQR = Q3 - Q1, to measure the spread of data while capping the extreme values.
4) The thresholds for outliers are calculated using the lower and upper bounds with the formulas:
   Lower = Q1 - 1.5 * IQR
   Upper = Q3 + 1.5 * IQR
   The value 1.5 is a standard/constant that is used when calculating as such. Any values below the lower bound and values above the upper bound are considered as outliers.
5) The logical OR operator is used to filter the rows of columns where values are outside the range calculated in the previous step. The number of outliers identified in each column is returned using the return len(outliers) method.
6) The summary of the outliers is given through the for loop which iterates through each feature and calls the “detect_outliers_iqr” function created in the 1st step. The outlier count is then stored in the outlier_summary dictionary. The percentage of outliers compared to the size of the dataset is also calculated using the formula: (outlier-count/len(df))*100.

```python
# Outliers present in each column
def detect_outliers_iqr(data, column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = data[(data[column] < lower_bound) | (data[column] > upper_bound)]
    return len(outliers)

outlier_summary = {}
for col in numerical_features:
    outlier_count = detect_outliers_iqr(df, col)
    outlier_summary[col] = outlier_count
    print(f"{col}: {outlier_count} outliers ({outlier_count/len(df)*100:.1f}%)")

outlier_df = pd.DataFrame(list(outlier_summary.items()), columns=['Feature', 'Outliers'])
outlier_df['Outlier_%'] = (outlier_df['Outliers'] / len(df) * 100).round(2)
print("\nOutlier Summary:")
print(outlier_df.sort_values('Outliers', ascending=False))
```

```
km: 689 outliers (4.3%)
age: 0 outliers (0.0%)
Previous_Owners: 1757 outliers (11.0%)
hp_kW: 361 outliers (2.3%)
Displacement_cc: 21 outliers (0.1%)
Weight_kg: 87 outliers (0.5%)
cons_comb: 125 outliers (0.8%)

Outlier Summary:
           Feature  Outliers  Outlier_%
2  Previous_Owners      1757      11.04
0               km       689       4.33
3            hp_kW       361       2.27
6        cons_comb       125       0.79
5        Weight_kg        87       0.55
4  Displacement_cc        21       0.13
1              age         0       0.00
```

From the output above we see the different outliers per column where: “km” = 689, “age” = 0, “Previous_Owners” = 1757, “hp_kW” = 361, “Displacement_cc” = 21, “Weight_kg” = 87, “cons_comb” = 125.

From the outlier summary, we see that “Previous_owners” has the highest number of outliers which could represent cases of reselling of the cars. The ‘km” column holds the next highest percentage of outliers in the dataset suggesting that they are realistic but not as accurate.

2.3.2. Handle the outliers suitably

Outliers are handled based on IQR capping where each numerical column is capped to the respective boundary instead of being removed. This helps in removing extreme values in the columns. The steps taken are given below:

1) A function called “cap_outliers” is created and takes in the dataset, in the ‘df’ variable and column name that has to be processed in the ‘column’ variable.

2) The 1st (25th percentile) and 3rd quartiles (75th percentile) are calculated using the *.quantile(0.25)* and *.quantile(0.75)* function. This helps in defining the middle of the data at 50%.
3) The thresholds for outliers are calculated using the lower and upper bounds with the formulas:
   Lower = Q1 - 1.5 * IQR
   Upper = Q3 + 1.5 * IQR
   The value 1.5 is a standard/constant that is used when calculating as such. Any values below the lower bound and values above the upper bound are considered as outliers. These outliers are capped using the *.clip()* function.
4) The dataset is returned and a for loop is used to loop through all the columns to cap the outliers based on their IQR.

```
# Handle outliers
def cap_outliers(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    df[column] = df[column].clip(lower_bound, upper_bound)
    return df

for col in numerical_features:
    df = cap_outliers(df, col)

print("Outliers capped using IQR method (1.5 * IQR)")
```

```
Outliers capped using IQR method (1.5 * IQR)
```

2.4. Feature Engineering

2.4.1. Fix any redundant columns and create new ones if needed

The code below removes any feature that has high cardinality and creates a new row if needed to capture patterns better for future model building. High cardinality refers to a feature that has a large number of values and increases the chances of overfitting when training the model. Hence, the steps below are done to fix these issues.

1) The “if” condition checks if the “make_model” column exists in the dataset. In the case that it does, the said column will be dropped using the .drop() function. Else, nothing will happen since it might not be in the dataset. The attribute “axis=1” in the code is to specify that the whole COLUMN has to be deleted, as the default is the row.

2) A new feature, “age_km_ratio” is created as it is numeric and also represents the average distance driven in each year. The formula is: KM / (Year + 1), to prevent zero division errors in the scenario where age = 0.
3) The newly created feature is then added to the list of existing numerical features in “numeric_features” using the .append() function. The output of the process is displayed using the print() function.

```
# Fix/create columns as needed
if 'make_model' in df.columns:
    df = df.drop('make_model', axis=1)
    print("Dropped 'make_model' due to high cardinality")
df['age_km_ratio'] = df['km'] / (df['age'] + 1)
numerical_features.append('age_km_ratio')
print("Created 'age_km_ratio' feature")
print(f"Final features shape: {df.shape}")
```

```
Dropped 'make_model' due to high cardinality
Created 'age_km_ratio' feature
Final features shape: (15915, 24)
```

The output above shows that the “make_model” column has been dropped and the new “age_km_ratio” feature has been added to the list of numerical features. Hence, the final shape of the dataset, as shown, is (15915, 24) representing 15915 rows and 24 columns.

2.4.2. Analysis and feature engineering on selected columns

The code is used to see what are the unique values present in the 4 features (refer to code 1) and then remove them (refer to code 2) if they are not suitable to use directly in building and training the model.

1) In code 1, the columns in the “feature_spec_cols” list are pre-defined as they usually have multiple options that are in string format. The for loop iterates through each feature and firstly checks if the respective column exists in the dataset, else it will return an error.
2) Then, *.nunique()* function is used to count the number of unique values present in the column to determine the type of encoding that can be done or if it should be dropped since it might lead to issues when training the model.
3) If the feature has more than 20 values, it returns a statement that there are a certain number of values.

4) In code 2, the columns are being dropped using a list comprehension such that it helps in preventing errors during dropping and keeps columns that actually exist in the dataset.
5) The .drop() function drops the selected columns in “cols_to_drop” and the attributes “axis = 1” allows for column-wise deletion and the “errors = ‘ignore’” skips any missing columns in the dataset without throwing up errors.

```
# Check unique values in each feature spec column
feature_spec_cols = ['Comfort_Convenience', 'Entertainment_Media', 'Extras', 'Safety_Security']
for col in feature_spec_cols:
    if col in df.columns:
        unique_count = df[col].nunique()
        print(f"{col}: {unique_count} are unique values")
        if unique_count < 20:
            print(f"Sample: {df[col].dropna().unique()[:5]}")
        print()
```

Code 1

```
Comfort_Convenience: 6196 are unique values

Entertainment_Media: 346 are unique values

Extras: 659 are unique values

Safety_Security: 4442 are unique values
```

Output 1

```
# Drop features from df
feature_spec_cols = ['Comfort_Convenience', 'Entertainment_Media', 'Extras', 'Safety_Security']
cols_to_drop = [col for col in feature_spec_cols if col in df.columns]
df = df.drop(cols_to_drop, axis=1, errors='ignore')
print(f"Dropped feature spec columns: {cols_to_drop}")
```

Code 2

```
Dropped feature spec columns: ['Comfort_Convenience', 'Entertainment_Media', 'Extras', 'Safety_Security']
```

Output 2

From output 1, we see that the 4 features have many unique values suggesting that these features have some unstructured text fields rather than being categorical columns. Encoding these would result in many dummy fields being created within the dataset, making it complex and inefficiently large for the model.

Hence, as shown in output 2, we drop these 4 feature columns to make it easier for training the model.

2.4.3. Perform Feature Encoding

The code below does the feature encoding of specific columns such that they can be used in training and building the models.

1) A list of categorical columns is specified as they need to be changed into numerical features due to the fields containing text which cannot be processed by models.
2) A dictionary named “label_encoders” is created to store the columns after encoding.
3) A for loop iterates over each feature and firstly checks if it exists in the dataset using the “if” condition. If it does not exist, a runtime error will not be thrown.
4) The LabelEncoder() object is initialised so that each unique category is mapped to a unique integer value.
5) Missing values are handled using the .fillna() function to replace NaN values with “Missing” and all the values are converted to the String datatype using .astype() so that the encoding does not fail due to any null values that may be present.
6) Each column is then being fitted and transformed using the .fit_transform() function. Fit means to learn the unique categories present in the column while Transform means to convert them into integers. The encoder is stored as it can be useful for encoding-decoding processes in the upcoming sections.

```
# Encode features
catFeaturesToEncode = ['body_type', 'vat', 'Type', 'Fuel', 'Gears', 'Paint_Type',
                       'Upholstery_type', 'Gearing_Type', 'Drive_chain']

label_encoders = {}
for col in catFeaturesToEncode:
    if col in df.columns:
        le = LabelEncoder()
        df[col] = df[col].fillna('Missing').astype(str)
        df[col] = le.fit_transform(df[col])
        label_encoders[col] = le
print(f"Encoded columns: {list(label_encoders.keys())}")
df.head(15)
```

```
Encoded columns: ['body_type', 'vat', 'Type', 'Fuel', 'Gears', 'Paint_Type', 'Upholstery_type', 'Gearing_Type', 'Drive_chain']
```

	body_type	price	vat	km	Type	Fuel	Gears	age	Previous_Owners	hp_kW	Inspection_new	Paint_Type	Upholstery_type	Gearing_Type	Displacement_
0	2	15770	1	56013.0	1	1	2	3.0	1.0	66.0	1	0	0	0	1422
1	2	14500	0	80000.0	1	0	2	2.0	1.0	141.0	0	0	0	0	1798
2	2	14640	1	83450.0	1	1	2	3.0	1.0	85.0	0	0	0	0	1598
3	2	14500	1	73000.0	1	1	1	3.0	1.0	66.0	0	0	0	0	1422
4	2	16790	1	16200.0	1	1	2	3.0	1.0	66.0	1	0	0	0	1422
5	2	15090	1	63668.0	1	1	2	3.0	1.0	85.0	0	0	1	0	1598
6	3	16422	1	62111.0	1	1	2	3.0	1.0	85.0	1	0	1	0	1598
7	0	14480	1	14986.0	1	1	2	3.0	1.0	66.0	1	0	0	0	1422

From the output above, we see that the 9 columns that were mentioned in the “catFeaturesToEncode” list have been encoded. The first few rows of

data show the numerical values present in the columns, after encoding. Hence, it is ready for training models and algorithms.

2.4.4. Split the data into training and testing sets

The code below splits the data such that a part of it can be used to train the linear regression model, and the other part is used to test the model after building and training.

1) The features and target variables are split from each other where the “X” variable contains the input features and the “y” variable contains the target variable. Both columns “price” and “log_price” are removed and “log_price” is considered as the target variable.
2) The *train_test_split()* method splits the entire dataset into 80% training data and 20% testing data, where the “test_size” attribute is set to 0.2. The “random_state” attribute is set to 42 to ensure reproducibility and the “shuffle = True” attribute ensures that all the rows are randomised when training the data.
3) The size of the training and testing sets are displayed using the *.shape* attribute. This helps in confirming that the data has been split correctly and the input counts are the same as before splitting.
4) The range of target values in both training and testing sets are also displayed to show the minimum and maximum prices that could be given as outputs. This ensures that there is no shift in distribution after the data was split.

```
# Split data
X = df.drop(['price', 'log_price'], axis=1)
y = df['log_price']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, shuffle=True)

print(f"Training set: {X_train.shape}")
print(f"Test set: {X_test.shape}")
print(f"Target range - Train: {y_train.min():.2f} to {y_train.max():.2f}")
print(f"Target range - Test: {y_test.min():.2f} to {y_test.max():.2f}")
```

```
Training set: (12732, 18)
Test set: (3183, 18)
Target range - Train: 8.52 to 11.22
Target range - Test: 8.51 to 11.07
```

From the output above, the shape of the sets are in the format of (observations, features) where the training set contains 12732 observations with 18 features and the test set contains 3183 observations with 18 features. Hence, showing that the dataset was split exactly into the 80% - 20% ratio.

The range values (minimum and maximum) of the target variable for both training and test sets are similar, as shown above. This shows that the sampling was completely randomised and that the test set is a good subset/representative of the training set.

2.4.5. Scale the features

The code below shows all the input features being scaled. Scaling is important so that they are all on the same scale, by having an average of 0. The formula of the scaler is the StandardScaler, which follows the Z-score distribution using:
z = (x - Mean) / Standard Deviation

1) The StandardScaler(), from the scikit-learn package, is initialised as an object.
2) The training data (in X_train) is fitted and scaled using the *.fit_transform()* function where:
   Fitting will compute the mean and standard deviation of the features within the training data only, and
   Transforming will scale the data (X_train) using the statistics provided in fitting.
3) Similar to the previous step, I have transformed the X_test data using the *.transform()* function instead so that the parameters, without fitting it again, will keep the evaluation of the model fair.
4) The shape and average of the scaled X_train set (X_train_scaled) are then displayed.

```
# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print(f"Scaler used = Standard Scaler \nShape of scaled X-train set -> {X_train_scaled.shape}")
print(f"Averages after scaling -> {np.mean(X_train_scaled, axis=0).round(3)}")
```

```
Scaler used = Standard Scaler
Shape of scaled X-train set -> (12732, 18)
Averages after scaling -> [ 0. -0. -0.  0. -0. -0. -0.  0. -0. -0. -0. -0.  0.  0. -0.  0. -0. -0.]
```

From the output above, we see that the scaled training set has 12732 observations with 18 features, same as before scaling.

We all see that each average value of the feature is either 0 or -0, ensuring that the StandardScaler was done correctly and each feature, after scaling, has an average of 0.

3. Linear Regression Models

3.1. Baseline Linear Regression Model

3.1.1. Build and fit a basic linear regression model. Perform evaluation using suitable metrics.

Part A - Initialise and train model

1) The LinearRegression() method initialises an object for the linear regression model. It is used to learn the relationship between the features and the target variable.
2) The *.fit()* function trains the linear regression model using the scaled training features (X_train_scaled) and the target value (y_train).
3) The learned coefficient is given as an output using the *.coef_* method that is in-built and the *len()* function results in the number of input features that were given to the model.

```
# Initialise and train model
lr_model = LinearRegression()
lr_model.fit(X_train_scaled, y_train)

print(f"Number of features: {len(lr_model.coef_)}")
```

```
Number of features: 18
```

From the output above, we see that the number of input features fitted to the model is 18 as shown in the previous steps as well.

Part B - Evaluate model performance

1) Both “y_train_pred_lr” and “y_test_pred_lr” are used to generate predicted log prices for the training and testing data, respectively.
2) The Mean Absolute Error (MAE) is calculated for both train and test data. It is called using the *mean_absolute_error()* function, which calculates the mean difference between the actual and predicated values.
   If the MAE is high, it means that the predictions are less accurate. Else if the MAE is low, it means that the predictions are more accurate.
3) Similarly, the Root Mean Squared Error (RMSE) is calculated for both types of data to detect any large mistakes in the predictions. It is called using both *mean_squared_error()* and *np.sqrt()* - as the name “Root” suggests.

4) The R^2 score of the target variable is also calculated using the *r2_score()* function. It measures the measure of variance in the target feature. Its values are in between 0 and 1, where a higher R^2 score suggests that the model is performing well.

```
# Evaluate the model's performance
y_train_pred_lr = lr_model.predict(X_train_scaled)
y_test_pred_lr = lr_model.predict(X_test_scaled)

maeLrTrain = mean_absolute_error(y_train, y_train_pred_lr)
maeLrTest = mean_absolute_error(y_test, y_test_pred_lr)
rmseLrTrain = np.sqrt(mean_squared_error(y_train, y_train_pred_lr))
rmseLrTest = np.sqrt(mean_squared_error(y_test, y_test_pred_lr))
r2LinearRegression = r2_score(y_test, y_test_pred_lr)

print(f"Train MAE: {maeLrTrain:.2f} \nTest MAE: {maeLrTest:.2f}")
print(f"Train RMSE: {rmseLrTrain:.2f} \nTest RMSE: {rmseLrTest:.2f}")
print(f"Test R²: {r2LinearRegression:.2f}")
```

```
Train MAE: 0.14
Test MAE: 0.14
Train RMSE: 0.18
Test RMSE: 0.18
Test R²: 0.81
```

As shown in the output above, the MAE and RMSE values for both train and test data are the same, indicating there is no overfitting of the model and it is able to perform well on new and unseen data.

As mentioned, a low MSE (0.14) means that the car pricing predictions are similar to those of the actual transformed car prices.

The R^2 score shows a strong performance of the model where 81% of it is able to accurately predict the car prices.

3.1.2. Analyse residuals and check other assumptions of linear regression.

Part A - Check for linearity by analysing residuals vs predicted values

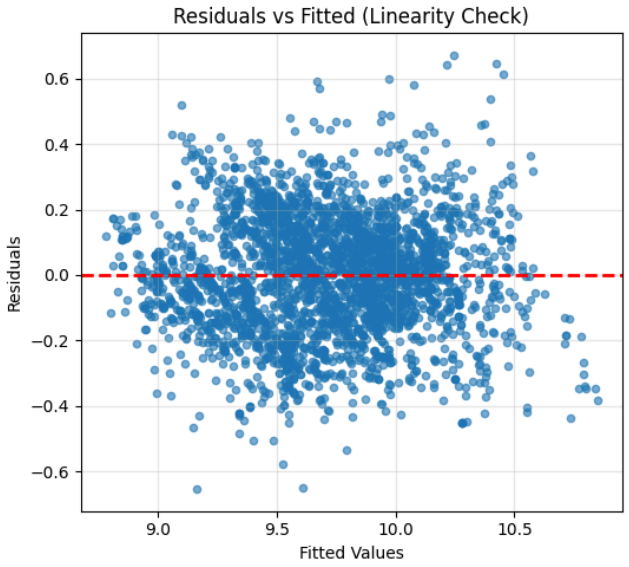
Checking for linearity will help in ensuring that the relationship between the features and target variable are in a linear relationship.

1) The residuals, referred to as the difference between actual and predicted values, are calculated using the formula: y_test - y_test_pred_lr. This helps in showing the error the model has made for each data point. Similarly, the predicted values from the model are stored in “fitted_lr”.
2) A canvas of size 15x5 is created and a scatter plot is created using the *.scatter()* function. The x-axis contains predicted values

while the y-axis contains residuals. The “alpha = 0.6” attribute adds transparency to the plots to reduce overlapped data points.

3) A horizontal red line is created as a reference for residual value being 0, using the *.axhline()* function. This helps to easily see if the residual points are centered around 0.

```
# Linearity check: Plot residuals vs fitted values
residuals = y_test - y_test_pred_lr
fitted_lr = y_test_pred_lr

plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.scatter(fitted_lr, residuals, alpha=0.6, s=20)
plt.axhline(y=0, color='r', linestyle='--', linewidth=2)
plt.xlabel('Fitted Values')
plt.ylabel('Residuals')
plt.title('Residuals vs Fitted (Linearity Check)')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```



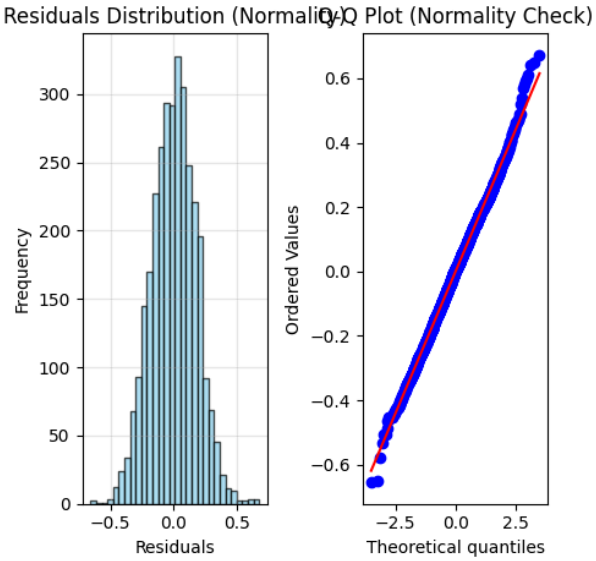
The output shows that most of the residual points are symmetrical around the 0 line, showing that it is consistent. Hence, this proves that the model is able to capture the linear relationship of the points.

Part B - Check normality in residual distribution

Checking the normality of the residual points is important as it ensures that statistical calculations that are done from the linear regression model are accurate. It also helps in ensuring that predictions are random and errors are symmetrically distributed.

1) A subplot canvas is created such that 2 plots can be placed side-by-side for easy comparison and analysis.
2) The first plot is a histogram of the distribution of residuals, using the *.hist()* function. It has 30 equal intervals and a black edge color to improve the visibility of the bar chart within the plot. The *plt.grid()* function adds a grid to the plot for easy readability of values.
3) The second plot is a quantile plot, using stats.probplot(), which compares the quantiles of all the residual points. This is done to check if the distribution matches the actual values.

```
# Check the normality of residuals by plotting their distribution
plt.subplot(1, 3, 2)
plt.hist(residuals, bins=30, edgecolor='black', alpha=0.7, color='skyblue')
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.title('Residuals Distribution (Normality)')
plt.grid(True, alpha=0.3)

plt.subplot(1, 3, 3)
stats.probplot(residuals, dist="norm", plot=plt)
plt.title('Q-Q Plot (Normality Check)')

plt.tight_layout()
plt.show()
```



From the histogram plot on the left , we see that the residual points form a symmetric distribution showing that the predictions and errors that are present are balanced.

From the Quantile plot on the right, we see that most of these residuals are close to the reference line (in red). This means that the residuals are

normal. However, there is a slight difference at the head and tail of the plot.

Part C - Check multicollinearity using Variance Inflation Factor (VIF) and handle features with high VIF.

Variance Inflation Factor (VIF) is used to check where the input features are highly correlated with each other. Doing so helps to improve the reliability and robustness of the model.

The formula for VIF is VIF = $1/(1-R^2)$. If the R^2 is highly predictable, there is strong multicollinearity within the input features in the dataset.

The steps followed are below.

1) A function named “calculate_vif” has been defined to calculate the Variance Inflation Factor for each input feature.
2) The “vif_data” is a DataFrame, created using *pd.DataFrame()*, to store the names of the input features. This can be extracted from the “X” dataset (after splitting) by using the *.columns* function.
3) A list comprehension calculates VIF for each feature by regressing it against the other features present. The VIF is then computed for all the training features.
4) The top 10 features with the highest multicollinearity are displayed using the *.head(10*) function and in descending order using *.sort_values()*. 2 new lists are created called “high_vif_features” and “features_to_drop” to be used in the next steps. The *.copy()* function creates a copy of the DataFrame to remove features instead of using the original DataFrame.
5) A while loop is used to repeatedly remove features with a high VIF. Within the while loop, the “calculate_vif” function is called and it goes through all the features to find those with the highest VIF in the dataset. If the maximum VIF is greater than 10 using the *.max()* function, it is considered as high multicollinearity.
6) The features that qualify from the previous step are dropped using the *.drop()* function. The *.append()* function will add these features into the “features_to_drop” list. The while loop stops once the other features have the VIF values within the acceptable range.

```python
# Check for multicollinearity and handle
def calculate_vif(X):
    vif_data = pd.DataFrame()
    vif_data["feature"] = X.columns
    vif_data["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
    return vif_data

vif_df = calculate_vif(X_train)
print("VIF Scores (Top 10 highest):")
print(vif_df.sort_values('VIF', ascending=False).head(10))
high_vif_features = []
features_to_drop = []
temp_X = X_train.copy()

while True:
    vif_temp = calculate_vif(temp_X)
    max_vif_feature = vif_temp.loc[vif_temp['VIF'].idxmax(), 'feature']
    if vif_temp['VIF'].max() > 10:
        high_vif_features.append(max_vif_feature)
        temp_X = temp_X.drop(max_vif_feature, axis=1)
        features_to_drop.append(max_vif_feature)
    else:break

print(f"\nDropped {len(features_to_drop)} high VIF features: {features_to_drop[:5]}...")
```

```
VIF Scores (Top 10 highest):
           feature         VIF
7   Previous_Owners  258.812680
2               km   16.728016
17    age_km_ratio   11.649112
4             Fuel    4.463297
13  Displacement_cc    3.896579
6              age    3.886380
8            hp_kW    3.750191
16       cons_comb    2.675391
14       Weight_kg    2.511989
5            Gears    1.703925

Dropped 7 high VIF features: ['Previous_Owners', 'Displacement_cc', 'Weight_kg', 'cons_comb', 'Drive_chain']
```

From the output above, there are features such as “Previous_Owners”, “km” and “age_km_ratio” with high VIF values. This suggests that they have high multicollinearity and are strongly dependent on the other features present in the dataset. The remaining features have a VIF value that is less than 5 and are acceptable, hence they have not been dropped. Those that have been dropped are strongly correlated with other variables or have information that is not necessary.

3.2. Ridge Regression Implementation

3.2.1. Define a list of random alpha values

The alpha values are a hyperparameter to control the regularisation. A large alpha value means that the coefficients will shrink more and the model behaves less like a linear regression, and the opposite for small alpha values.

A ridge regularisation adds the penalty to the function to lessen the coefficients. This reduces overfitting of the data and can work with new and unseen data.

1) The *np.logspace()* function is used to get the optimal alpha values not in a linear fashion. It generates 50 logarithmic values (base 10) within a (-3,3) linear space.

2) The range of alpha values are displayed in an ascending order with the formatting in 4 decimal places.

```
# List of alphas to tune for Ridge regularisation
alphaRidge = np.logspace(-3, 3, 50)
print(f"Coarse alpha range: {alphaRidge[0]:.4f} to {alphaRidge[-1]:.4f}")
```

```
Coarse alpha range: 0.0010 to 1000.0000
```

From the output above, we see that the range is from the smallest alpha value of 0.001 to the largest alpha value of 1000. The smallest value displays a very weak regularisation where the model is close to the linear regression and the largest value has a very strong regularisation for coefficients to be shrunk. This covers all the types of regularised models that could have occurred.

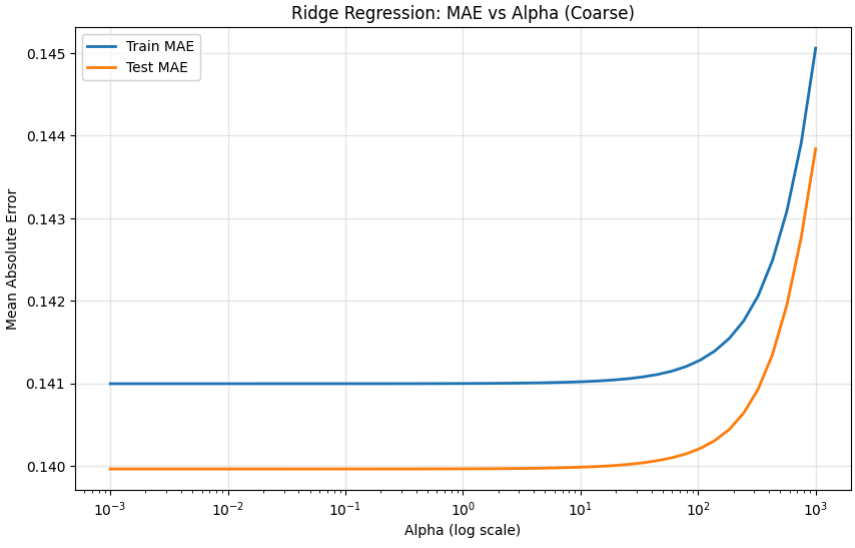
3.2.2. Apply Ridge Regularisation and find the best value of alpha from the list

The codes for this section are split into 2 parts - applying ridge regression, plotting the output and the output are in part A, and best alpha value is in part B.

Part A - Applying ridge regression and plotting of graph

1) There are 2 lists created, “train_scores_ridge” and “test_scores_ridge”, to store the training and testing Mean Absolute Errors (MAEs) for each alpha value.
2) The for loop iterates through the different alpha values that have been previously defined. Within it, a ridge regression model is initialized using the *Ridge()* package and it is fitted with the scaled training data of both input features and target variable using the *.fit()* function.
3) The predictions for both the training and testing datasets of input features are generated using *.predict()*. Then, the MAE is calculated for each of these using *.mean_absolute_error()*. This is stored in the 2 lists initially created by appending (*.append()*) them. The lists are then converted to NumPy arrays for easier analysis and indexing when needed.
4) The MAE values of both training and test sets are plotted against the alpha values using a logarithmic x-axis graph using *.semilogx()*.
5) The best alpha values are found using *np.argmin()* and displayed. The function returns the index position of the minimum error result in the array, as seen with the code “alphaRidge[bestIndexC]”.

```python
# Applying Ridge regression
train_scores_ridge = []
test_scores_ridge = []

for alpha in alphaRidge:
    ridge = Ridge(alpha=alpha)
    ridge.fit(X_train_scaled, y_train)
    train_pred = ridge.predict(X_train_scaled)
    test_pred = ridge.predict(X_test_scaled)
    train_scores_ridge.append(mean_absolute_error(y_train, train_pred))
    test_scores_ridge.append(mean_absolute_error(y_test, test_pred))
train_scores_ridge = np.array(train_scores_ridge)
test_scores_ridge = np.array(test_scores_ridge)


# Plot train and test scores against alpha
plt.figure(figsize=(10, 6))
plt.semilogx(alphaRidge, train_scores_ridge, label='Train MAE', linewidth=2)
plt.semilogx(alphaRidge, test_scores_ridge, label='Test MAE', linewidth=2)
plt.xlabel('Alpha (log scale)')
plt.ylabel('Mean Absolute Error')
plt.title('Ridge Regression: MAE vs Alpha (Coarse)')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

best_idx_coarse = np.argmin(test_scores_ridge)
best_alpha_coarse = alphaRidge[best_idx_coarse]
print(f"Best coarse alpha: {best_alpha_coarse:.4f}")
print(f"Best coarse test MAE: {test_scores_ridge[best_idx_coarse]:.4f}")
```



From the output of the plot above, we see that there is a flat region when the alpha value is small, suggesting that the ridge regression model behaves similarly to a basic linear regression model. Over time, the MAE value increases as the alpha value increases, implying that these 2 variables are proportionate to each other. We also see that the best alpha and best MAE values identified from the plot are 0.0010 and 0.1400, respectively.

Part B - Finding best alpha value

The best alpha value is extracted from all the values of alpha that were initially present in “alphaRidge”. The following steps show how it was done.

1) The alpha value that corresponds to the minimum test Mean Absolute Error (MAE) is extracted using the *np.argmin()* function. The function, as stated in the previous part, gets the index position of the best value to be used on another list and finds the best alpha value.
2) The best MAE value is extracted from “test_scores_ridge”, using the same *np.argmin()* function, which corresponds to the best alpha value.

```
# Best alpha value
best_alpha_ridge_coarse = alphaRidge[np.argmin(test_scores_ridge)]
print(f"Best alpha (coarse): {best_alpha_ridge_coarse:.4f}")

# Best score (negative MAE)
best_score_ridge_coarse = test_scores_ridge[np.argmin(test_scores_ridge)]
print(f"Best test MAE (coarse): {best_score_ridge_coarse:.2f}")
```

```
Best coarse alpha: 0.0010
Best coarse test MAE: 0.1400
```

From the output above the best alpha values are 0.0010 and 0.1400, which are similar to the values in the previous part. The alpha value of 0.001 gives the lowest MAE, representing a weak regularisation. Hence, this suggests that the ridge regression model does not significantly improve the performance, compared to linear regression.

3.2.3. Fine tune by taking a closer range of alpha based on the previous result.

The codes and outputs for this section are split into 4 parts - Setting the alpha values in part A, Applying the ridge regression and plotting the features in part B, finding the best alpha values in part C, and evaluating the performance of the model on the test data in part D.

Part A - Setting the alpha values

1) The “best_alpha_ridge_coarse” variable contains the best alpha value from fine tuning.
2) A finer search range is created around the best value from the initial step. The range has 2 bounds, as defined below:
   Lower bound = 0.1 * alpha-value

Upper bound = 10 * alpha-value
This helps in narrowing the search region instead of wasting computing resources.

3) There are 50 alpha values that are evenly spaced and are generated using *np.linspace(start, stop, step)*.
Start = lower bound of alpha values
Stop = upper bound of alpha values
Step = 50

```
# Take a smaller range of alpha to test
fineTuneAlphaValue = np.linspace(best_alpha_ridge_coarse * 0.1,
                                 best_alpha_ridge_coarse * 10, 50)
print(f"Fine alpha range: {fineTuneAlphaValue[0]:.4f} to {fineTuneAlphaValue[-1]:.4f}")
```

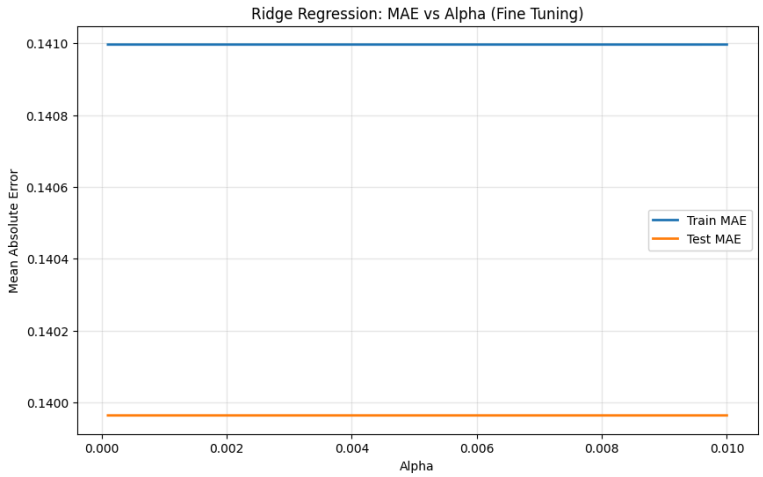
```
Fine alpha range: 0.0001 to 0.0100
```

The output above shows the range of alpha values from 0.0001 to 0.0100. These values have been fine-tuned and centred around the best alpha value. This suggests that there is a stronger regularisation that could be present among the features.

Part B - Applying the ridge regression and plotting the features

1) 2 lists, “train_scores_ridge_fine” and “test_scores_ridge_fine”, are created to store the MAE values for both training and testing data respectively. For each alpha value, a MAE value is stored in the mentioned lists.
2) The for loop iterates through the fine-tuned alpha values and the regression model will be created for each of them. The models are then scaled on the training data (X).
3) The predictions are done on the training and testing sets with the features to give the training and testing MAE values, using the *.mean_absolute_error()* and *.predict()* functions.
4) A line plot is created to show the MAE and alpha values, where the alpha values are on the x-axis and the MAE values are on the y-axis. The plot is used to determine how the model performance changes while the strength of regularisation changes.
5) The best alpha value is then selected using the index where there is a minimum value for the test MAE. These values are then displayed by extracting the best fine-tuned alpha value and its corresponding test MAE value.

```python
# Applying Ridge regression
train_scores_ridge_fine = []
test_scores_ridge_fine = []

for alpha in fineTuneAlphaValue:
    ridge = Ridge(alpha=alpha)
    ridge.fit(X_train_scaled, y_train)
    train_scores_ridge_fine.append(mean_absolute_error(y_train, ridge.predict(X_train_scaled)))
    test_scores_ridge_fine.append(mean_absolute_error(y_test, ridge.predict(X_test_scaled)))


    # Plot train and test scores against alpha
    plt.figure(figsize=(10, 6))
    plt.plot(fineTuneAlphaValue, train_scores_ridge_fine, label='Train MAE', linewidth=2)
    plt.plot(fineTuneAlphaValue, test_scores_ridge_fine, label='Test MAE', linewidth=2)
    plt.xlabel('Alpha')
    plt.ylabel('Mean Absolute Error')
    plt.title('Ridge Regression: MAE vs Alpha (Fine Tuning)')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

    best_idx_fine = np.argmin(test_scores_ridge_fine)

    # Best alpha value
    best_alpha_ridge = alphas_ridge_fine[best_idx_fine]
    print(f"Best alpha (fine): {best_alpha_ridge:.6f}")

    # Best score (negative MAE)
    best_test_mae_ridge = test_scores_ridge_fine[best_idx_fine]
    print(f"Best test MAE (fine): {best_test_mae_ridge:.6f}")
```




```
Best alpha (fine): 0.000100
Best test MAE (fine): 0.139964
```

The outputs above show that the curves for train and test MAE are flat, suggesting that there is no significant change when the alpha values increase. This represents a stable model having no changes in performance, hence it means that the features do not have any impact on the model.

We can also say that there is no overfitting of the data on the model since the training and testing MAE values are close to each other.

Part C - Finding the best alpha values

1) The optimal alpha value is found from the fine-tuning of the values. This is done using the *Ridge()* package and the "alpha" attribute controls the regularisation to fit the data and generalise well to new and unseen data.
2) The scaled training data is fitted to the model using the *.fit()* function to learn the coefficients with the regularisation.
3) The model coefficients are extracted by firstly creating a DataFrame and mapping each feature to its coefficient. Then, the values are sorted in descending order, using the *.sort_values()* function.
4) The top 10 ridge coefficients are displayed using the *.head(10)* function and the values are rounded off to 4 decimal places using *.round()*.

```
# Set best alpha for Ridge regression
# Fit the Ridge model to get the coefficients of the fitted model
ridge_final = Ridge(alpha=best_alpha_ridge)
ridge_final.fit(X_train_scaled, y_train)
```

```
# Show the coefficients for each feature
ridge_coef_df = pd.DataFrame({
    'Feature': X_train.columns,
    'Ridge_Coefficient': ridge_final.coef_
}).sort_values('Ridge_Coefficient', key=abs, ascending=False)

print("Top 10 Ridge Coefficients:")
print(ridge_coef_df.head(10).round(4))
```

```
Top 10 Ridge Coefficients:
            Feature  Ridge_Coefficient
8             hp_kW             0.2433
5             Gears             0.1140
6               age            -0.1000
13  Displacement_cc            -0.0861
2                km            -0.0638
4              Fuel             0.0528
16        cons_comb            -0.0505
17     age_km_ratio            -0.0422
14        Weight_kg             0.0379
11  Upholstery_type             0.0208
```

From the output above, we see that "hp_kW" is the most dominant feature with a positive impact in the dataset. The "Gear" feature is the next

positive impact where more gears usually have a higher price. Similarly, the “Fuel” also shows that if the fuel prices increase, the price of the car increases as well.

Based on this, we can say that the ridge regression model could control multicollinearity and retain all the features in the dataset.

Part D - Evaluate the Ridge model on the test data

1) The model is then used to predict on unseen and new scaled data using the *.predict()* function. This helps in the evaluation of generalisation of the model to the dataset.
2) The MAE - mean absolute error) *mean_absolute_error()*, RMSE - root mean squared error *(np.sqrt(mean_squared_error())* and R^2 score, *r2_score()*, is calculated using their appropriate functions. The functions are used for the following:
   MAE: used for checking robustness and seeing if the model is interpretable or not.
   RMSE: used in penalising larger errors more than MAE
   R^2: to check the proportion of variance of the model.

```
y_pred_ridge_test = ridge_final.predict(X_test_scaled)
ridge_test_mae = mean_absolute_error(y_test, y_pred_ridge_test)
ridge_test_rmse = np.sqrt(mean_squared_error(y_test, y_pred_ridge_test))
ridge_test_r2 = r2_score(y_test, y_pred_ridge_test)

print(f"Ridge Final Model Performance:")
print(f"Test MAE: {ridge_test_mae:.4f}")
print(f"Test RMSE: {ridge_test_rmse:.4f}")
print(f"Test R²: {ridge_test_r2:.4f}")
```

```
Ridge Final Model Performance:
Test MAE: 0.1400
Test RMSE: 0.1753
Test R²: 0.8078
```

The results of the model performance are as shown above.

The test MAE has a value of 0.14 is a low value which suggests that the ridge regression model has a strong accuracy when predicting the target variable. Similarly, the RMSE value of 0.1753, although low, could suggest that there are some large extremes that exist but not enough to influence the target variable.

The R^2 score has a value of 0.8078 where the model has a percentage of 81% accuracy. This means that the model is able to select good features with effective regularisation happening in the model.

3.3. Lasso Regression Implementation

3.3.1. Define a list of random alpha values

1) Similar to section 3.2.1, a list of alpha values are created using the *np.logspace(-4, 1, 50)* function, where 50 alpha values are evenly spaced where the alphas are logarithmic values that are generated within the (-4, 1) linear space.
2) The range of alpha values are displayed in an ascending order with the formatting from 6 to 4 decimal places.

```
# List of alphas to tune for Lasso regularisation
alphas_lasso_coarse = np.logspace(-4, 1, 50)
print(f"Lasso coarse alpha range: {alphas_lasso_coarse[0]:.6f} to {alphas_lasso_coarse[-1]:.4f}")
```

```
Lasso coarse alpha range: 0.000100 to 10.0000
```

From the output above, we see that the lasso alpha values range is from 0.0001 to 10.0000, and these will be used during hyperparameter tuning.

Alpha = 0.0001 means that lasso regression will behave like the baseline linear regression where coefficients are not penalised. This could result in overfitting of the model and it being unable to generalise to new and unseen data.

Alpha = 10 means that there is very strong regularisation on the model where the coefficients will be penalised heavily (most of the coefficients will be 0 for feature selection). This could result in underfitting of the model due to the lack of sufficient data to train on.

3.3.2. Apply Lasso Regularisation and find the best value of alpha from the list

Part A - Initialise Model

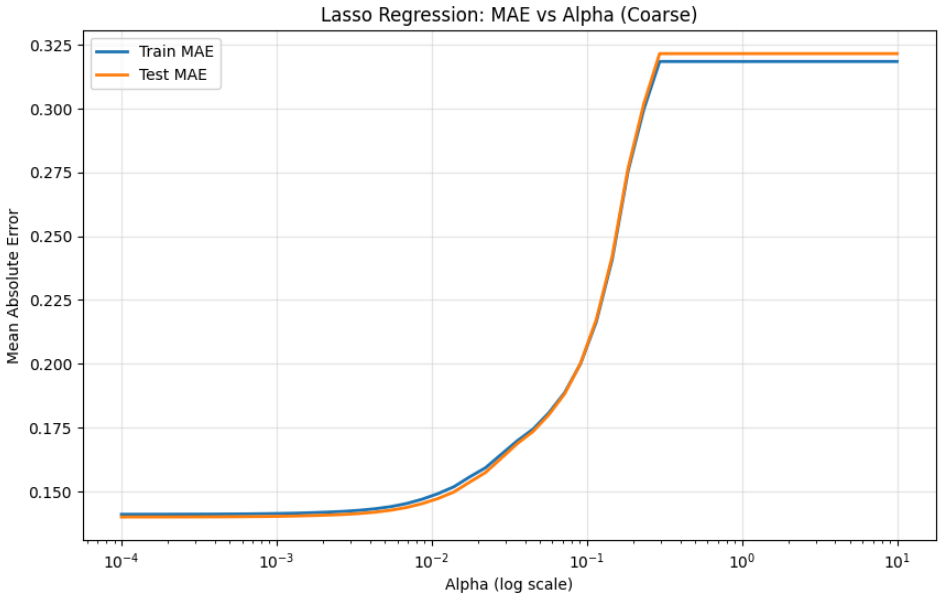
The Lasso Regression model is created and the scores, from training and testing, are plotted against the respective alpha values.

1) 2 empty lists are created, namely “train_scores_lasso_coarse” and “test_scores_lasso_coarse” to store the train and test MAE values respectively. These will be used to compare the performance with different alpha values.
2) The for loop iterates over each alpha value “alphas_lasso_coarse” from the previous section.
3) The lasso regression model is initialised with the following attributes in *Lasso()*:
   “alpha=alpha”, to see the regularisation strength
   “max_iter=5000”, for optimisation
   “random_state=42” to make reproducible results on random data.

4) The model is trained on the scaled data using the *.fit()* function and produces predicted values. Then, the MAE is calculated to find the difference between the actual and predicted values. This value is then inserted into the “train_scores_lasso_coarse” list using *.append()*. Similarly, the MAE is calculated for the test data and appended to the “test_scores_lasso_coarse” list.
5) A canvas of size 10x6 is created and 2 logarithmic graphs are plotted on it using *plt.semilogx()* - one being the train scores and the other being test scores. The MAE, on the y-axis, is plotted against the alpha values, which exist on the x-axis. A legend for these 2 plots is created to identify which line belongs to train MAE and test MAE, using *plt.legend()*.

```
# Initialise Lasso regression model
train_scores_lasso_coarse = []
test_scores_lasso_coarse = []

for alpha in alphas_lasso_coarse:
    lasso = Lasso(alpha=alpha, max_iter=5000, random_state=42)
    lasso.fit(X_train_scaled, y_train)
    train_scores_lasso_coarse.append(mean_absolute_error(y_train, lasso.predict(X_train_scaled)))
    test_scores_lasso_coarse.append(mean_absolute_error(y_test, lasso.predict(X_test_scaled)))
```

```
# Plot train and test scores against alpha
plt.figure(figsize=(10, 6))
plt.plot(alphas_lasso_fine, train_scores_lasso_fine, label='Train MAE', linewidth=2)
plt.plot(alphas_lasso_fine, test_scores_lasso_fine, label='Test MAE', linewidth=2)
plt.xlabel('Alpha')
plt.ylabel('Mean Absolute Error')
plt.title('Lasso Regression: MAE vs Alpha (Fine Tuning)')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```



From the graph, we see that at low alpha values, both the train and test MAE values are low and similar to one another. This suggests that the model is able to generalise well on random data. While the alpha value increases, both train and test MAE values also rise proportionally implying that there is underfitting of the model with insufficient data provided to it.

At a high alpha value, there is a flat line which suggests that there was too much regularisation done on the model where most of the coefficients are 0 and hence having insufficient data for the model to train on.

Part B - Best Alpha Value from the Model

The best alpha value is then calculated to find the one that suits the model.

1) The best alpha value is retrieved from the "test_scores_lasso_coarse" list using the *np.argmin()* function. The function returns the index of the smallest value present in the list, where the smallest value is the one with the least MAE value.
2) The lowest test MAE that has been achieved across all alpha values is extracted using the *np.argmin()* function on the "test_scores_lasso_coarse" which works to give the index position of where the value is in the list.

```
# Best alpha value
best_alpha_lasso_coarse = alphas_lasso_coarse[np.argmin(test_scores_lasso_coarse)]
print(f"Best alpha (coarse): {best_alpha_lasso_coarse:.6f}")

# Best score (negative MAE)
best_score_lasso_coarse = test_scores_lasso_coarse[np.argmin(test_scores_lasso_coarse)]
print(f"Best test MAE (coarse): {best_score_lasso_coarse:.6f}")
```

```
Best alpha (coarse): 0.000100
Best test MAE (coarse): 0.139981
```

We see that the outputs of best Alpha and MAE values are 0.0001 and 0.139981, respectively.

Where the best alpha is 0.0001, it shows that it was the smallest alpha value that was tested and that a weak regularisation works best for the dataset. If the regularisation was much stronger, it would lead to more errors being produced during prediction.

Where the best test MAE is 0.139981, shows that it is the lowest mean absolute error of prediction on completely new and unseen data. It also means that the actual prediction and estimated prediction have a difference of 0.14 arbitrary units.

3.3.3. Fine tune by taking a closer range of alpha based on the previous result.

This section is split into 5 parts where: Part A is “Selecting alpha values”, Part B is “Tuning Lasso Parameters and Plotting”, Part C is “Find the best alpha values”, Part D is “Set the best alpha value & Check the Coefficients” and Part E is “Evaluate the Performance of the Model”.

Part A - Selecting alpha values

The alpha values are being selected again to find a better regularization strength and use it especially when there are small improvements that were not shown.

1) The “best_alpha_lasso_coarse” is taken from the previous section where the best alpha value was 0.0001.
2) The boundaries of the regularisation values are created where:
   the lower bound is 0.5 of the best alpha value (50% of it), for a weaker regularisation to be done on the data, and
   the upper bound is 2 of the best alpha value (200% of it), for a strong regularisation to be done on the data.
3) The *np.linspace(...., 50)* function creates 50 evenly-spaced alpha values that are within the boundaries of alpha values created in the previous step.

```
# List of alphas to tune for Lasso regularization
alphas_lasso_fine = np.linspace(best_alpha_lasso_coarse * 0.5,
                                best_alpha_lasso_coarse * 2, 50)
print(f"Lasso fine alpha range: {alphas_lasso_fine[0]:.6f} to {alphas_lasso_fine[-1]:.6f}")
```

```
Lasso fine alpha range: 0.000050 to 0.000200
```

From the output above, we see that the range of alpha values are from 0.00005 to 0.0002. The fine-tuning done will help in identifying any improvements made to small MAE values compared to a sudden or large change in performance of the model.

Part B - Tuning Lasso Parameters and Plotting

Tuning of the lasso parameters helps in finding the alpha value with lowest error in generalisation while the model is stable.

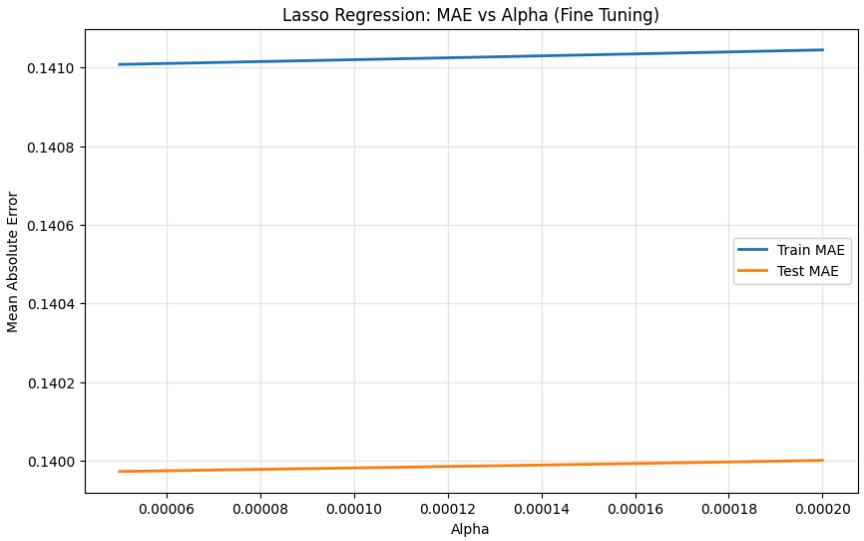
1) 2 new lists are created, named “train_scores_lasso_fine” and “test_scores_lasso_fine”, to store the train and test MAE values for each fine-tuned alpha value.
2) The for loop iterates through each fine-tuned alpha value and initialises a lasso regression model using Lasso(). The attributes specified are:
   “alpha=alpha”, to see the regularisation strength
   “max_iter=5000”, the increased iterations help in increasing

optimisation
"random_state=42" to make reproducible results on random data.

3) The model is then trained on scaled training data using the *.fit()* model, for regularization. The train and test MAE values are calculated using *mean_absolute_error()* and are then appended into their respective lists using the *.append()* function.
   The train MAE values are used to measure how well the model fits onto the data, and the test MAE values are used to evaluate the performance of the mode..

```
# Tuning Lasso hyperparameters
train_scores_lasso_fine = []
test_scores_lasso_fine = []

for alpha in alphas_lasso_fine:
    lasso = Lasso(alpha=alpha, max_iter=5000, random_state=42)
    lasso.fit(X_train_scaled, y_train)
    train_scores_lasso_fine.append(mean_absolute_error(y_train, lasso.predict(X_train_scaled)))
    test_scores_lasso_fine.append(mean_absolute_error(y_test, lasso.predict(X_test_scaled)))
```

```
# Plot train and test scores against alpha
plt.figure(figsize=(10, 6))
plt.plot(alphas_lasso_fine, train_scores_lasso_fine, label='Train MAE', linewidth=2)
plt.plot(alphas_lasso_fine, test_scores_lasso_fine, label='Test MAE', linewidth=2)
plt.xlabel('Alpha')
plt.ylabel('Mean Absolute Error')
plt.title('Lasso Regression: MAE vs Alpha (Fine Tuning)')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

best_idx_lasso_fine = np.argmin(test_scores_lasso_fine)
best_alpha_lasso = alphas_lasso_fine[best_idx_lasso_fine]
print(f"Best fine alpha: {best_alpha_lasso:.6f}")
print(f"Best fine test MAE: {test_scores_lasso_fine[best_idx_lasso_fine]:.6f}")
```



The graph above shows how the MAE changes for both the train and test sets while the fine-tuned alpha value changes as well.

We see that the train MAE line (blue) is flat throughout all alpha values, suggesting that the model has correctly fit the train dataset and it is able to generalise well to new and unseen data. Any small changes in the alpha value does not affect the performance of the model.

We also see that the test MAE line (orange) is flat initially but increases slightly when the alpha values are also increasing. This suggests that while the alpha values increase, the model's generalisation to the data is slightly worse than the model in training.

Part C - Find the Best Alpha Values

The best alpha value will be used for the training and evaluation of the final model. This is done as the value will avoid overfitting or underfitting of the model.

1) The "test_scores_lasso_fine" has the test MAE values for the fine-tuned alpha values. The *np.argmin()* function returns the index of the smallest MAE value and this index position is used to get the alpha value with the lowest test error, which is then used in the final model.
2) The fine-tuned alpha is printed/displayed with 6 decimal places to see the value.
3) The minimum MAE value is also extracted using *np.argmin()* to represent the best generalisation of the model that was achieved during the fine-tuning process.

```
# Best alpha value
best_alpha_lasso_final = alphas_lasso_fine[np.argmin(test_scores_lasso_fine)]
print(f"Best alpha (fine): {best_alpha_lasso_final:.6f}")

# Best score (negative MAE)
best_score_lasso_final = test_scores_lasso_fine[np.argmin(test_scores_lasso_fine)]
print(f"Best test MAE (fine): {best_score_lasso_final:.6f}")
```

```
Best alpha (fine): 0.000050
Best test MAE (fine): 0.139972
```

The alpha = 0.0005 is the best yet smallest alpha value after fine-tuning, implying that there is a weak regularisation applied on the dataset. Similarly, the MAE = 0.139972 is the lowest mean absolute prediction error value that was achieved in the test data. It can be considered as a slight improvement from the previous value, hence showing that the fine-tuning was an effective method.

Part D - Set the best alpha value & Check the Coefficients

1) The final lasso regression model is initialised using Lasso() with the following attributes:
   “alpha=alpha”, to see the regularisation strength
   “max_iter=5000”, helping in optimisation of the model
   “random_state=42” to ensure reproducible results.
2) The model is then trained using the scaled training data (X_train_scaled) and the target values (y_train). The model is able to learn the coefficients of each feature and apply regularisation on it.
3) The model’s results, where the coefficients are not 0, are displayed which indicates that these are the features that are retained by the model.

```
# Set best alpha for Lasso regression
lasso_final = Lasso(alpha=best_alpha_lasso_final, max_iter=5000, random_state=42)
lasso_final.fit(X_train_scaled, y_train)

# Fit the Lasso model on scaled training data
# Get the coefficients of the fitted model
print("Final Lasso model fitted")
print(f"Number of non-zero coefficients: {(lasso_final.coef_ != 0).sum()}")
```

```
Final Lasso model fitted
Number of non-zero coefficients: 17
```

From the output above, we see that the count of non-zero coefficients is 17. This suggests that the model has retained these features as they are considered “important” in the model’s training whereas the remaining features are removed. Hence, the “feature selection” process of the model has improved the model’s ability to generalise to new and unseen data.

Similarly, a coefficient analysis is done as shown below.

1) A DataFrame map, using *pd.DataFrame()*, is created to map each feature’s name to its corresponding coefficient value from the model.
2) The *.sort_values()* function sorts the features in descending order of its coefficient. The features with a bigger coefficient have a higher influence on the predictions.
3) The top 10 influential coefficients are displayed using *.head(10)* and are rounded to 4 decimal places for easy reading using *.round(4)*.
4) The code also displays the percentage of 0 coefficients, rounded off to 1 decimal place using *.round(1)*, to quantify the feature strength of the lasso regression model. The formula used to calculate the percentage is:
   (Sum of Coefficients == 0/Number of Coefficients == 0) * 100

```python
# Check the coefficients for each feature
lasso_coef_df = pd.DataFrame({
    'Feature': X_train.columns,
    'Lasso_Coefficient': lasso_final.coef_
}).sort_values('Lasso_Coefficient', key=abs, ascending=False)

print("Top 10 Lasso Coefficients:")
print(lasso_coef_df.head(10).round(4))
print(f"\nZero coefficients: {(lasso_final.coef_ == 0).sum()} ({(lasso_final.coef_ == 0).sum()/len(lasso_final.coef_)*100:.1f}%)")
```

```
Top 10 Lasso Coefficients:
            Feature  Lasso_Coefficient
8             hp_kW             0.2430
5             Gears             0.1140
6               age            -0.1000
13  Displacement_cc            -0.0858
2                km            -0.0637
4              Fuel             0.0525
16        cons_comb            -0.0505
17    age_km_ratio            -0.0421
14       Weight_kg             0.0378
11  Upholstery_type             0.0207

Zero coefficients: 1 (5.6%)
```

From the output above, if the coefficient has a positive sign, it helps in increasing prediction, else if it has a negative sign, it decreases prediction.

We see that “hp_kW” has the highest and positive coefficient meaning that it has the strongest influence on the price prediction (target variable). Similarly, we see that “age” and “km” have negative coefficients which suggests that as these variables increase, the prediction done on the target variable decreases inversely.

The output also shows that a feature was removed due to its zero coefficient. This suggests that it is not a useful feature and the regularisation done on the model was small due to a weak alpha value.

Part E - Evaluate the Performance of the Model

1) The *.predict()* function helps to predict the price of the car. The model has already been trained on the scaled test dataset (X_test_scaled).
2) The Mean Absolute Error (MAE) is calculated with the help of *mean_absolute_error()*. It shows the difference between the actual and predicted values.
3) Similarly, the RMSE and R^2 score are calculated using the *np.sqrt(mean_absolute_error())* and *r2_score()* functions respectively. The RMSE helps to give a balanced view of the model as it heavily penalises the large coefficients. R^2 score measures the variance in “price” (target column). The range of the

scores are from 0 (no power or influence on the dataset) to 1 (a perfect prediction is made).

4) The results of MAE, RMSE and R^2 are printed, rounded up to 4 decimal places using the *.4f* attribute to make it easier for viewing and comparing.

```
# Evaluate the Lasso model on the test data
y_pred_lasso_test = lasso_final.predict(X_test_scaled)
lasso_test_mae = mean_absolute_error(y_test, y_pred_lasso_test)
lasso_test_rmse = np.sqrt(mean_squared_error(y_test, y_pred_lasso_test))
lasso_test_r2 = r2_score(y_test, y_pred_lasso_test)

print(f"Lasso Final Model Performance:")
print(f"Test MAE: {lasso_test_mae:.4f}")
print(f"Test RMSE: {lasso_test_rmse:.4f}")
print(f"Test R²: {lasso_test_r2:.4f}")
```

```
Lasso Final Model Performance:
Test MAE: 0.1400
Test RMSE: 0.1753
Test R²: 0.8078
```

The output shows that the MAE value is 0.14, meaning that the model is able to perform consistently and provide mostly accurate predictions with minor errors.

The RMSE value of 0.1753 shows that there are some large values in the features although they do not overly influence the model. The R^2 score of the model is approximately 81% indicating that it is able to predict well despite the regularisation done on it.

3.4. Regularisation Comparison and Analysis

3.4.1. Compare the evaluation metrics for each model.

The evaluation metrics are compared to understand which model has performed better than the others. The code is explained below.

1) A DataFrame is created using *pd.DataFrame()* to map each model to its scores. It helps in comparing the models' performance and also makes it easily understandable.
2) The names of the models are defined within the DataFrame. There were 3 types of models built - Linear Regression, Ridge Regression and Lasso Regression.
3) In the DataFrame, "Test_MAE" contains the MAE values for each model. Between these 3 models, the lowest MAE means that the model has the best predictive accuracy.
4) Similarly, "Test_RMSE" and "Test_R2" contain the root mean squared error and R^2 score for the models respectively. RMSE

heavily penalises large coefficients and having a high R^2 score means that the model has been trained well.

5) The numerical values are rounded off to 4 decimal places using *.round(4)*.

```
# Compare metrics for each model
metrics_df = pd.DataFrame({
    'Model': ['Linear Regression', 'Ridge Regression', 'Lasso Regression'],
    'Test_MAE': [maeLrTest, ridge_test_mae, lasso_test_mae],
    'Test_RMSE': [rmseLrTest, ridge_test_rmse, lasso_test_rmse],
    'Test_R2': [r2LinearRegression, ridge_test_r2, lasso_test_r2]
}).round(4)

print("Model Performance Comparison:")
print(metrics_df.to_string(index=False))
```

```
Model Performance Comparison:
            Model  Test_MAE  Test_RMSE  Test_R2
Linear Regression      0.14     0.1753   0.8078
 Ridge Regression      0.14     0.1753   0.8078
 Lasso Regression      0.14     0.1753   0.8078
```

From the output above, we see that the 3 models have the same MAE of 0.14, indicating a similar prediction being done by the 3 models.

The 3 models also have the same RMSE value of 0.1753, meaning that the models handle the large coefficients similarly and that none of them are influenced/affected by any outliers.

They also have the same R^2 score of 0.8078, suggesting that the 3 models have a variance of approximately 81%. The features are a strong fit for the model, providing meaningful insights for those models to train on.

3.4.2. Compare the coefficients for the three models. Also visualise a few of the largest coefficients and the coefficients of features dropped by Lasso.

This section is split into 2 parts - Part A is “Compare the coefficients for the three models” and Part B is “Feature Elimination"

Part A - Compare the Coefficients for the three models

The comparison is done to show how regularisation affects the importance of features and causes shrinkage of important data that might not be useful for the model to train on.

1) A DataFrame is created using *pd.DataFrame()* to compare the input feature coefficients of the 3 models. The *.columns* and *.coef_* attributes are used to list all the feature names and store the learned coefficients of the models, respectively.
2) Only the Ridge and Lasso regression models are selected for analysis. The *.abs()* function converts the coefficients into positive values so that only those values can be compared. The *.max(axis=1)* function is used to pick the largest coefficient present in both models.
3) Similarly, *.nlargest(15)* is used to select the top 15 features that are influential on the model, and *.index* returns the index position of the row.
4) The coefficients of the top 15 features are then rounded off to 4 decimal places using *.round(4)* and the *.to_string()* is used for easier reading and comparison without any truncation of the values.

```
# Compare highest coefficients and coefficients of eliminated features
coef_comparison = pd.DataFrame({
    'Feature': X_train.columns,
    'LinearRegression': lr_model.coef_,
    'Ridge': ridge_final.coef_,
    'Lasso': lasso_final.coef_
})
```

```
# Top 15 absolute coefficients across all models
top_features = coef_comparison[['Ridge', 'Lasso']].abs().max(axis=1).nlargest(15).index
top_coef_comparison = coef_comparison.loc[top_features].round(4)
print("Top 15 Coefficients Comparison:")
print(top_coef_comparison.to_string())
```

```
Top 15 Coefficients Comparison:
            Feature  LinearRegression   Ridge   Lasso
8             hp_kW            0.2433  0.2433  0.2430
5             Gears            0.1140  0.1140  0.1140
6               age           -0.1000 -0.1000 -0.1000
13  Displacement_cc           -0.0861 -0.0861 -0.0858
2                km           -0.0638 -0.0638 -0.0637
4              Fuel            0.0528  0.0528  0.0525
16        cons_comb           -0.0505 -0.0505 -0.0505
17     age_km_ratio           -0.0422 -0.0422 -0.0421
14        Weight_kg            0.0379  0.0379  0.0378
11  Upholstery_type            0.0208  0.0208  0.0207
9    Inspection_new            0.0083  0.0083  0.0082
3              Type           -0.0066 -0.0066 -0.0066
0         body_type           -0.0060 -0.0060 -0.0059
15      Drive_chain            0.0022  0.0022  0.0022
10       Paint_Type           -0.0018 -0.0018 -0.0017
```

The output shows that the coefficients of the features are almost identical across the 3 models, with the Lasso regression model having a slight difference when compared to Ridge and Linear regression models. This shows that there is no multicollinearity between the features and that regularization has not affected the model significantly (it can be also considered as negligible).

We can also interpret that the fine-tuned alpha values are also small, hence the Ridge and Lasso regression models behaved similarly to the Linear regression model.

The positive and negative signs on the coefficient indicate the direction of influence of the feature on the model. We see that “hp_kW” had the highest positive correlation indicating its importance for the predictions being made by the model.

Part B - Feature Elimination

1) The “coef_comparison” DataFrame contains the names of the features and their respective coefficients from the 3 models. A Boolean condition is done to see which coefficients in the Lasso regression model are exactly 0. The *.tolist()* function converts that feature into a list.
2) The *len(lasso_eliminated)* function counts the number of features where the coefficient is 0 and prints it.
3) The final line of code is used to display only the first 10 features that have been removed from the dataset. This ensures that the output is easily readable when printed.

```
# Lasso eliminated features
lasso_eliminated = coef_comparison[coef_comparison['Lasso'] == 0]['Feature'].tolist()
print(f"\nFeatures eliminated by Lasso ({len(lasso_eliminated)} features):")
print(lasso_eliminated[:10], "..." if len(lasso_eliminated) > 10 else "")
```

```
Features eliminated by Lasso (1 features):
['Previous Owners']
```

From the output above, we see that the Lasso regression model slightly shrinked the coefficients to the point where one of the features, “Previous_Owners” was removed since it was determined as not useful for the model. Hence, this shows that effective feature selection was done in a way that does not affect the performance of the model.

4. Conclusion & Key Takeaways

4.1. Conclude with outcomes and insights gained

- From the start, we see that the dataset has 15915 cars surveyed and 23 initial rows/features. Alongside, 3 models - Linear, Ridge and Lasso regression models were created to predict the prices of the cars.
- The heavy skewness of the distributions show a small number of highly-priced vehicles influencing the price of the car and the medium- to low-priced vehicles taking a major part of the distribution.
- This skewness was handled by capping the outliers using the Inter-Quartile Range (IQR) method. Feature encoding using LabelEncoder was also done to drop the features where their cardinality is high.
  High cardinality refers to a column with many unique values in relation to the total size of the dataset.
- Similarly, features with multicollinearity were also removed as their Variance Inflation Factor (VIF) is above 10 since it impacts the regression models.
- Overall, the ridge regression model uses its best alpha value of 0.0001 and the lasso regression model uses its best alpha value of 0.00005. With these alpha values, these 2 models performed similarly to the baseline linear regression model.
- From the results of the model, we see that “hp_kW” (engine power), gears, have a strong correlation with a domination over the dataset with different correlations of 0.68 and 0.59 respectively.
- Similarly, age and “km” (mileage) have a negative correlation and a negative impact on the price of the vehicle where the higher the usage, the lesser the price/cost of the car.
- From the outputs of the model, we see that the body size of the vehicle ranges from vans, sedans/wagons and compacts. This means that as the size of the vehicle increases, the price of the car also increases.
- Similarly, the fuel type also impacted the price of the car where vehicles with petrol and diesel have similar prices and influence the data and model more than the other types of fuel that were surveyed for the dataset.
- The different gear types, such as automatics, semi-automatics and manuals, also cause prices of the vehicles to increase.
- A proper train-test-split was done on the data in the 80/20 ratio to ensure fair evaluation on the dataset.
- The final results of the 3 models show that they behave and perform similarly to each other on test data, as shown below:
  Mean Absolute Error (MAE) of 0.14
  Root Mean Squared Error (RMSE) of 0.1753, and
  R^2 Score of 0.8078

- The above results suggest that the models were able to generalise quite well to the random and unseen test data without any signs of overfitting shown. The regularisation done on the model did not significantly improve the predictions as multicollinearity and outliers were already handled.
- The logarithmic transformation was done on the target variable, “price” column, which reduced the skewness because it improved the performance of the model and reduced the residuals present in the dataset.
- Hence, based on the outputs, it clearly shows that car prices are mostly influenced by their performance, usage, and vehicle configuration. The regression models used are enough to achieve good predictions with high accuracy without much regularisation applied.