Used Cars Price Prediction

Problem Definition

- There is a huge demand for used cars in the Indian Market today. As sales of new cars have slowed down in the recent past, the pre-owned car market has continued to grow over the past years and is larger than the new car market now. Cars4U is a budding tech start-up that aims to find footholes in this market.
- In 2018-19, while new car sales were recorded at 3.6 million units, around 4 million second-hand cars were bought and sold. There is a slowdown in new car sales and that could mean that the demand is shifting towards the pre-owned market. In fact, some car owners replace their old cars with pre-owned cars instead of buying new ones. The used car market is a very different beast with huge uncertainty in both pricing and supply. Keeping this in mind, the pricing scheme of these used cars becomes important in order to grow in the market.

The Context:

- The used car market in India is larger than the new car market and is experiencing significant growth despite a slowdown in new car sales. This shift highlights the increasing consumer preference for pre-owned vehicles. Understanding and addressing the pricing scheme for these used cars is critical for companies like Cars4U to capitalize on the market potential and gain a competitive edge.

The objective:

- The intended goal is to develop an accurate and reliable pricing model for used cars that reflects the current market dynamics. This model should help Cars4U offer competitive prices to both buyers and sellers, thereby increasing their market share and customer satisfaction.

The key questions:

- Which features (e.g., make, model, age, mileage, location, etc.) significantly affect the price?
- What are the distributions of the variables?
- Are variables correlated?
- Which type of model is best suit for the task?

The problem formulation:

- Develop a predictive pricing model: Create a machine learning model that predicts the price of a used car based on various attributes such as make, model, age, mileage, condition, location, and other relevant factors.

Data Dictionary

S.No. : Serial Number

Name : Name of the car which includes Brand name and Model name

Location : The location in which the car is being sold or is available for purchase (Cities)

Year : Manufacturing year of the car

Kilometers_driven : The total kilometers driven in the car by the previous owner(s) in KM

Fuel_Type : The type of fuel used by the car (Petrol, Diesel, Electric, CNG, LPG)

Transmission : The type of transmission used by the car (Automatic / Manual)

Owner : Type of ownership

Mileage : The standard mileage offered by the car company in kmpl or km/kg

Engine : The displacement volume of the engine in CC

Power : The maximum power of the engine in bhp

Seats : The number of seats in the car

New_Price : The price of a new car of the same model in INR 100,000

Price : The price of the used car in INR 100,000 (**Target Variable**)

Loading libraries

In [79]:
```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from statsmodels.formula.api import ols

import statsmodels.api as sm

from statsmodels.stats.outliers_influence import variance_inflation_factor
```

```
from sklearn.metrics import r2_score, mean_absolute_percentage_error, mean_a

from sklearn.model_selection import train_test_split, GridSearchCV

from statsmodels.stats.diagnostic import het_white

from statsmodels.compat import lzip

import statsmodels.stats.api as sms

import pylab

import scipy.stats as stats

from sklearn.model_selection import KFold

from sklearn.linear_model import Ridge

from sklearn.tree import DecisionTreeRegressor

from sklearn.ensemble import RandomForestRegressor

import sklearn

import warnings
warnings.filterwarnings("ignore")
```

In [80]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, cal
l drive.mount("/content/drive", force_remount=True).
```

Let us load the data

In [81]:
```
data = pd.read_csv('/content/drive/MyDrive/MIT course/Capstone Project/used_
```

Data Overview

- Observations
- Sanity checks

In [82]:
```
data.head()
```

Out[82]:

	S.No.	Name	Location	Year	Kilometers_Driven	Fuel_Type	Transmission	C
0	0	Maruti Wagon R LXI CNG	Mumbai	2010	72000	CNG	Manual	
1	1	Hyundai Creta 1.6 CRDi SX Option	Pune	2015	41000	Diesel	Manual	
2	2	Honda Jazz V	Chennai	2011	46000	Petrol	Manual	
3	3	Maruti Ertiga VDI	Chennai	2012	87000	Diesel	Manual	
4	4	Audi A4 New 2.0 TDI Multitronic	Coimbatore	2013	40670	Diesel	Automatic	

We can see all the information about a vehicle, there are missing values in 'New_price' variable.

In [83]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7253 entries, 0 to 7252
Data columns (total 14 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   S.No.              7253 non-null   int64  
 1   Name               7253 non-null   object 
 2   Location           7253 non-null   object 
 3   Year               7253 non-null   int64  
 4   Kilometers_Driven  7253 non-null   int64  
 5   Fuel_Type          7253 non-null   object 
 6   Transmission       7253 non-null   object 
 7   Owner_Type         7253 non-null   object 
 8   Mileage            7251 non-null   float64
 9   Engine             7207 non-null   float64
 10  Power              7078 non-null   float64
 11  Seats              7200 non-null   float64
 12  New_price          1006 non-null   float64
 13  Price              6019 non-null   float64
dtypes: float64(6), int64(3), object(5)
memory usage: 793.4+ KB
```

There are 14 variables total, 9 numerical, 5 categorical.

In [84]:
```
data.isnull().sum()
```

Out[84]:
```
S.No.                 0
Name                  0
Location              0
Year                  0
Kilometers_Driven     0
Fuel_Type             0
Transmission          0
Owner_Type            0
Mileage               2
Engine               46
Power               175
Seats                53
New_price          6247
Price              1234
dtype: int64
```

In [85]:
```
# Check percentage of missing values
(data.isnull().sum() / data.shape[0])*100
```

Out[85]:
```
S.No.                 0.000000
Name                  0.000000
Location              0.000000
Year                  0.000000
Kilometers_Driven     0.000000
Fuel_Type             0.000000
Transmission          0.000000
Owner_Type            0.000000
Mileage               0.027575
Engine                0.634220
Power                 2.412795
Seats                 0.730732
New_price            86.129877
Price                17.013650
dtype: float64
```

There are missing values in 'Engine', 'Power', 'Seats', 'New_Price', and 'Price'. \ New_Price has ~86% missing values, the column may not be useful at all. \ Price has ~17% missing values, this is the dependent variable.

In [86]:
```
data.shape
```

Out[86]:
```
(7253, 14)
```

The data has 7253 rows and 14 columns.

In [87]:
```
# Check for duplicates
data.duplicated().sum()
```

Out[87]:
```
0
```

There is no duplicated information.

In [88]:
```
# Checking descriptive statistics
```

```
data.describe(include = 'all').T
```

Out[88]:

	count	unique	top	freq	mean	std	m
S.No.	7253.0	NaN	NaN	NaN	3626.0	2093.905084	(
Name	7253	2041	Mahindra XUV500 W8 2WD	55	NaN	NaN	N
Location	7253	11	Mumbai	949	NaN	NaN	N
Year	7253.0	NaN	NaN	NaN	2013.365366	3.254421	1996
Kilometers_Driven	7253.0	NaN	NaN	NaN	58699.063146	84427.720583	17
Fuel_Type	7253	5	Diesel	3852	NaN	NaN	N
Transmission	7253	2	Manual	5204	NaN	NaN	N
Owner_Type	7253	4	First	5952	NaN	NaN	N
Mileage	7251.0	NaN	NaN	NaN	18.14158	4.562197	(
Engine	7207.0	NaN	NaN	NaN	1616.57347	595.285137	72
Power	7078.0	NaN	NaN	NaN	112.765214	53.493553	34
Seats	7200.0	NaN	NaN	NaN	5.280417	0.809277	2
New_price	1006.0	NaN	NaN	NaN	22.779692	27.759344	3.
Price	6019.0	NaN	NaN	NaN	9.479468	11.187917	0.

Observations

- There are missing values on some of the columns.
- The dependent variable (Price) has missing values.
- There are two types of variables, numeric and categorical.
- There are extreme values in some columns, like in kilometers driven where one value is of 6,500,000.
- Variables may not follow normal distributions.

Exploratory Data Analysis

- EDA is an important part of any project involving data.
- It is important to investigate and understand the data better before building a model with it.

In [89]:

```
# Function to plot a boxplot and a histogram along the same scale

def histogram_boxplot(data, feature, figsize = (12, 7), kde = True, bins = N
    """
```

```
    Boxplot and histogram combined

    data: dataframe
    feature: dataframe column
    figsize: size of figure (default (12,7))
    kde: whether to show density curve (default False)
    bins: number of bins for histogram (default None)
    """
    f2, (ax_box2, ax_hist2) = plt.subplots(
        nrows = 2,      # Number of rows of the subplot grid = 2
        sharex = True,  # x-axis will be shared among all subplots
        gridspec_kw = {"height_ratios": (0.25, 0.75)},
        figsize = figsize,
    )                   # Creating the 2 subplots
    sns.boxplot(
        data = data, x = feature, ax = ax_box2, showmeans = True, color = "v
    )                   # Boxplot will be created and a star will indicate t
    sns.histplot(
        data = data, x = feature, kde = kde, ax = ax_hist2, bins = bins, pal
    ) if bins else sns.histplot(
        data = data, x = feature, kde = kde, ax = ax_hist2
    )                   # For histogram
    ax_hist2.axvline(
        data[feature].mean(), color = "green", linestyle = "--"
    )                   # Add mean to the histogram
    ax_hist2.axvline(
        data[feature].median(), color = "black", linestyle = "-"
    )                   # Add median to the histogram
```

Univariate Analysis

In [90]:

```
# Create lists with numerical and categorical variable names.
num_variables = ['S.No.', 'Year','Kilometers_Driven','Mileage','Engine','Pow
cat_variables = ['Name','Location','Fuel_Type','Transmission', 'Owner_Type']
```

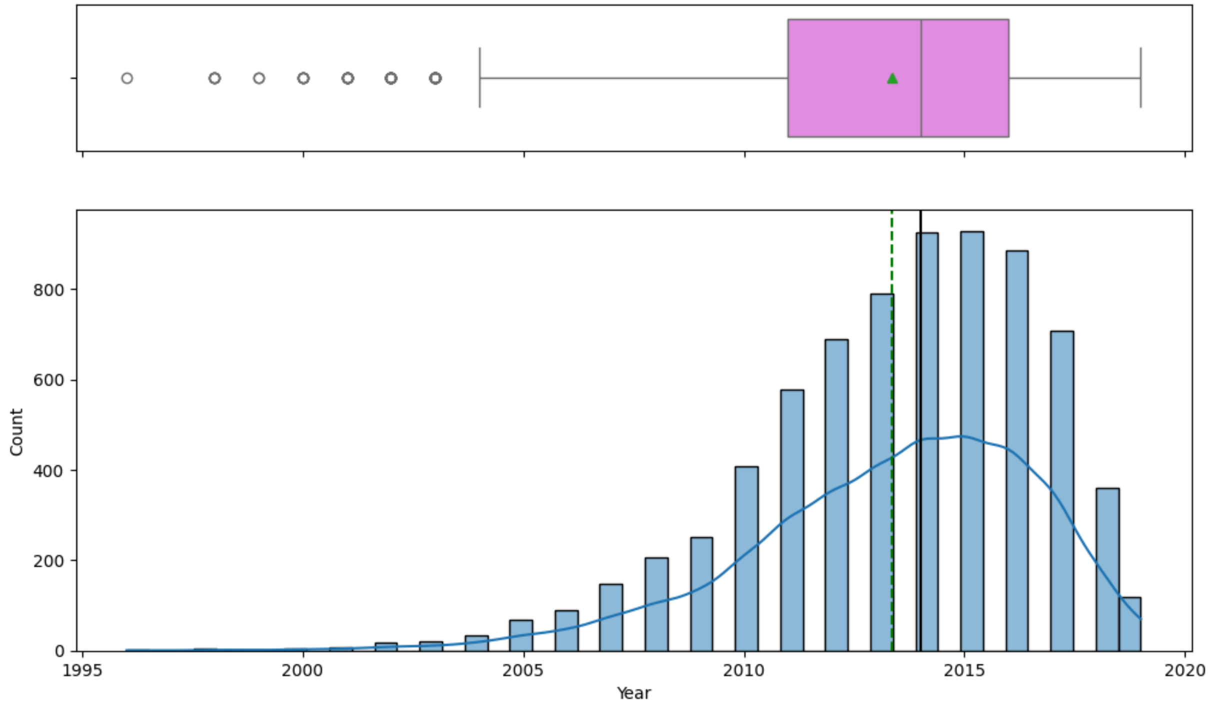
In [91]:

```
# Check unique values of Serial Number
data['S.No.'].value_counts()
data.drop('S.No.', axis = 1, inplace = True)
```

All the values in the serial number columns are different, no relevant information for predicting is provided.

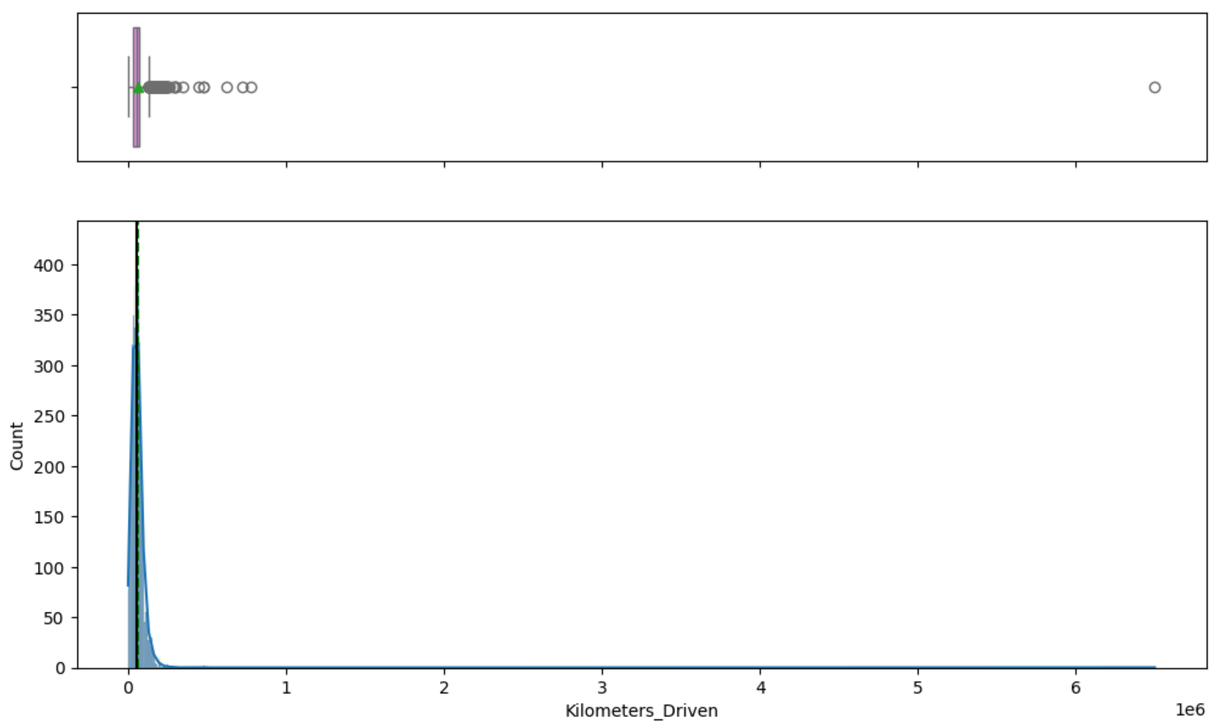
In [92]:

```
# Histogram and boxplot for year
histogram_boxplot(data, 'Year')
```



The data is a bit skewed to the left, but all values seem reasonable.

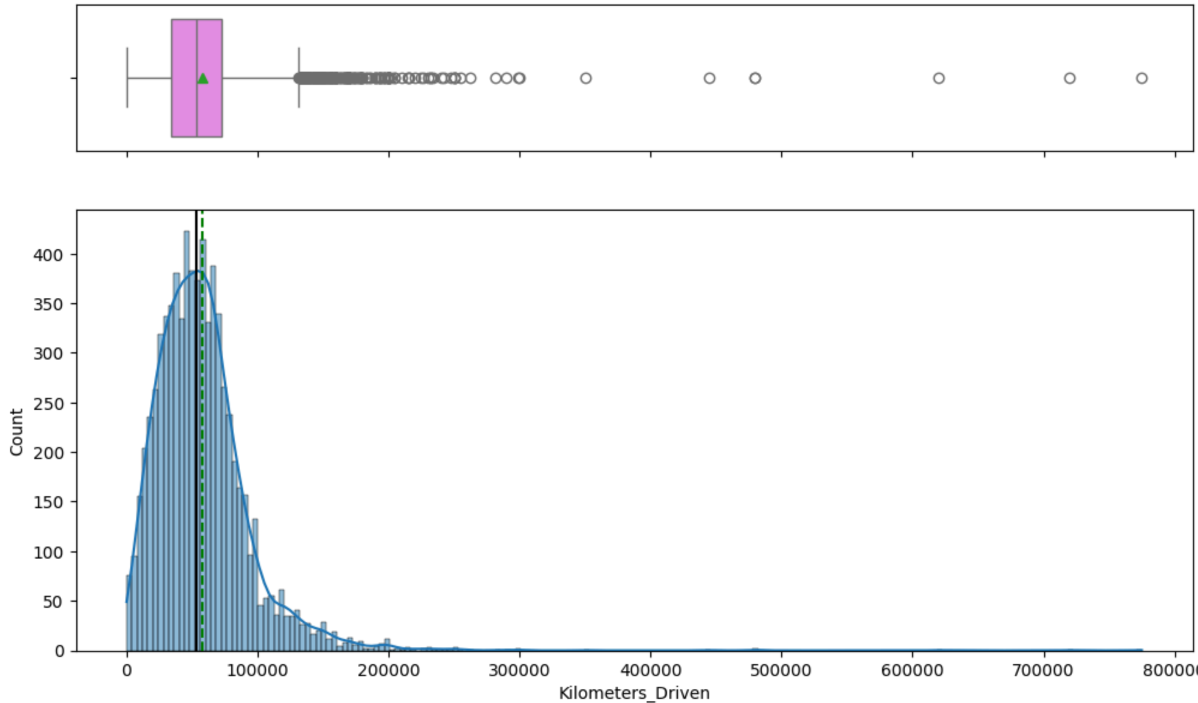
In [93]:
```
# Histogram and boxplot for Kilometers driven
histogram_boxplot(data, 'Kilometers_Driven')
```



The graph looks highly skewed to the right, it is unreasonable to think that a car still works after 6.5 million kilometers.

In [94]:
```
# Drop the row with the extreme value
data.drop(index = 2328, inplace = True)
```

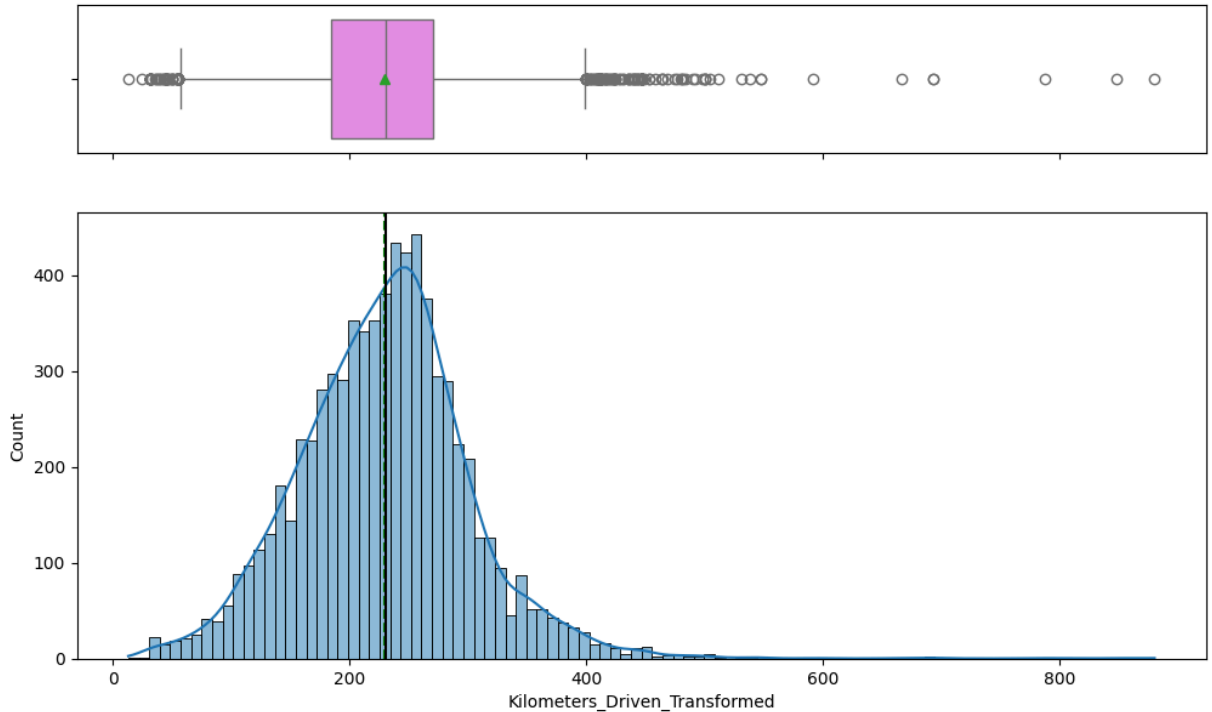
In [95]:
```
# Histogram and boxplot for Kilometers Driven
histogram_boxplot(data, 'Kilometers_Driven' )
```



The data is still skewed to the right, as there are less cars that have a high kilometer count, performing a log or sqrt transform might help normalize data.

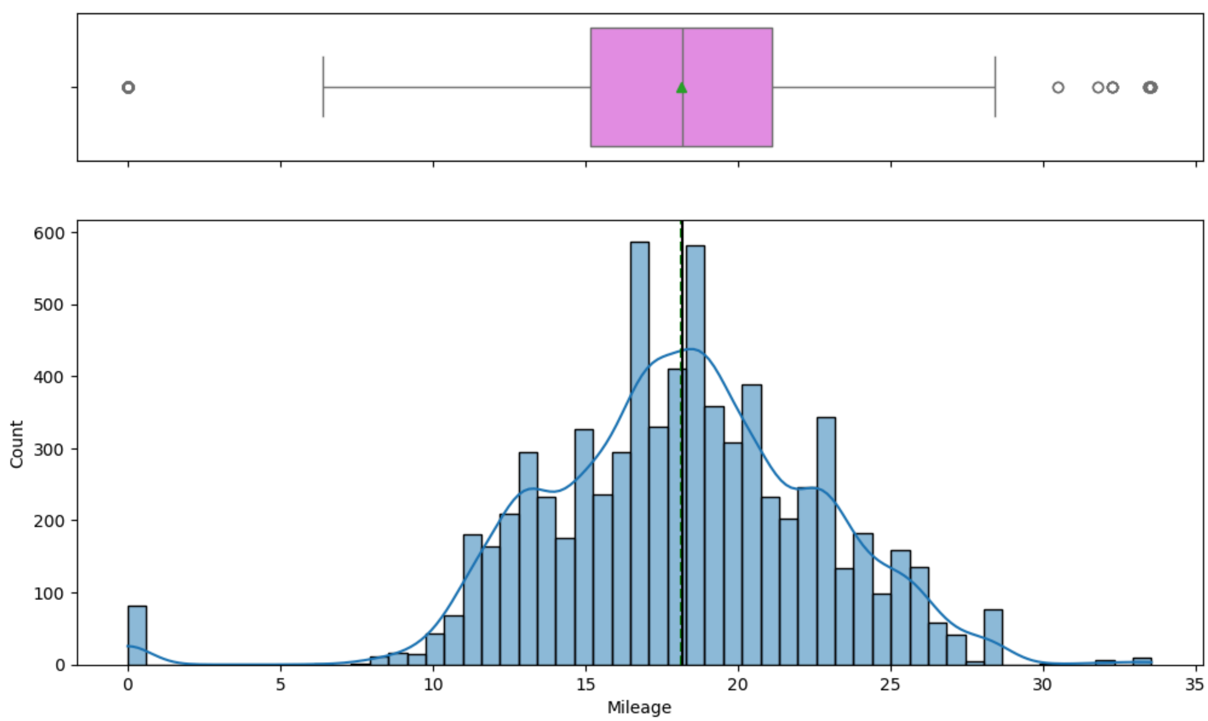
In [96]:
```
# Transform Kilometers_driven
data['Kilometers_Driven_Transformed'] = np.sqrt(data['Kilometers_Driven'])
```

In [97]:
```
# Plot the transformed variable
histogram_boxplot(data, 'Kilometers_Driven_Transformed', kde = True)
```



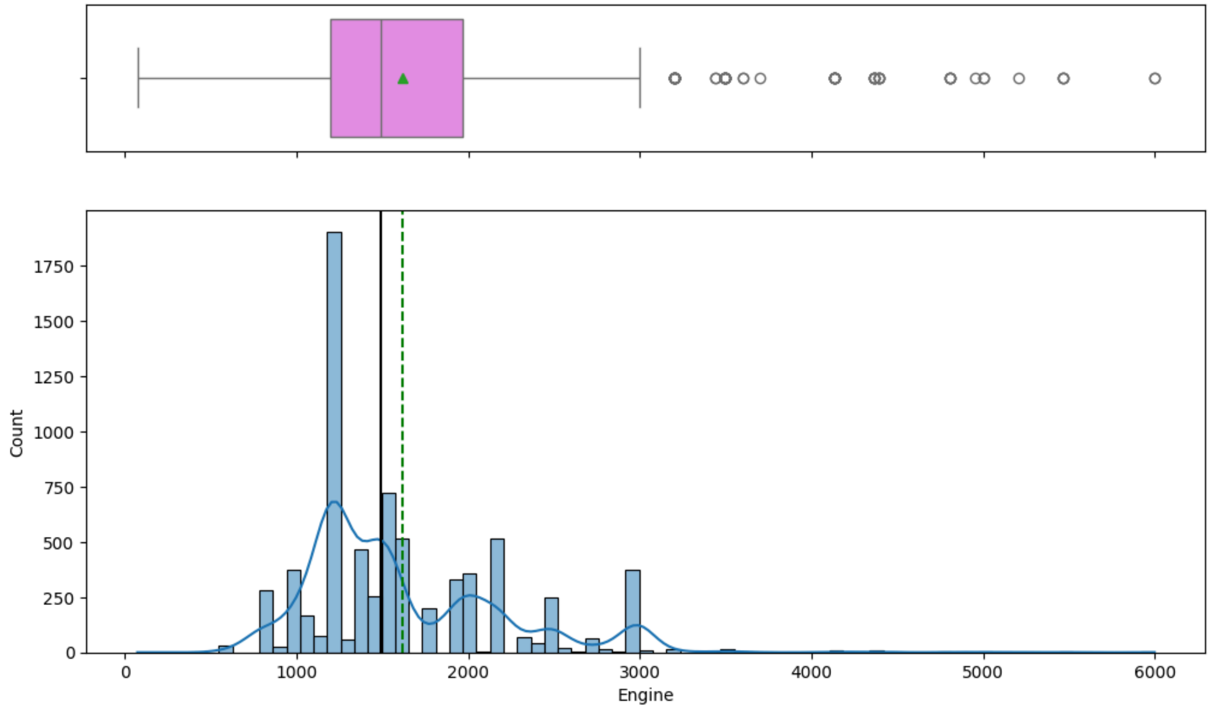
It is still slightly skewed, but it is better than before.

In [98]:
```
# Histogram and boxplot for Mileage
histogram_boxplot(data, 'Mileage')
```



The 'Mileage' variable has some outliers, however they are not extreme, most of them are proper values, except the 0's, since no car can have 0 mileage.

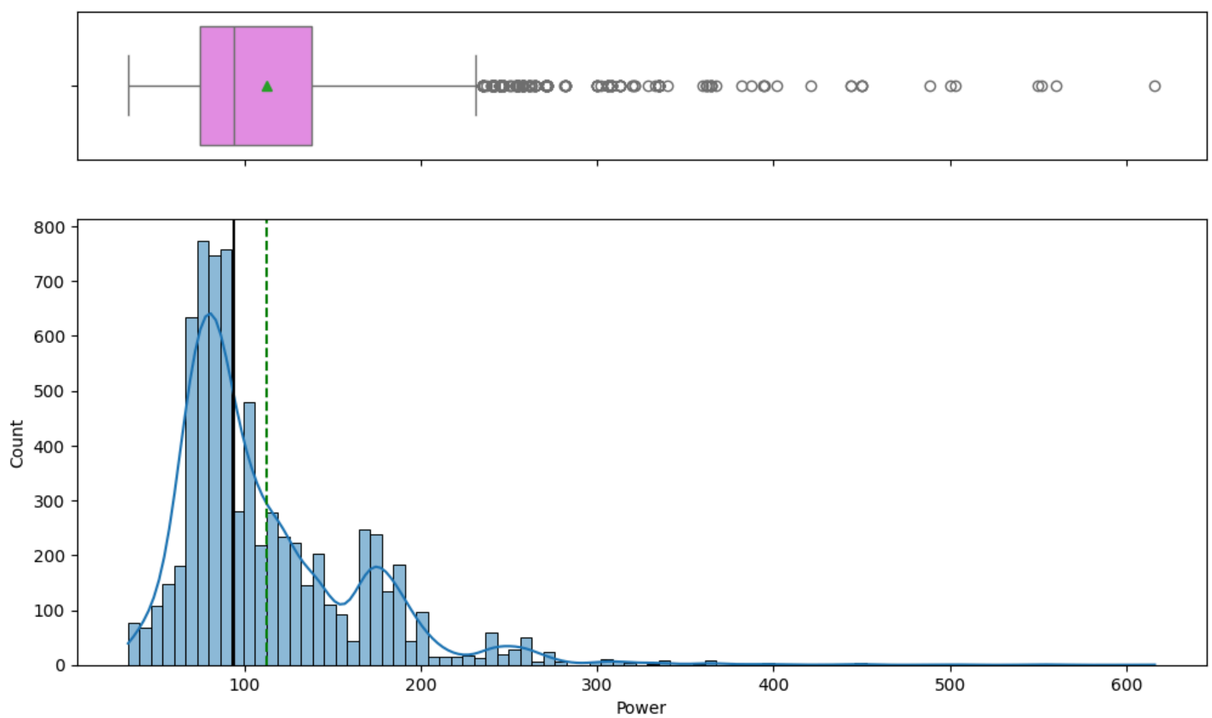
In [99]:
```
# Histogram and boxplot for Engine
histogram_boxplot(data,'Engine')
```



There are a few outliers in the data, it is skewed to the right.

In [100…

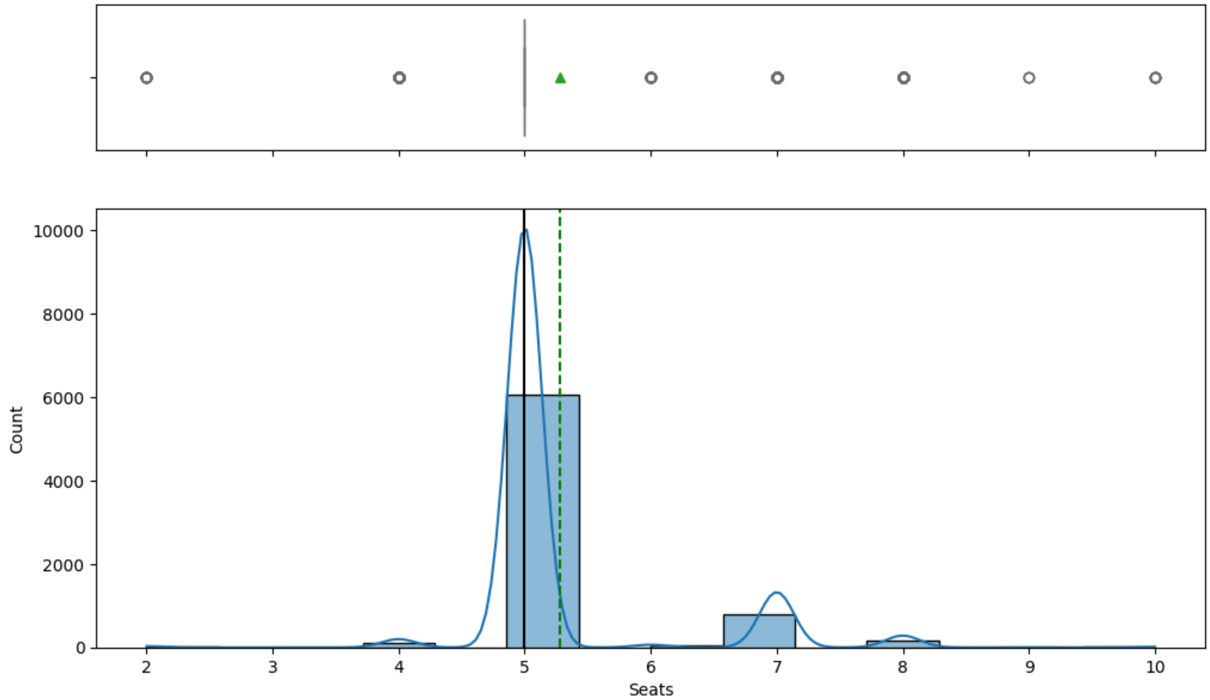
```
# Histogram and boxplot for power
histogram_boxplot(data, 'Power')
```



Another variable that is skewed, we also have some outliers in the upper quartile.

In [101…

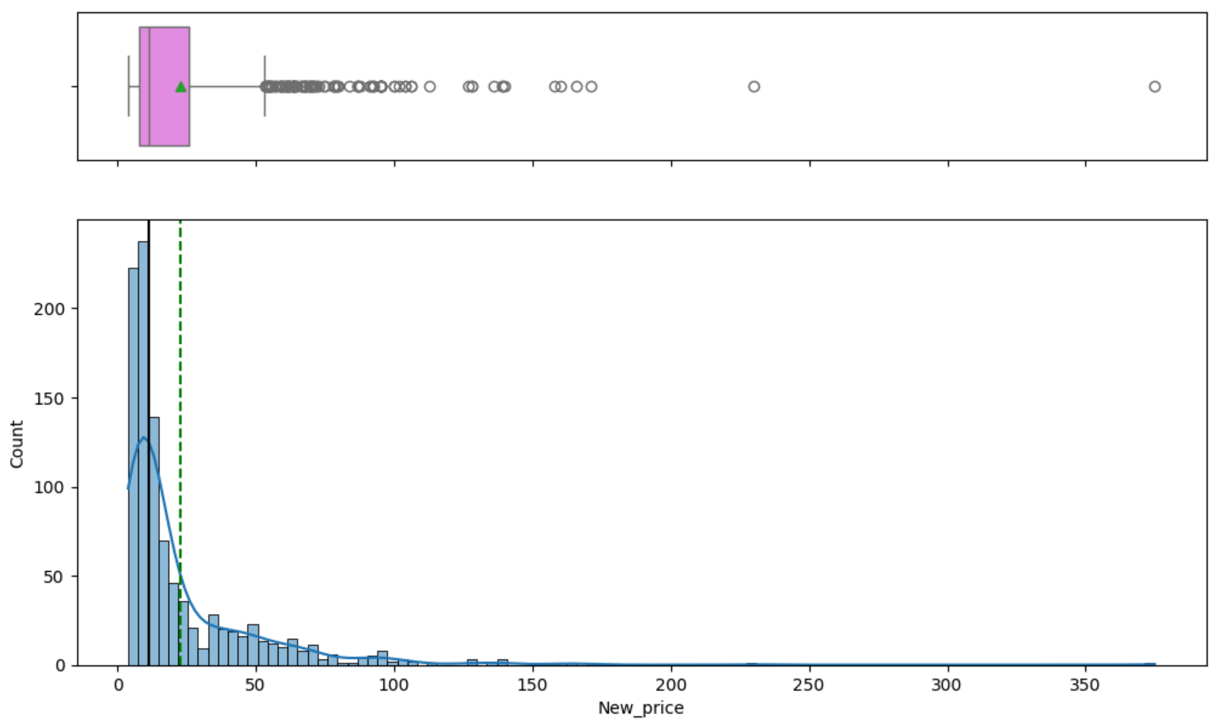
```
# Histogram and boxplot for seats
histogram_boxplot(data, 'Seats')
```



There is no particular distribution since the seats can only be an integer, all the values are proper values.

In [102…

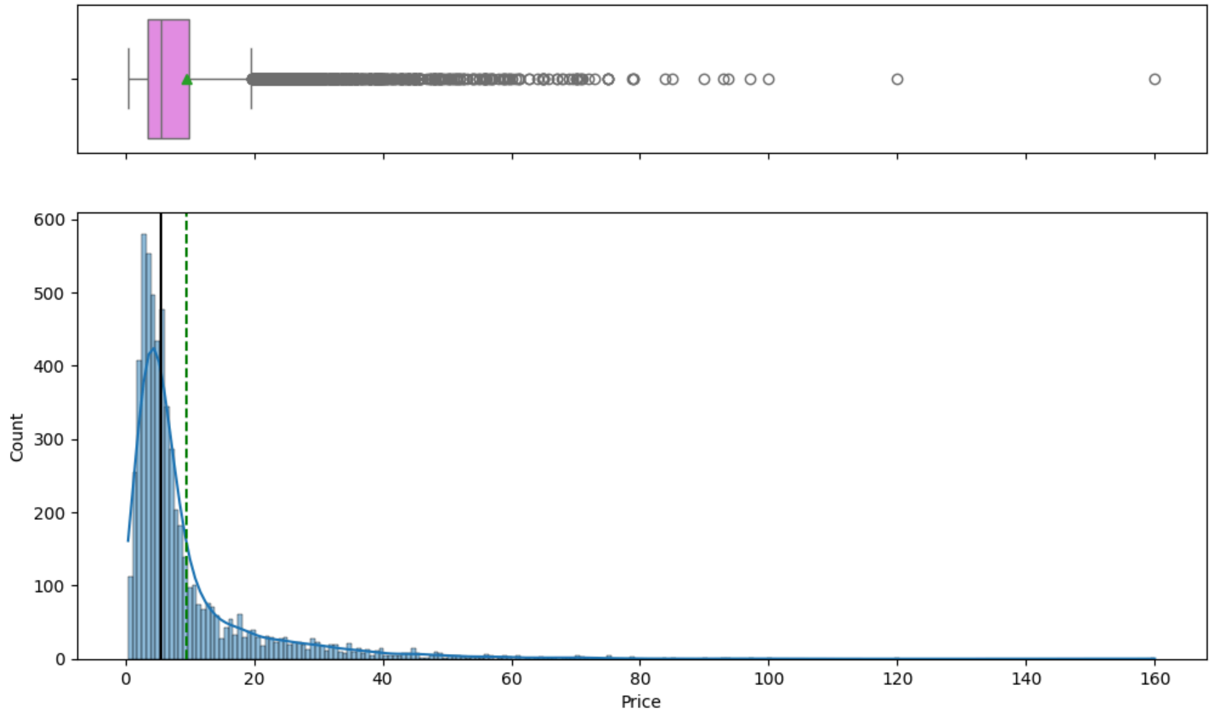
```
# Histogram and boxplot for new price
histogram_boxplot(data, 'New_price')
```



Data is skewed to the right, there are a lot of outliers.

In [103…

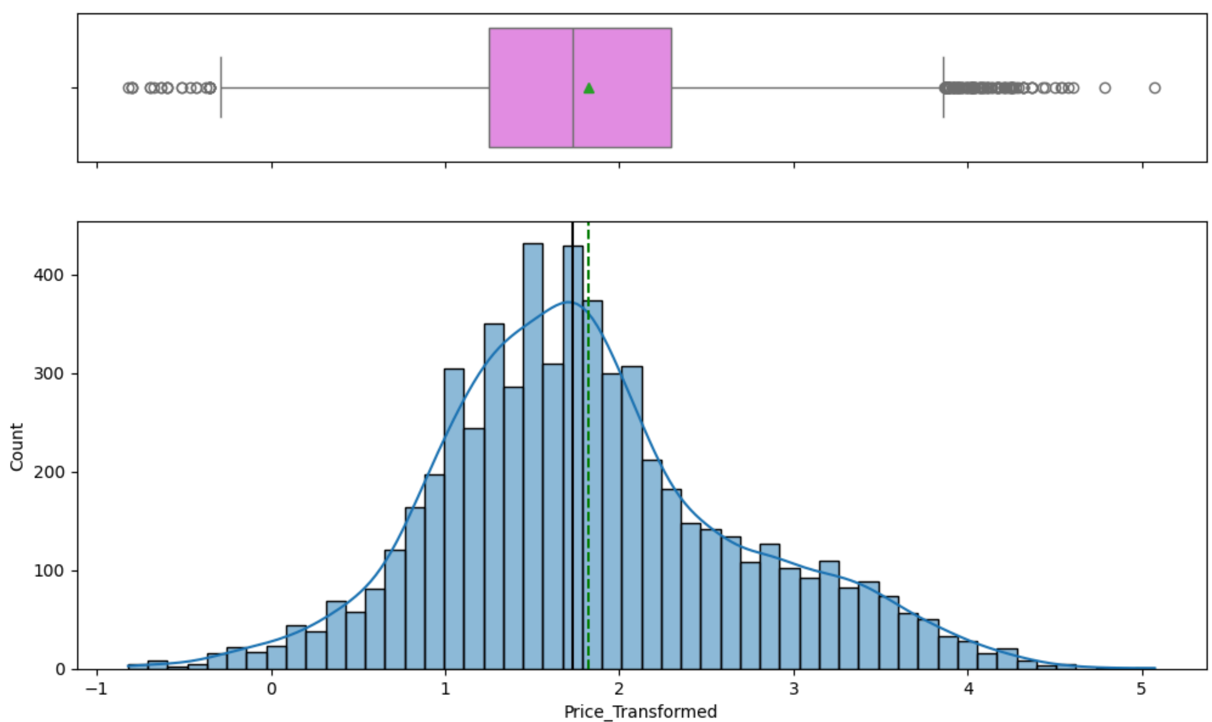
```
# Histogram and boxplot for price
histogram_boxplot(data, 'Price')
```



This is the dependent variable, it is skewed to the right and has a lot of outliers. Performing a log or sqrt transform might be useful.

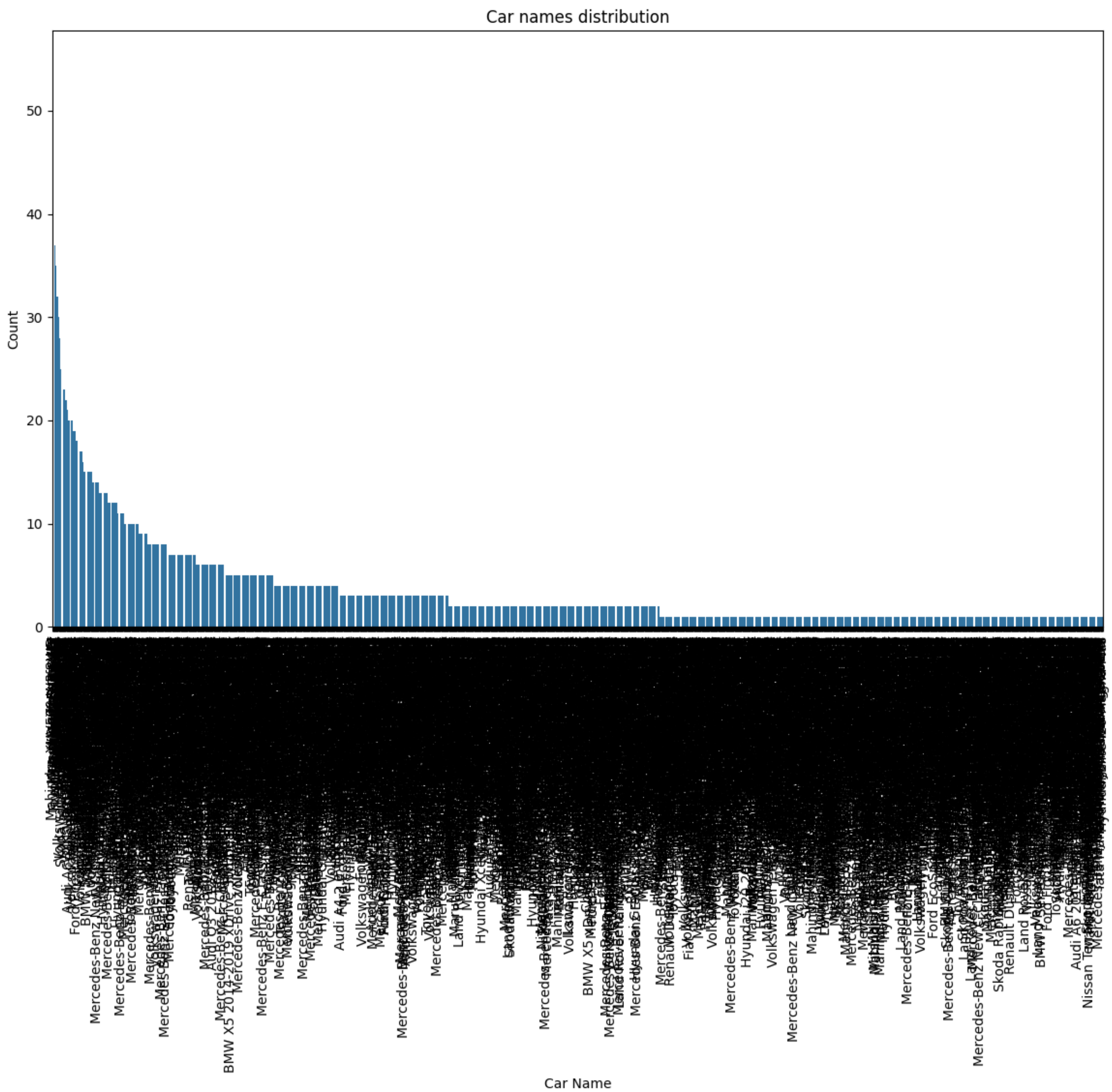
In [104…
```
data['Price_Transformed'] = np.log(data['Price'])
```

In [105…
```
# Histogram and boxplot for price
histogram_boxplot(data, 'Price_Transformed')
```



Now the data is more normally distributed

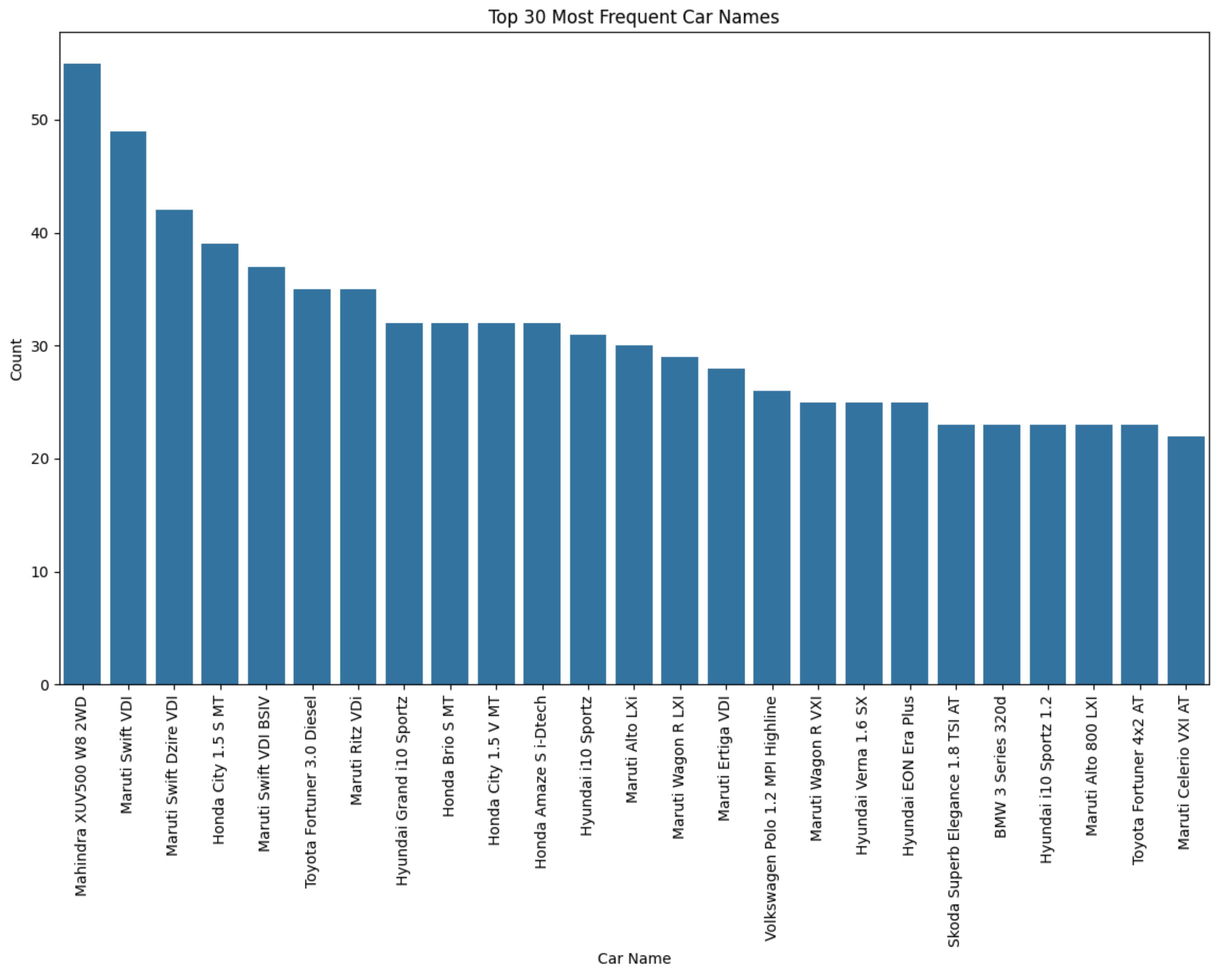
In [106…
```
# Plot the countplot for car names
plt.figure(figsize=(14, 8))
sns.countplot(x='Name', data=data, order=data['Name'].value_counts().index)
plt.title('Car names distribution')
plt.ylabel('Count')
plt.xlabel('Car Name')
plt.xticks(rotation = 90)
plt.show()
```



Some models are a lot more popular than others, this can give information about popularity and trends.

In [107…
```
top_car_names = data['Name'].value_counts().nlargest(25)

# Plot the countplot for the top 30 car names
plt.figure(figsize=(14, 8))
sns.countplot(x='Name', data=data, order=top_car_names.index)
plt.title('Top 30 Most Frequent Car Names')
plt.ylabel('Count')
```

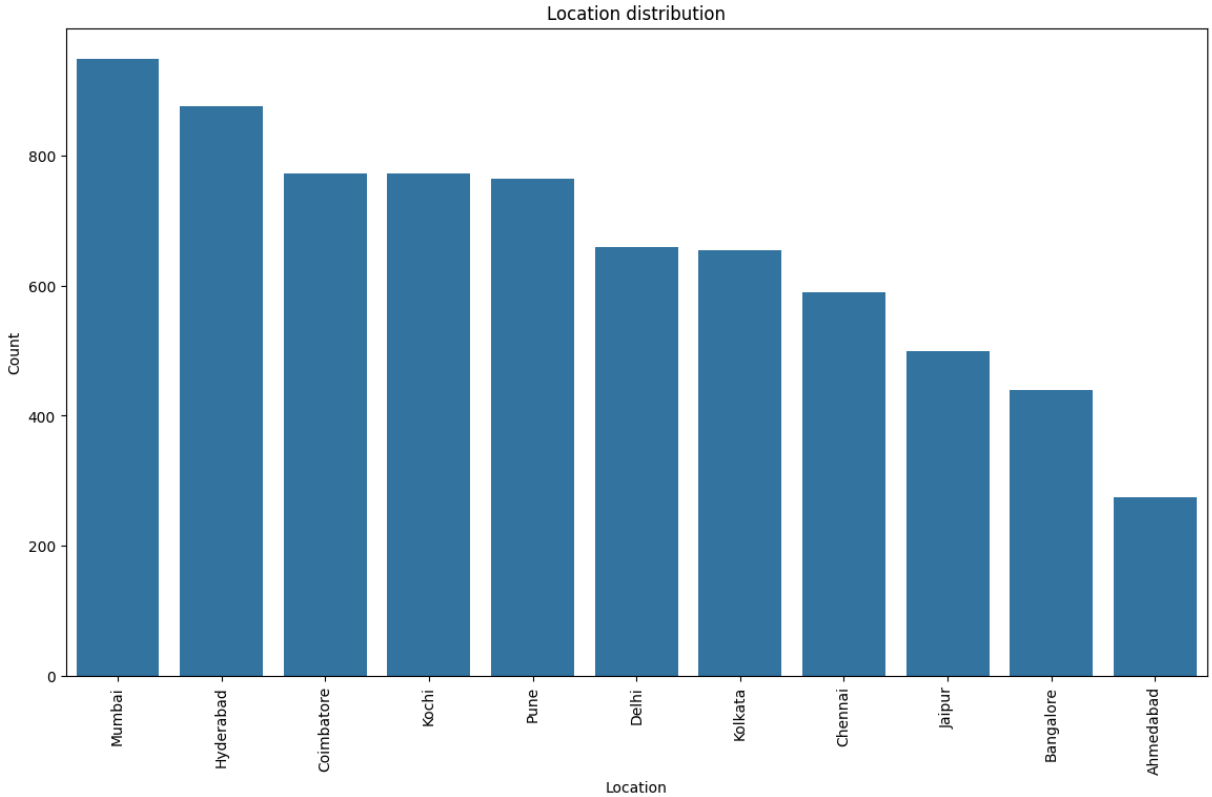
```
plt.xlabel('Car Name')
plt.xticks(rotation = 90)
plt.show()
```



These are the top 25 models, all of them have a frequency of over 20. We can see that hyundai is quite popular. Splitting this variable into multiple variables like Brand and model might be useful to be able to get more information.

In [108…

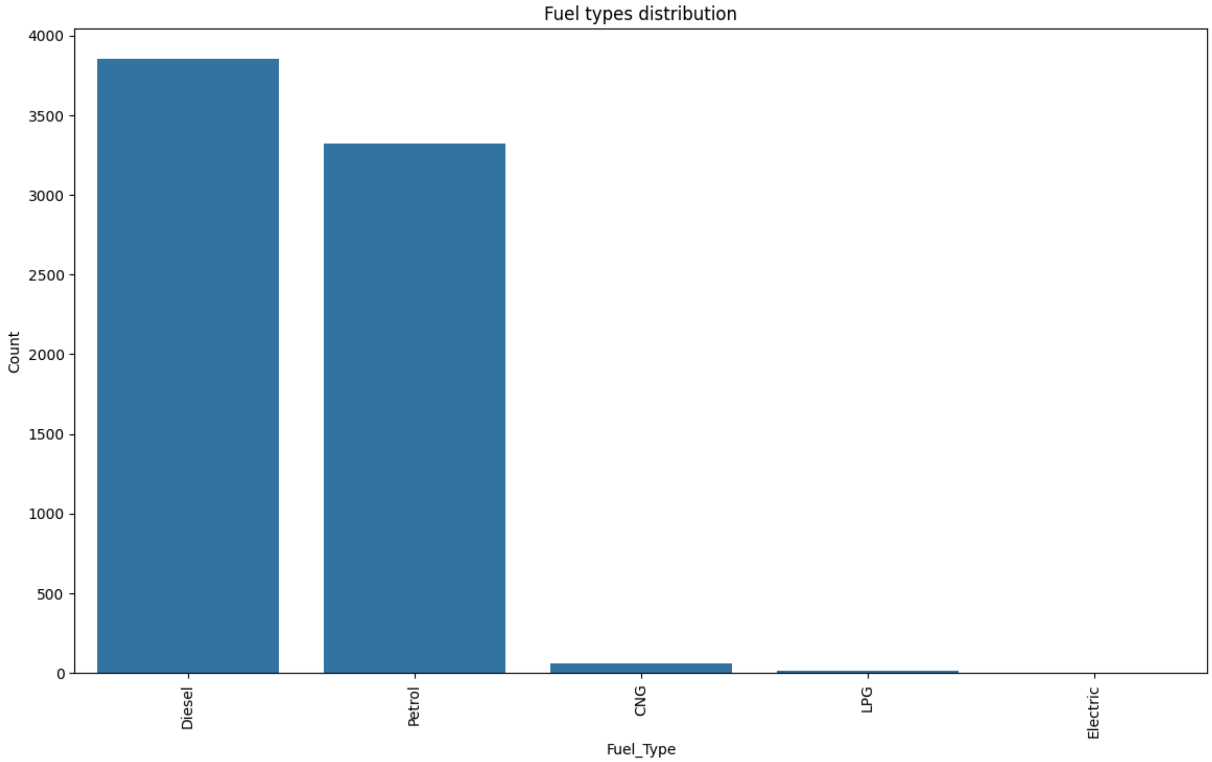
```
# Plot the countplot for location
plt.figure(figsize=(14, 8))
sns.countplot(x='Location', data=data, order=data['Location'].value_counts()
plt.title('Location distribution')
plt.ylabel('Count')
plt.xlabel('Location')
plt.xticks(rotation = 90)
plt.show()
```



There are not a lot of locations, just 11, with the largest being Mumbai and the lowest being Ahmedabad.

In [109…

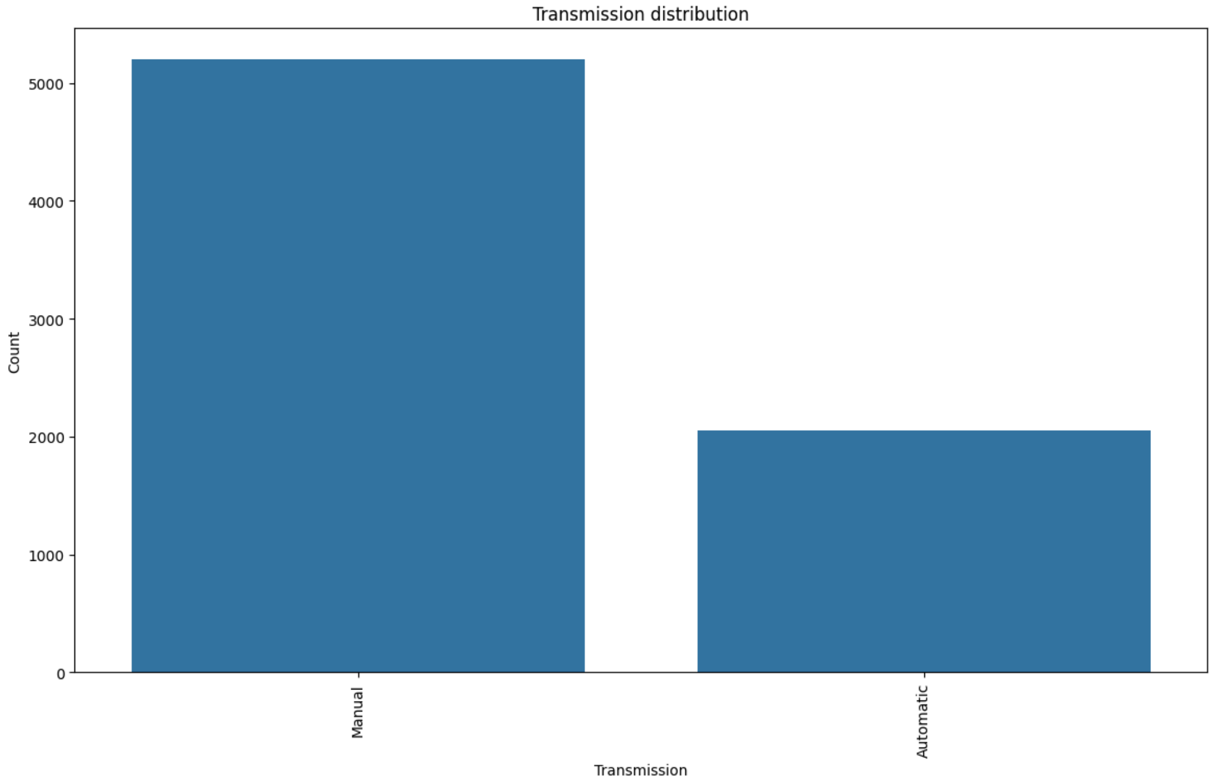
```
# Plot the countplot for car names
plt.figure(figsize=(14, 8))
sns.countplot(x='Fuel_Type', data=data, order=data['Fuel_Type'].value_counts
plt.title('Fuel types distribution')
plt.ylabel('Count')
plt.xlabel('Fuel_Type')
plt.xticks(rotation = 90)
plt.show()
```



The most common type of fuel is Diesel, the least common is Electric. There are only 5 types of fuel.

In [110…

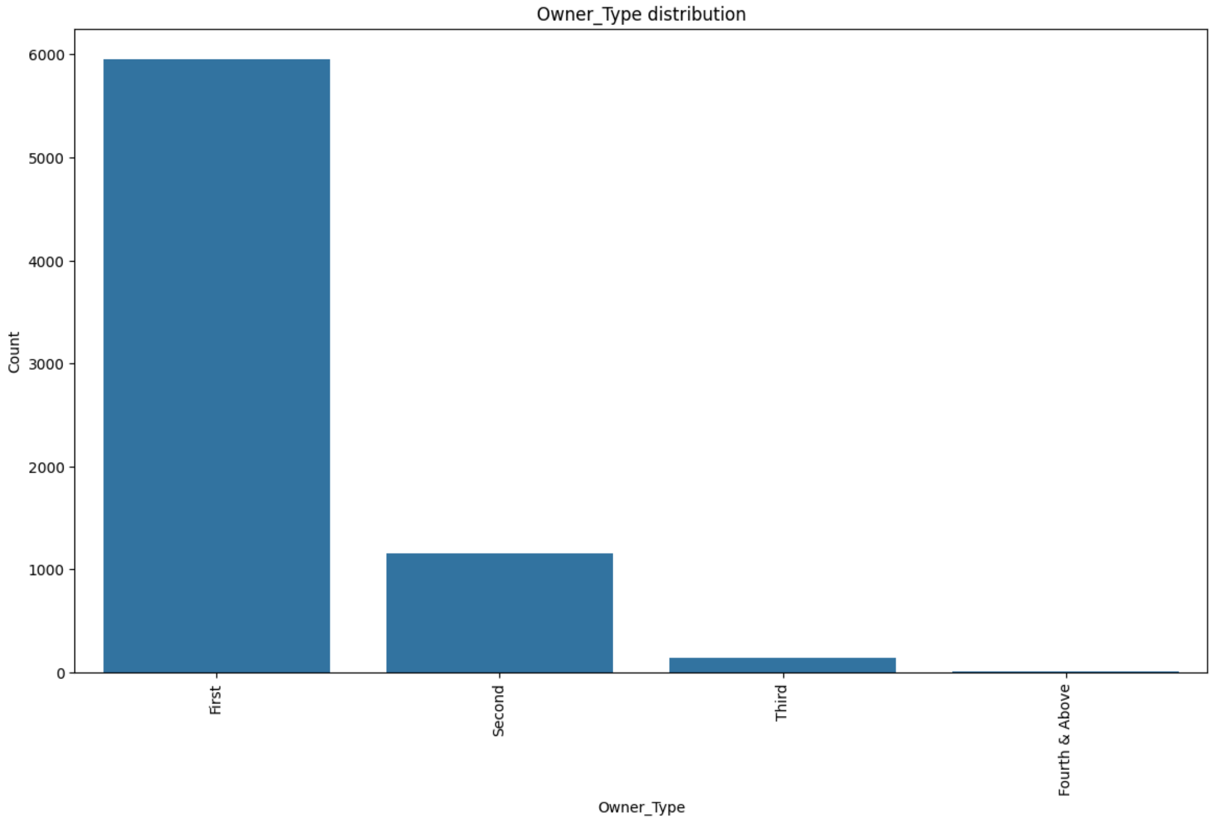
```
# Plot the countplot for car names
plt.figure(figsize=(14, 8))
sns.countplot(x='Transmission', data=data, order=data['Transmission'].value_
plt.title('Transmission distribution')
plt.ylabel('Count')
plt.xlabel('Transmission')
plt.xticks(rotation = 90)
plt.show()
```



There are only two types of transmission, with manual being the most popular.

In [111…

```
# Plot the countplot for car names
plt.figure(figsize=(14, 8))
sns.countplot(x='Owner_Type', data=data, order=data['Owner_Type'].value_cour
plt.title('Owner_Type distribution')
plt.ylabel('Count')
plt.xlabel('Owner_Type')
plt.xticks(rotation = 90)
plt.show()
```



There are 4 types of Owner, the most popular being first and the least fourth.

Bivariate Analysis

In [112…
```
num_df = data[['Year','Kilometers_Driven_Transformed','Mileage','Engine','Po
plt.figure(figsize = (15,7))
sns.heatmap(num_df.corr(), annot = True, vmin = -1, vmax = 1, fmt = ".2f", c
```

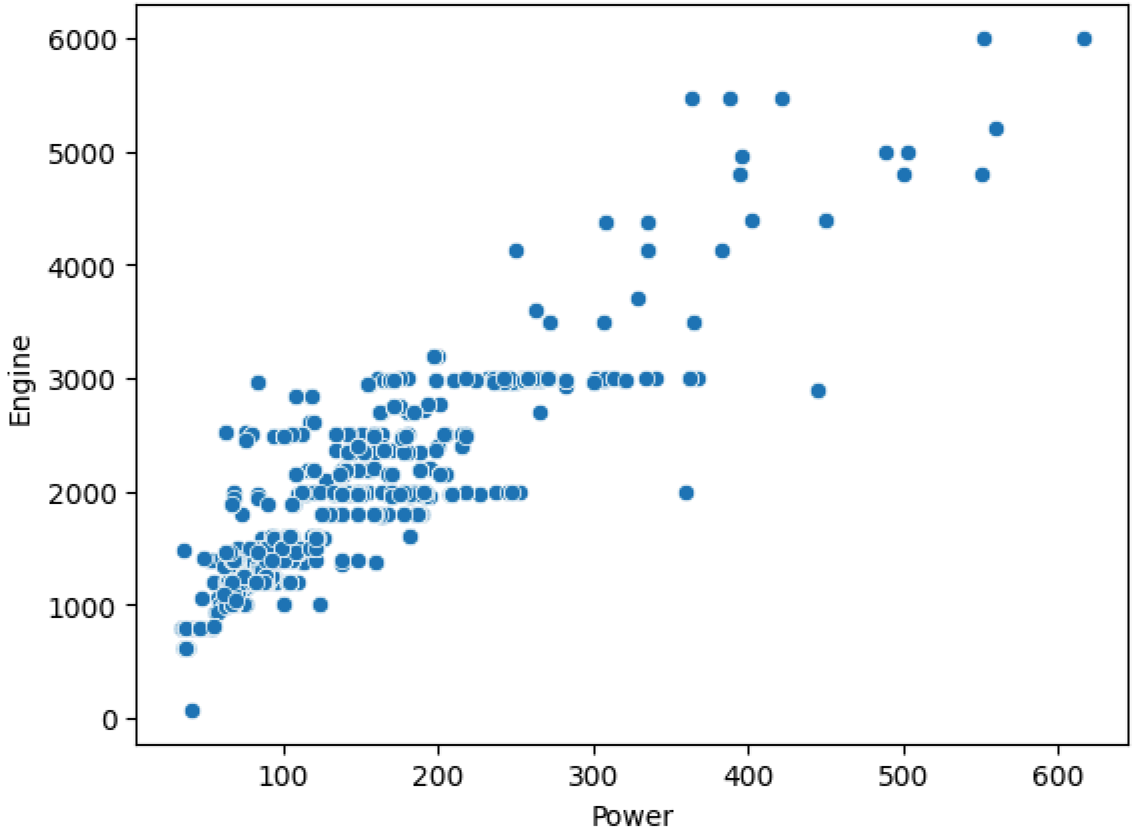
Out[112… <Axes: >

There are some significant correlations (>0.7 or <-0.7)shown in the heatmap.

- Power and Engine (0.86)
- New Price and Engine (0.74)
- Price transformed and Engine (0.69)
- New Price and Power (0.88)
- Price Transformed and Power (0.77)
- New Price and Price Transformed (0.79)

In [113…
```
sns.scatterplot(x = data['Power'], y = data['Engine'])
```

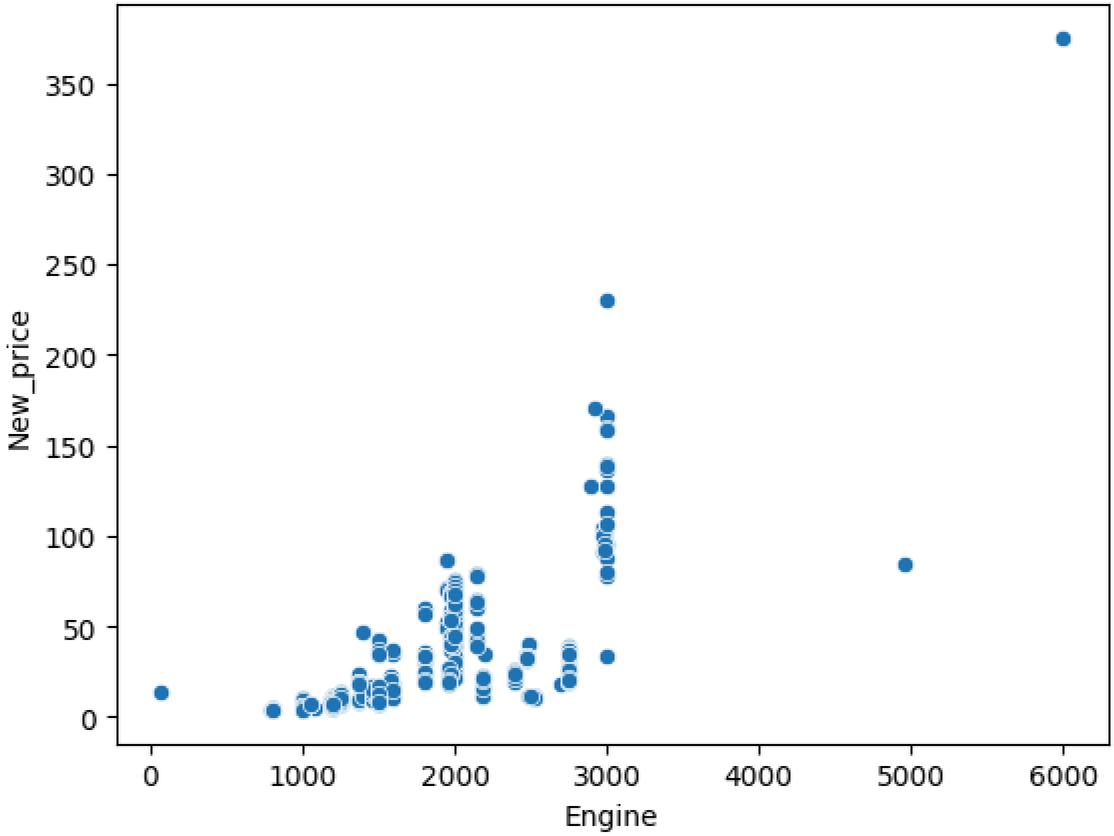
Out[113…
```
<Axes: xlabel='Power', ylabel='Engine'>
```



As the engine's displacement volume increases, so does Power.

In [114…
```
sns.scatterplot(x = data['Engine'], y = data['New_price'])
```

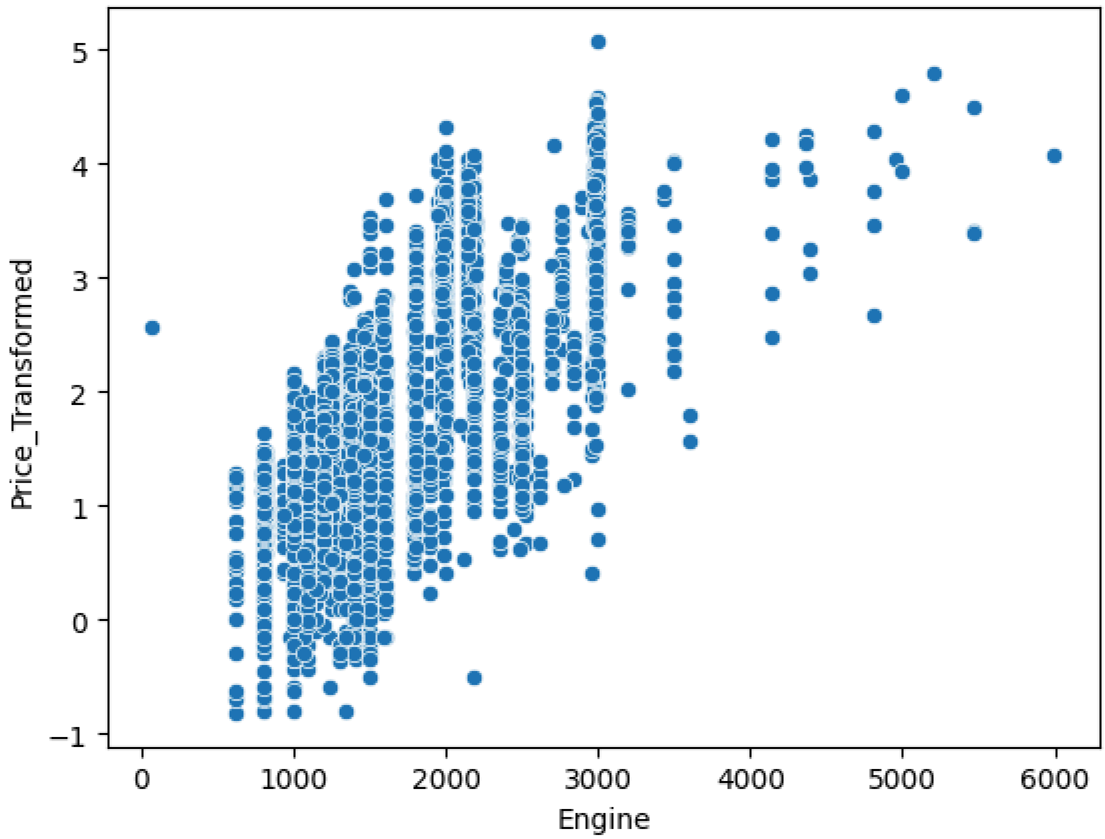
Out[114…
```
<Axes: xlabel='Engine', ylabel='New_price'>
```



As engine's displacement volume increases so does new price, similar thing happens with new price and power.

In [115…
```
sns.scatterplot(x = data['Engine'], y = data['Price_Transformed'])
```

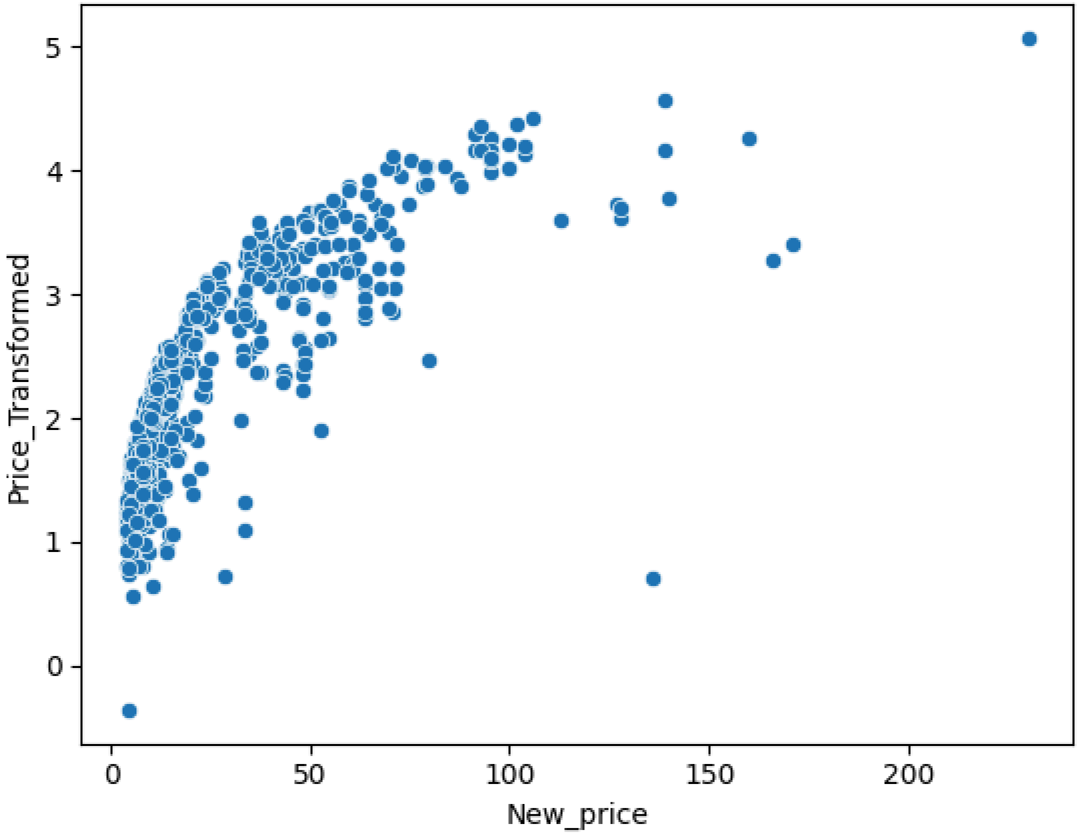
Out[115…
```
<Axes: xlabel='Engine', ylabel='Price_Transformed'>
```



As engine's displacement volume increases, so does price, similar thing happens with price and power.

In [116…
```
sns.scatterplot(x = data['New_price'], y = data['Price_Transformed'])
```

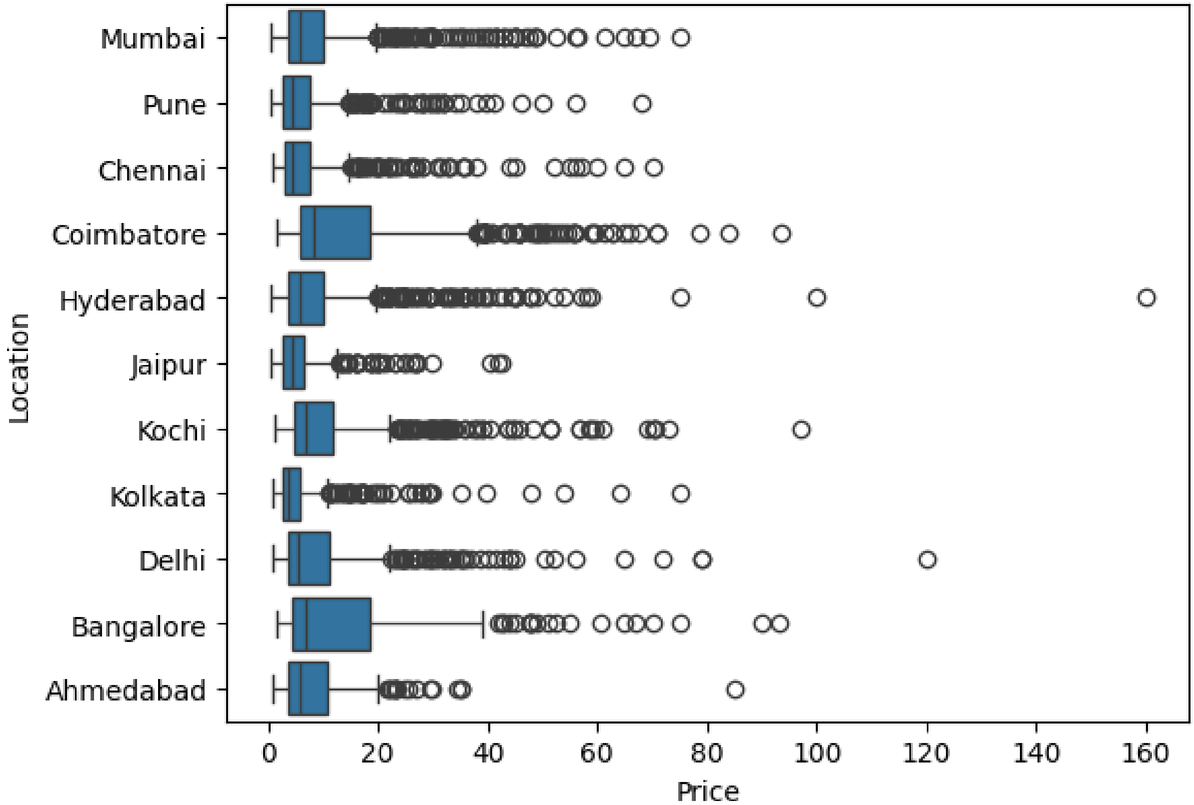
Out[116…
```
<Axes: xlabel='New_price', ylabel='Price_Transformed'>
```



There is correlation between new_price and price_transformed. This is expected since expensive new cars can become relatively expensive used cars.

In [117…
```
sns.boxplot(x = data['Price'], y = data['Location'])
```

Out[117…
```
<Axes: xlabel='Price', ylabel='Location'>
```



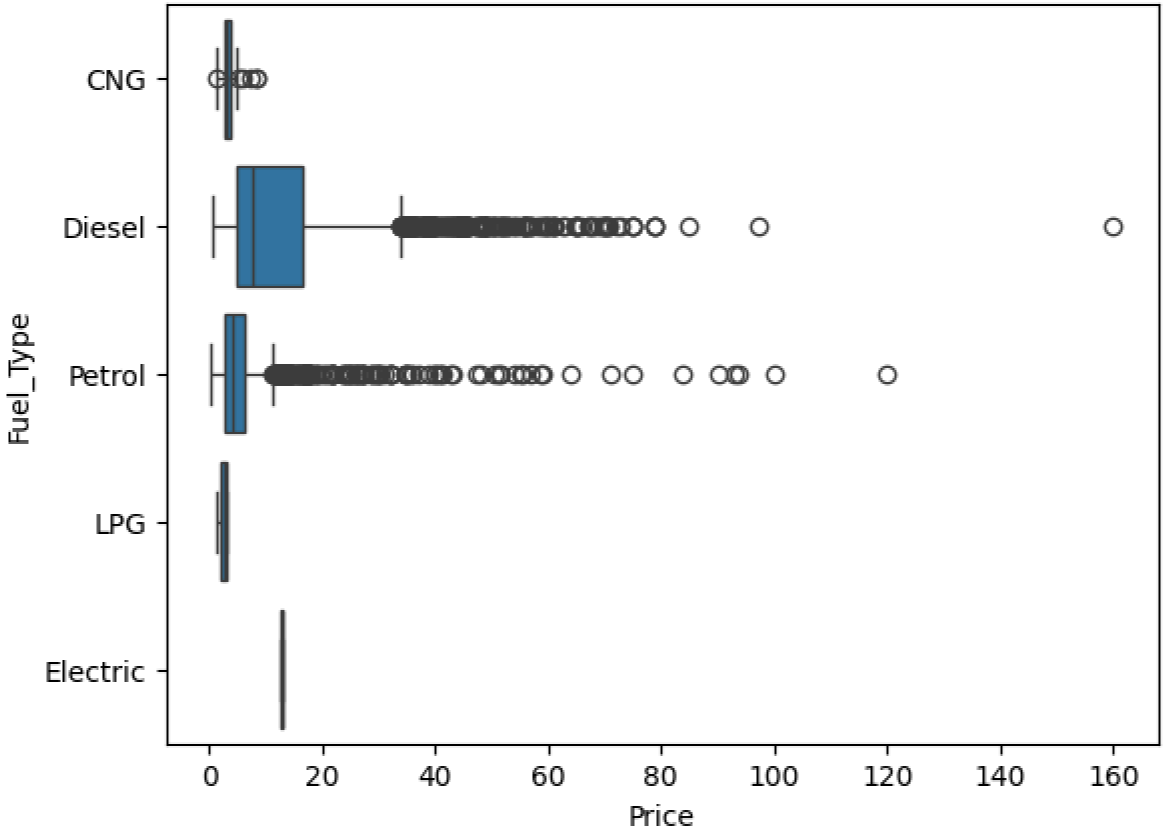
The distribution of prices may vary a little depending on the location.

In [118…

```
sns.boxplot(x = data['Price'], y = data['Fuel_Type'])
```

Out[118…

```
<Axes: xlabel='Price', ylabel='Fuel_Type'>
```



In [119…

```
data[data['Fuel_Type']=='Electric']
```

Out[119…

	Name	Location	Year	Kilometers_Driven	Fuel_Type	Transmission	Owner_
4446	Mahindra E Verito D4	Chennai	2016	50000	Electric	Automatic	
4904	Toyota Prius 2009-2016 Z4	Mumbai	2011	44000	Electric	Automatic	

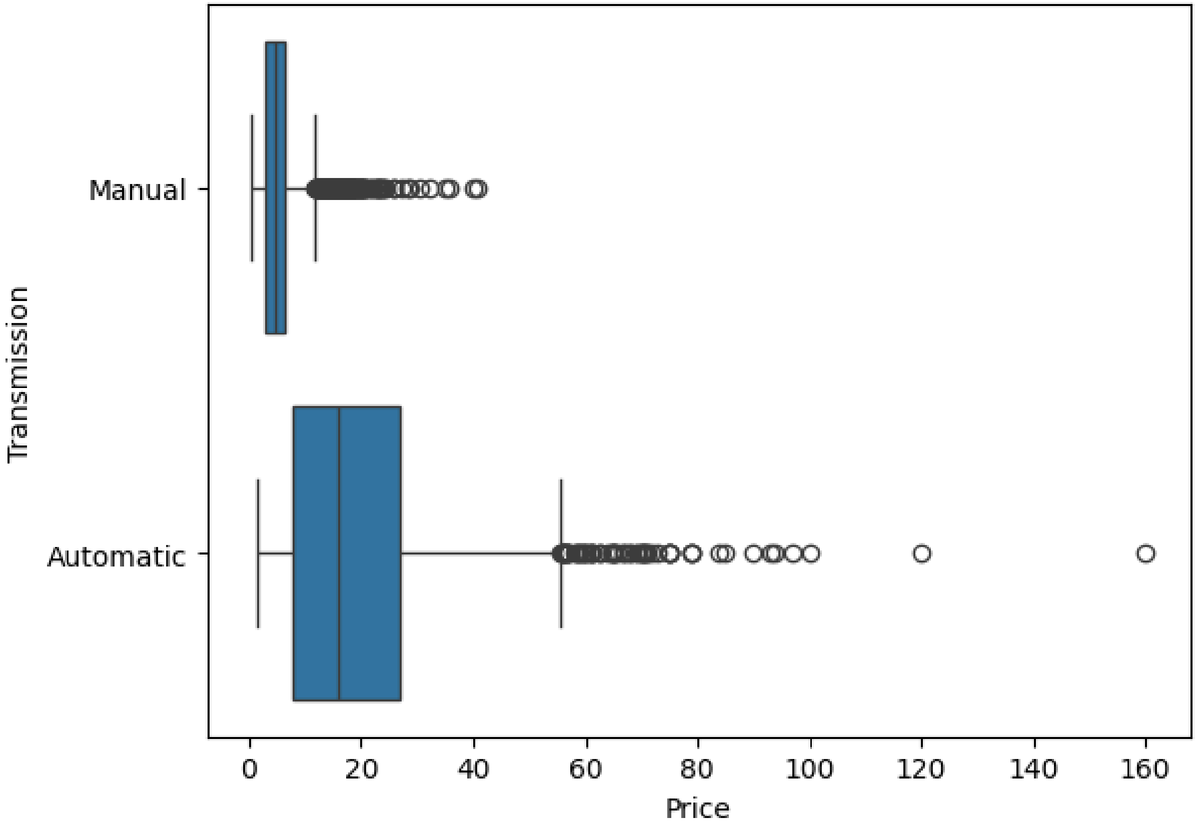
There are only 2 electric vehicles listed. It seems that vehicles that use diesel are more expensive.

In [120…

```
sns.boxplot(x = data['Price'], y = data['Transmission'])
```

Out[120…

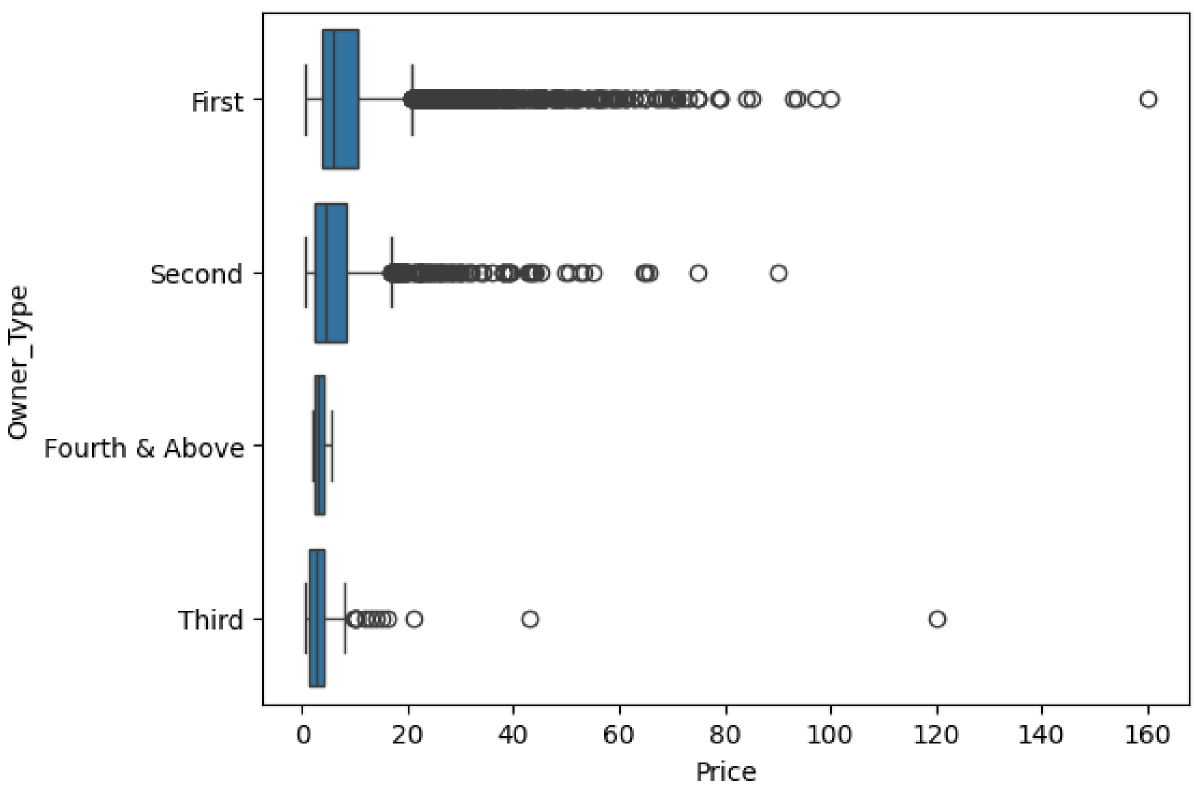
```
<Axes: xlabel='Price', ylabel='Transmission'>
```



Automatic transmission cars are more expensive, as expected.

In [121…
```
sns.boxplot(x = data['Price'], y = data['Owner_Type'])
```

Out[121…
```
<Axes: xlabel='Price', ylabel='Owner_Type'>
```



It makes sense that when the number of previous owners is lower, the price is higher.

Feature Engineering

In [122…

```
# Split the 'Name' column into 'Brand' and 'Model'
data[['Brand', 'Model']] = data['Name'].str.split(n=1, expand=True)

# Display the first few rows of the dataframe to verify the new columns
data.head(7)
```

Out[122…

	Name	Location	Year	Kilometers_Driven	Fuel_Type	Transmission	Owner_T
0	Maruti Wagon R LXI CNG	Mumbai	2010	72000	CNG	Manual	
1	Hyundai Creta 1.6 CRDi SX Option	Pune	2015	41000	Diesel	Manual	
2	Honda Jazz V	Chennai	2011	46000	Petrol	Manual	
3	Maruti Ertiga VDI	Chennai	2012	87000	Diesel	Manual	
4	Audi A4 New 2.0 TDI Multitronic	Coimbatore	2013	40670	Diesel	Automatic	Sec
5	Hyundai EON LPG Era Plus Option	Hyderabad	2012	75000	LPG	Manual	
6	Nissan Micra Diesel XV	Jaipur	2013	86999	Diesel	Manual	

In [123…

```
data['Brand'].value_counts()
```

Out[123…
```
Brand
Maruti           1444
Hyundai          1340
Honda             743
Toyota            507
Mercedes-Benz     380
Volkswagen        374
Ford              351
Mahindra          331
BMW               311
Audi              285
Tata              228
Skoda             202
Renault           170
Chevrolet         151
Nissan            117
Land               67
Jaguar             48
Fiat               38
Mitsubishi         36
Mini               31
Volvo              28
Porsche            19
Jeep               19
Datsun             17
ISUZU               3
Force               3
Isuzu               2
Bentley             2
Smart               1
Ambassador          1
Lamborghini         1
Hindustan           1
OpelCorsa           1
Name: count, dtype: int64
```

In [124…
```
data['Model'].value_counts()
```

Out[124…
```
Model
XUV500 W8 2WD                    55
Swift VDI                        49
Swift Dzire VDI                  42
City 1.5 S MT                    39
Swift VDI BSIV                   37
                                 ..
Manza Aura Plus Quadrajet BS IV   1
Indigo eCS LS (TDI) BS-III        1
Grand i10 Era                     1
Figo Diesel                       1
Elite i20 Magna Plus              1
Name: count, Length: 2041, dtype: int64
```

Missing value treatment

In [125…

```
data.isnull().sum()
```

Out[125…

```
Name                             0
Location                         0
Year                             0
Kilometers_Driven                0
Fuel_Type                        0
Transmission                     0
Owner_Type                       0
Mileage                          2
Engine                          46
Power                          175
Seats                           53
New_price                     6246
Price                         1234
Kilometers_Driven_Transformed    0
Price_Transformed             1234
Brand                            0
Model                            0
dtype: int64
```

Mileage missing values

In [126…

```
# Impute null values with median
data['Mileage'].fillna(data['Mileage'].median(), inplace=True)
```

In [127…

```
# Check that the previous method worked
data['Mileage'].isnull().sum()
```

Out[127…

```
0
```

Also treat cases where mileage = 0

In [128…

```
# Find rows where 'Mileage' is 0
missing_mileage = data[data['Mileage'] == 0]

# Find rows where 'Mileage' is not 0
not_missing_mileage = data[data['Mileage'] != 0]

# Impute missing 'Mileage' values based on the same model's existing values
for index, row in missing_mileage.iterrows():
    model = row['Model']
    mileage_value = not_missing_mileage[not_missing_mileage['Model'] == mode
    if pd.notnull(mileage_value) and mileage_value != 0:
        data.loc[index, 'Mileage'] = mileage_value

# Fill any remaining missing values with the overall median
median_mileage = data[data['Mileage'] != 0]['Mileage'].median()
data.loc[data['Mileage'] == 0, 'Mileage'] = median_mileage
```

In [129…

```
data[data['Mileage'] == 0]
```

Out [129…

Name	Location	Year	Kilometers_Driven	Fuel_Type	Transmission	Owner_Type	M

This ensures that the values where treated properly.

Engine missing values

In [130…
```
# Find rows where 'Engine' is missing
missing_engine = data[data['Engine'].isnull()]

# Find rows where 'Engine' is not missing
not_missing_engine = data[data['Engine'].notnull()]

# Impute missing 'Engine' values based on the same model's existing values
for index, row in missing_engine.iterrows():
    model = row['Model']
    engine_value = not_missing_engine[not_missing_engine['Model'] == model][
    if pd.notnull(engine_value):
        data.loc[index, 'Engine'] = engine_value

# Fill any remaining missing values with the overall median
median_engine = data['Engine'].median()
data['Engine'].fillna(median_engine, inplace=True)
```

In [131…
```
data['Engine'].isnull().sum()
```

Out[131… 0

Now there are no more missing Engine values.

Power missing values

In [132…
```
# Find rows where 'Power' is missing
missing_power = data[data['Power'].isnull()]

# Find rows where 'Power' is not missing
not_missing_power = data[data['Power'].notnull()]

# Impute missing 'Power' values based on the same model's existing values
for index, row in missing_power.iterrows():
    model = row['Model']
    power_value = not_missing_power[not_missing_power['Model'] == model]['Pc
    if pd.notnull(power_value):
        data.loc[index, 'Power'] = power_value

# Fill any remaining missing values with the overall median
median_power = data['Power'].median()
data['Power'].fillna(median_power, inplace=True)
```

In [133…
```
data['Power'].isnull().sum()
```

Out[133… 0

No more missing values in 'Power'

Seats missing values

In [134…

```
# Find rows where 'Seats' is missing
missing_seats = data[data['Seats'].isnull()]

# Find rows where 'Seats' is not missing
not_missing_seats = data[data['Seats'].notnull()]

# Impute missing 'Seats' values based on the same model's existing values
for index, row in missing_seats.iterrows():
    model = row['Model']
    seats_value = not_missing_seats[not_missing_seats['Model'] == model]['Se
    if pd.notnull(seats_value):
        data.loc[index, 'Seats'] = seats_value

# Fill any remaining missing values with the overall median
median_seats = data['Seats'].median()
data['Seats'].fillna(median_seats, inplace=True)
```

In [135…

```
data.isnull().sum()
```

Out[135…

```
Name                              0
Location                          0
Year                              0
Kilometers_Driven                 0
Fuel_Type                         0
Transmission                      0
Owner_Type                        0
Mileage                           0
Engine                            0
Power                             0
Seats                             0
New_price                      6246
Price                          1234
Kilometers_Driven_Transformed     0
Price_Transformed              1234
Brand                             0
Model                             0
dtype: int64
```

New_price missing values

Since there are a lot of new_price missing values, comparing models will not work on the majority of cases, so in order to get a more precise replacement of null values, a comparison between a few attributes might help.

In [136…

```
# Find rows where 'New_price' is missing
missing_new_price = data[data['New_price'].isnull()]

# Find rows where 'New_price' is not missing
```

```
not_missing_new_price = data[data['New_price'].notnull()]

# Step 1: Model-specific median
for index, row in missing_new_price.iterrows():
    model = row['Model']
    new_price_value = not_missing_new_price[not_missing_new_price['Model'] =
    if pd.notnull(new_price_value):
        data.loc[index, 'New_price'] = new_price_value

# Update missing_new_price after first step
missing_new_price = data[data['New_price'].isnull()]

# Step 2: Brand and Year
for index, row in missing_new_price.iterrows():
    brand = row['Brand']
    year = row['Year']
    new_price_value = not_missing_new_price[(not_missing_new_price['Brand']
    if pd.notnull(new_price_value):
        data.loc[index, 'New_price'] = new_price_value

# Update missing_new_price after second step
missing_new_price = data[data['New_price'].isnull()]

# Step 3: Brand and Fuel_Type
for index, row in missing_new_price.iterrows():
    brand = row['Brand']
    fuel_type = row['Fuel_Type']
    new_price_value = not_missing_new_price[(not_missing_new_price['Brand']
    if pd.notnull(new_price_value):
        data.loc[index, 'New_price'] = new_price_value

# Update missing_new_price after third step
missing_new_price = data[data['New_price'].isnull()]

# Step 4: Overall Brand
for index, row in missing_new_price.iterrows():
    brand = row['Brand']
    new_price_value = not_missing_new_price[not_missing_new_price['Brand'] =
    if pd.notnull(new_price_value):
        data.loc[index, 'New_price'] = new_price_value

# Verify the imputation
missing_new_price = data[data['New_price'].isnull()]
```

In [137…

```
data['New_price'].isnull().sum()
```

Out[137… 162

There are still 162 missing values, since these are very few compared to the initial 6+ thousand, the overall median will be used.

In [138…

```
data['New_price'].fillna(data['New_price'].median(), inplace=True)
```

In [139…

```
data['New_price'].isnull().sum()
```

Out[139…

```
0
```

There are no more missing values.

Price missing values

Since this is the dependent variable, imputing may affect the quality of the models.

In [140…

```
# Drop rows where Price is missing
data.dropna(subset=['Price'], inplace = True)
```

In [141…

```
data.isnull().sum()
```

Out[141…

```
Name                           0
Location                       0
Year                           0
Kilometers_Driven              0
Fuel_Type                      0
Transmission                   0
Owner_Type                     0
Mileage                        0
Engine                         0
Power                          0
Seats                          0
New_price                      0
Price                          0
Kilometers_Driven_Transformed  0
Price_Transformed              0
Brand                          0
Model                          0
dtype: int64
```

There are no more missing values within the dataset.

Important Insights from EDA and Data Preprocessing

- There was a square root transformation made on Kilometers_Driven and a Log transformation made to the dependent variable.
- Missing values were imputed based on other features like Model, Brand and/or Fuel type.
- Feature 'Name' was separated into two, Brand and Model.

Building Various Models

1. What we want to predict is the "Price". We will use the normalized version 'price_log' for modeling.

2. Before we proceed to the model, we'll have to encode categorical features. We will drop categorical features like Name.
3. We'll split the data into train and test, to be able to evaluate the model that we build on the train data.
4. Build Regression models using train data.
5. Evaluate the model performance.

Split the Data

- Step1: Seperating the indepdent variables (X) and the dependent variable (y).
- Step2: Encode the categorical variables in X using pd.dummies.
- Step3: Split the data into train and test using train_test_split.

Linear regression

In [142…

```
# Function to check VIF
check_vif = data.select_dtypes(include=['number']).drop(columns = ['Price_Tr
def checking_vif(train):
    vif = pd.DataFrame()
    vif["feature"] = train.columns

    # Calculating VIF for each feature
    vif["VIF"] = [
        variance_inflation_factor(train.values, i) for i in range(len(train.
    ]
    return vif

print(checking_vif(check_vif))
```

```
                        feature       VIF
0                       Mileage  7.024115
1                         Power  8.234872
2                     New_price  3.130134
3  Kilometers_Driven_Transformed  8.439496
```

After checking the collinearity between variables, these are the only ones remaining to get VIF scores under 10.

In [143…

```
# Take dependent and independent variables
X = data.drop(['Name', 'Price', 'Price_Transformed', 'Kilometers_Driven', 'Y
X = pd.get_dummies(X, drop_first=True)
X = sm.add_constant(X)  # Add constant

Y = data['Price_Transformed']

# Split the data
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, rar

# Convert all boolean columns to integers
bool_columns = X_train.select_dtypes(include='bool').columns
```

```
for col in bool_columns:
    X_train[col] = X_train[col].astype(int)
```

In [144…
```
# Model Performance on test and train data
def model_performance(olsmodel, x_train, x_test, y_train, y_test):

    # In-sample Prediction
    y_pred_train = olsmodel.predict(x_train)
    y_observed_train = y_train

    # Prediction on test data
    y_pred_test = olsmodel.predict(x_test)
    y_observed_test = y_test

    print(
        pd.DataFrame(
            {
                "Data": ["Train", "Test"],
                "RMSE": [
                    np.sqrt(mean_squared_error(y_observed_train, y_pred_trai
                    np.sqrt(mean_squared_error(y_observed_test,y_pred_test))
                ],
                "MAE": [
                    mean_absolute_error(y_observed_train, y_pred_train),
                    mean_absolute_error(y_observed_test,y_pred_test),
                ],
                "MAPE": [
                    mean_absolute_percentage_error(y_observed_train, y_pred_
                    mean_absolute_percentage_error(y_observed_test,y_pred_te
                ],

                'r2': [
                    r2_score(y_observed_train, y_pred_train),
                    r2_score(y_observed_test,y_pred_test),
                ],
            }
        )
    )
```

In [145…
```
# Train model
linear1 = sm.OLS(Y_train, X_train).fit()
model_performance(linear1, X_train, X_test, Y_train, Y_test)
```

```
    Data      RMSE       MAE          MAPE        r2
0  Train  0.154192  0.100564  2.535729e+12  0.968666
1   Test  0.384684  0.238129  2.106962e+12  0.808400
```

Removing the variables that have a p-value higher than 0.05 may improve the model's performance.

In [146…
```
# Extract p-values
p_values = linear1.pvalues

# Identify columns with p-values > 0.05 (excluding the constant)
```

```
columns_to_drop = p_values[p_values > 0.05].index
columns_to_drop = columns_to_drop[columns_to_drop != 'const']

# Drop these columns from the dataframe
X = X.drop(columns=columns_to_drop)
```

In [147…

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, rar
# Convert all boolean columns to integers
bool_columns = X_train.select_dtypes(include='bool').columns

for col in bool_columns:
    X_train[col] = X_train[col].astype(int)

# Refit the model with remaining columns
linear2 = sm.OLS(Y_train, X_train).fit()
model_performance(linear2, X_train, X_test, Y_train, Y_test)
#print(linear2.summary())
```

	Data	RMSE	MAE	MAPE	r2
0	Train	0.180648	0.127895	2.713904e+12	0.956992
1	Test	0.321485	0.222521	2.330986e+12	0.866184

In [148…

```
# Repeat the steps
# Extract p-values
p_values = linear2.pvalues

# Identify columns with p-values > 0.05 (excluding the constant)
columns_to_drop = p_values[p_values > 0.05].index
columns_to_drop = columns_to_drop[columns_to_drop != 'const']

# Drop these columns from the dataframe
X = X.drop(columns=columns_to_drop)
```

In [149…

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.33 ,ra
# Convert all boolean columns to integers
bool_columns = X_train.select_dtypes(include='bool').columns

for col in bool_columns:
    X_train[col] = X_train[col].astype(int)

# Refit the model with remaining columns
linear3 = sm.OLS(Y_train, X_train).fit()
model_performance(linear3, X_train, X_test, Y_train, Y_test)
#print(linear3.summary())
```

	Data	RMSE	MAE	MAPE	r2
0	Train	0.228469	0.156834	1.542889e+12	0.931519
1	Test	0.313798	0.215494	6.489165e+12	0.871222

Check assumptions of linear regression

Mean of residuals = 0

In [150…

```
residuals = linear3.resid
```

```
np.mean(residuals)
```

Out[150… 1.3645059509100869e-14

Mean of residuals is almost equal to 0.

Homoscedasticity

- Residuals must be symetrically distributed across the regresion line.
- Goldfelquandt test with alpha = 0.05
- Null hypotheses: Residuals are homoscedastic.
- Alternate hypotheses: Residuals are heteroscedastic.

In [151…

```
# Perform test and display results
name = ["F statistic", "p-value"]

test = sms.het_goldfeldquandt(Y_train, X_train)

lzip(name, test)
```

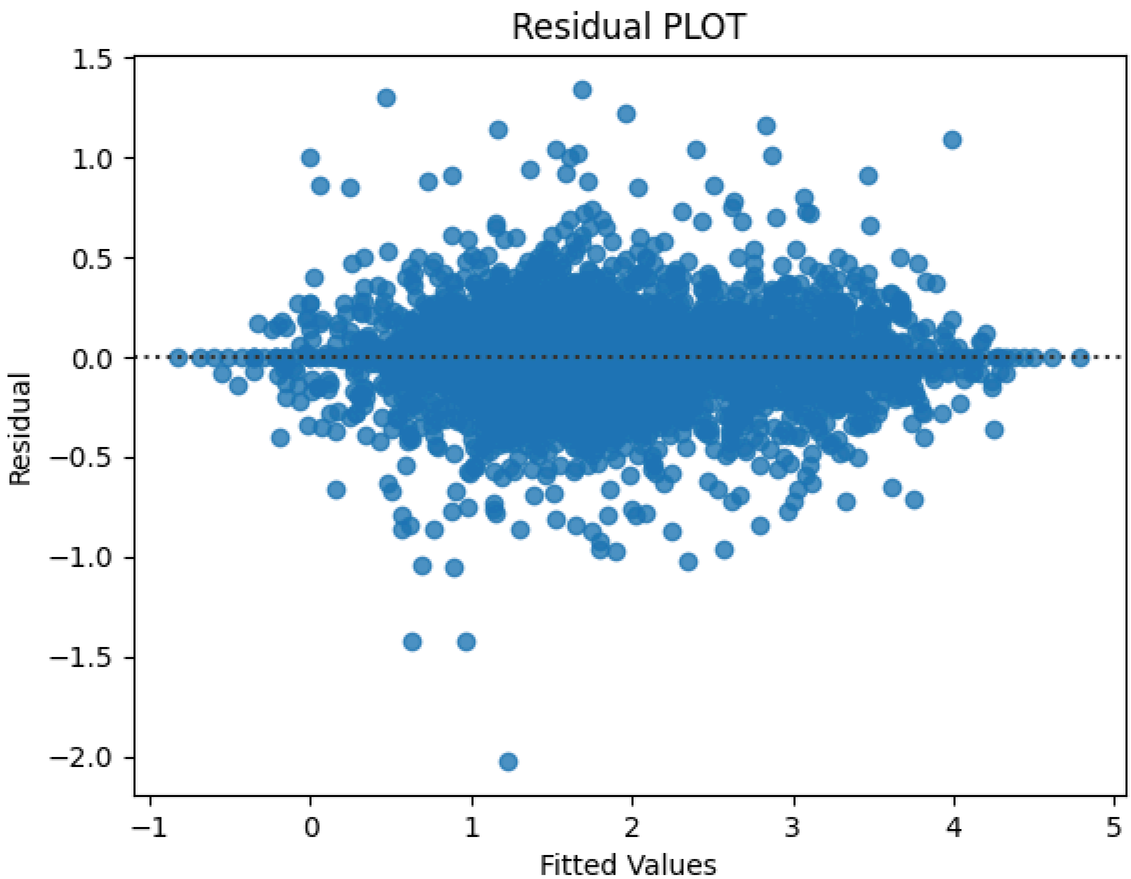
Out[151… [('F statistic', 1.037947472721264), ('p-value', 0.23744596342610716)]

Since the p-value is greater than 0.05, the assumption holds.

Linearity of variables

In [152…

```
# Predicted values
fitted = linear3.fittedvalues

sns.residplot(x = fitted, y = residuals)
plt.xlabel("Fitted Values")
plt.ylabel("Residual")
plt.title("Residual PLOT")

plt.show()
```



The residuals are randomly and uniformely scattered along the x axis, they do not form any pattern or follow any trend.

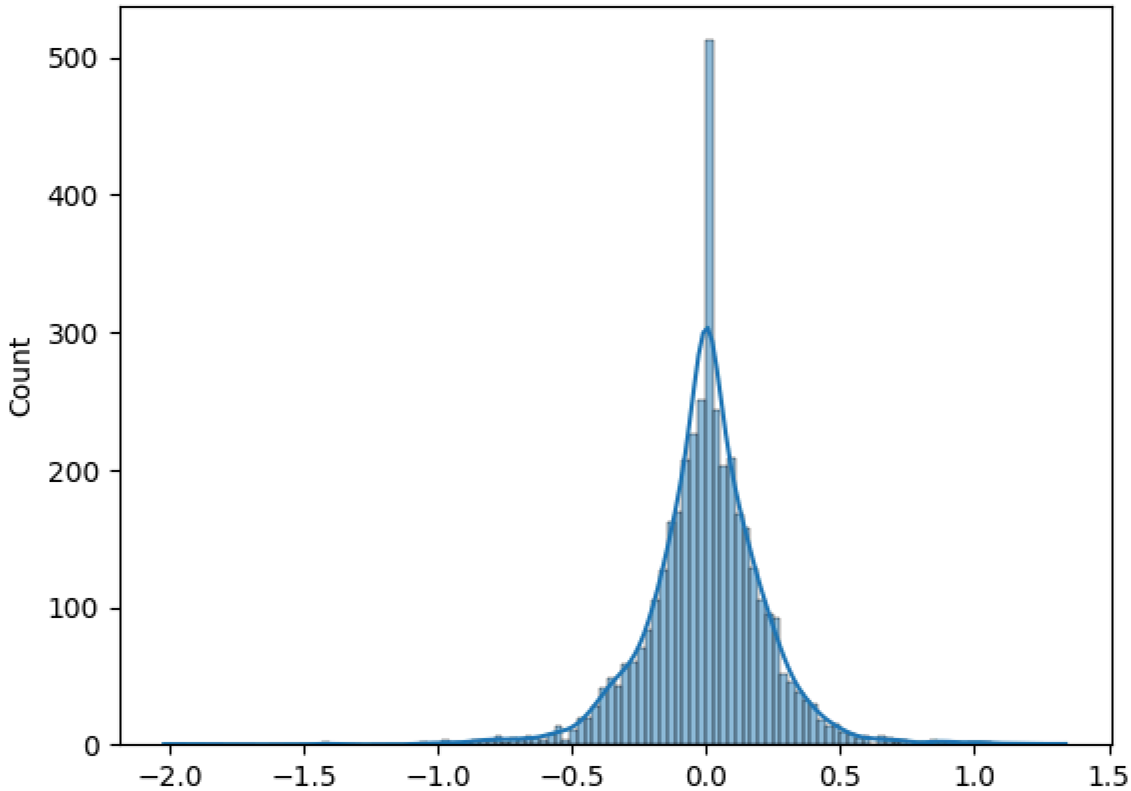
Normality of error terms

In [153…

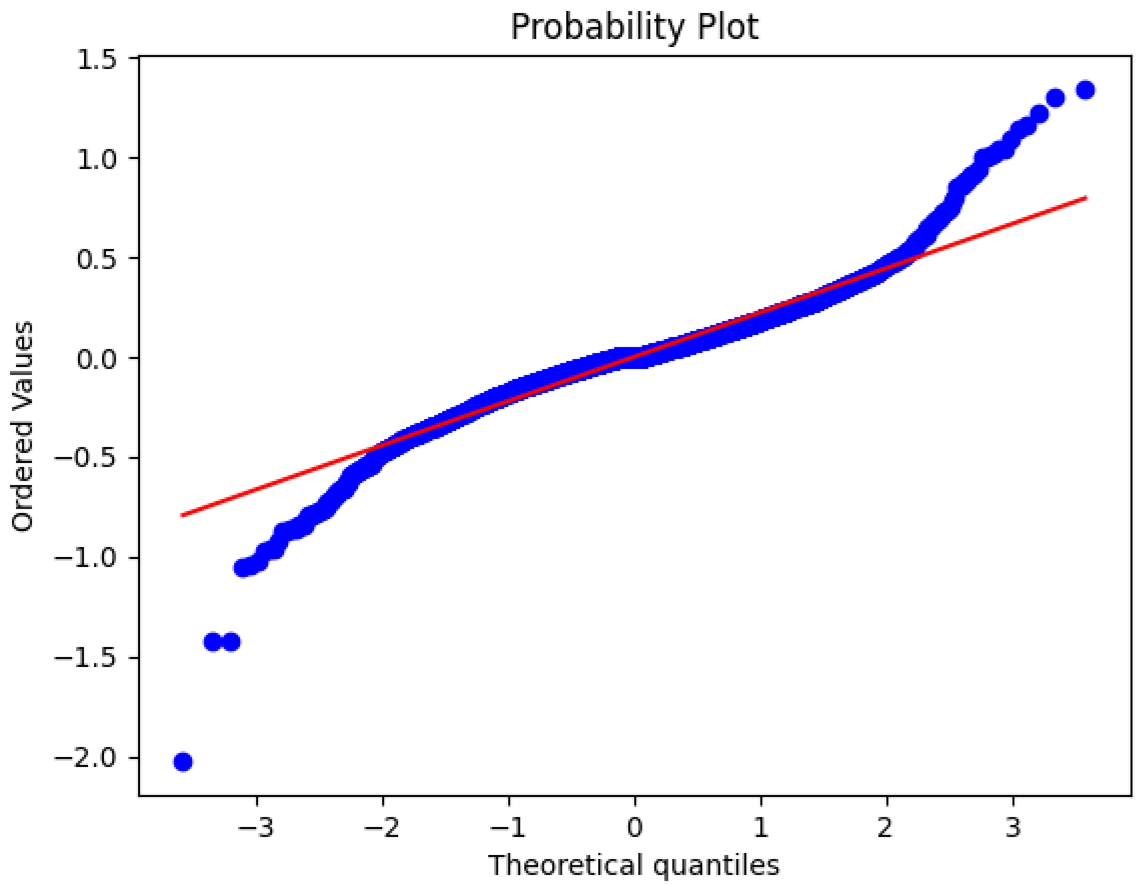
```
# Plot histogram to see distribution of residuals
sns.histplot(residuals, kde = True)
```

Out[153…

```
<Axes: ylabel='Count'>
```



In [154…
```
# Q-Q plot to confirm normality
stats.probplot(residuals, dist = "norm", plot = pylab)

plt.show()
```



The residuals follow a fairly normal distribution.

In [155…

```
# Function to perform cross validation
def cross_validate_sm_ols(X, y, k=10):
    "Perform cross-validation, takes the values of the dependent variables a

    kf = KFold(n_splits=k, shuffle=True, random_state=1)
    r2_scores = []
    rmse_scores = []

    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        model = sm.OLS(y_train, X_train).fit()
        y_pred = model.predict(X_test)

        r2_scores.append(r2_score(y_test, y_pred))
        rmse_scores.append(mean_squared_error(y_test, y_pred, squared=False)

    mean_r2 = np.mean(r2_scores)
    std_r2 = np.std(r2_scores)
    mean_rmse = np.mean(rmse_scores)
    std_rmse = np.std(rmse_scores)

    return mean_r2, std_r2, mean_rmse, std_rmse
```

In [156…

```
# Use cross validation function for different values of k
results = []
for k in range(2, 10):
    for col in bool_columns:
      X[col] = X[col].astype(int)
    mean_r2, std_r2, mean_mrse, std_mrse = cross_validate_sm_ols(X.values, Y
    results.append((k, mean_r2, 2 * std_r2, mean_mrse, 2 * std_mrse))

results_df = pd.DataFrame(results, columns=['k', 'R-squared', ' +/-', 'MRSE'
print(results_df)
```

```
   k  R-squared      +/-      MRSE      +/-
0  2   0.856162  0.000409  0.331282  0.004762
1  3   0.866473  0.005125  0.318974  0.008838
2  4   0.870237  0.017880  0.314437  0.028861
3  5   0.873612  0.016838  0.310348  0.036898
4  6   0.874409  0.019001  0.309261  0.032835
5  7   0.876201  0.017040  0.307160  0.031769
6  8   0.875481  0.023062  0.307739  0.034164
7  9   0.876509  0.033370  0.306183  0.049714
```

Splitting the data in different ratios slightly affect the performance of the model. Overall, the model is able to explain ~87% of the variation.

Ridge regression

- Now ridge regression will be used to create a model to predict the values. -Ridge regression is used instead of Lasso regression due to the correlation between some variables, Lasso regression is able to make coefficients zero while Ridgre regression is not able to do that.

In [168…

```
# Take dependent and independent variables
X = data.drop(['Name', 'Price', 'Price_Transformed', 'Kilometers_Driven'], a
X = pd.get_dummies(X, drop_first=True)

Y = data['Price_Transformed']

# Split the data
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, rar

# Convert all boolean columns to integers
bool_columns = X_train.select_dtypes(include='bool').columns

for col in bool_columns:
    X_train[col] = X_train[col].astype(int)
```

In [169…

```
# Check the best alpha
params = {'alpha': [0.001, 0.1, 0.2, 0.5, 0.9, 1, 2, 5, 8, 10]}
folds = KFold(n_splits = 10, shuffle = True, random_state = 1)

ridge = Ridge()
```

```
ridge_cv = GridSearchCV(estimator=ridge, param_grid=params, cv=folds, scorir
ridge_cv.fit(X_train, Y_train)
```

Out[169…

```
▸ GridSearchCV
▸ estimator: Ridge
    ▸ Ridge
```

In [171…

```
# View the result
ridge_cv.best_params_
```

Out[171… {'alpha': 0.5}

In [172…

```
# Build the model with the right alpha
ridge_model =  Ridge(alpha = 0.5)
ridge_model.fit(X_train, Y_train)
```

Out[172…

```
▾ Ridge
Ridge(alpha=0.5)
```

In [173…

```
# Evaluate model performance
model_performance(ridge_model, X_train, X_test, Y_train, Y_test)
```

	Data	RMSE	MAE	MAPE	r2
0	Train	0.123351	0.088638	2.211775e+12	0.979938
1	Test	0.188580	0.128700	1.559865e+12	0.954479

In [174…

```
# Function to cross validate Ridge
def cross_validate_ridge(X, y, alpha=0.5, k=10):
    """
    Perform cross-validation for Ridge regression.
    X: Features
    y: Target variable
    alpha: Regularization strength
    k: Number of folds for cross-validation
    """
    kf = KFold(n_splits=k, shuffle=True, random_state=1)
    r2_scores = []
    rmse_scores = []

    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        model = Ridge(alpha=alpha)
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

        r2_scores.append(r2_score(y_test, y_pred))
        rmse_scores.append(mean_squared_error(y_test, y_pred, squared=False)

    mean_r2 = np.mean(r2_scores)
```

```
    std_r2 = np.std(r2_scores)
    mean_rmse = np.mean(rmse_scores)
    std_rmse = np.std(rmse_scores)

    return mean_r2, std_r2, mean_rmse, std_rmse
```

In [175…

```
# Use cross validation function for different values of k
results = []
for k in range(2, 10):
    for col in bool_columns:
      X[col] = X[col].astype(int)
    mean_r2, std_r2, mean_mrse, std_mrse = cross_validate_ridge(X.values, Y.
    results.append((k, mean_r2, 2 * std_r2, mean_mrse, 2 * std_mrse))

results_df = pd.DataFrame(results, columns=['k', 'R-squared', ' +/-', 'MRSE'
print(results_df)
```

```
   k  R-squared       +/-      MRSE       +/-
0  2   0.944542  0.008589  0.205614  0.019188
1  3   0.946116  0.013123  0.202461  0.030702
2  4   0.947002  0.016497  0.200739  0.038563
3  5   0.948281  0.014026  0.198236  0.034635
4  6   0.948834  0.015132  0.197189  0.036058
5  7   0.948941  0.025387  0.196107  0.048159
6  8   0.949121  0.023119  0.196058  0.046776
7  9   0.949356  0.028700  0.195112  0.056382
```

- This model has a better performance than the Linear regression model made previously.
- It is able to consistently explain about ~94% - 95% of variance found in the target variable
- Thus far, this is the best model

Observations

- The optimized value of alpha was of 0.5
- This model performs better than the simple linear regression since Ridge regression is more robust when variables have collinearity.
- The ridge regression model was able to explain about ~94% of the variation.

Hyperparameter Tuning: Decision Tree

In [176…

```
# Check the best parameters
params = {
    'max_depth': [None, 10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 2, 5, 10]
}
folds = KFold(n_splits = 10, shuffle = True, random_state = 1)

decision_tree = DecisionTreeRegressor()
```

```
decision_tree_cv = GridSearchCV(estimator=decision_tree, param_grid=params,
decision_tree_cv.fit(X_train, Y_train)
```

Out[176…

```
GridSearchCV
estimator: DecisionTreeRegressor
DecisionTreeRegressor
```

In [177…

```
# show best parameters
decision_tree_cv.best_params_
```

Out[177…

```
{'max_depth': 50, 'min_samples_leaf': 1, 'min_samples_split': 20}
```

Now, use those parameters to build a model

In [184…

```
# Build model with best parameters
tree_model = DecisionTreeRegressor(max_depth=50, min_samples_leaf=1, min_sam
tree_model.fit(X_train, Y_train)
```

Out[184…

```
DecisionTreeRegressor
DecisionTreeRegressor(max_depth=50, min_samples_split=20)
```

In [185…

```
# Evaluate model performance
model_performance(tree_model, X_train, X_test, Y_train, Y_test)
```

	Data	RMSE	MAE	MAPE	r2
0	Train	0.147550	0.104590	2.960163e+12	0.971294
1	Test	0.236967	0.172893	1.657659e+12	0.928122

This model is able to explain around 93% of variance in the target variable.

In [186…

```
# Function to cross validate Decision Tree
def cross_validate_tree(X, y, k=10, max_depth = 20, min_samples_leaf = 1, mi
    """
    Perform cross-validation for Ridge regression.
    X: Features
    y: Target variable
    alpha: Regularization strength
    k: Number of folds for cross-validation
    """
    kf = KFold(n_splits=k, shuffle=True, random_state=1)
    r2_scores = []
    rmse_scores = []

    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        model = DecisionTreeRegressor(max_depth = max_depth, min_samples_lea
        model.fit(X_train, y_train)
```

```
        y_pred = model.predict(X_test)

        r2_scores.append(r2_score(y_test, y_pred))
        rmse_scores.append(mean_squared_error(y_test, y_pred, squared=False)

    mean_r2 = np.mean(r2_scores)
    std_r2 = np.std(r2_scores)
    mean_rmse = np.mean(rmse_scores)
    std_rmse = np.std(rmse_scores)

    return mean_r2, std_r2, mean_rmse, std_rmse
```

In [187…

```
# Use cross validation function for different values of k
results = []
for k in range(2, 10):
    mean_r2, std_r2, mean_mrse, std_mrse = cross_validate_tree(X.values, Y.v
    results.append((k, mean_r2, 2 * std_r2, mean_mrse, 2 * std_mrse))

results_df = pd.DataFrame(results, columns=['k', 'R-squared', ' +/-', 'MRSE'
print(results_df)
```

```
   k  R-squared       +/-      MRSE       +/-
0  2   0.898688  0.008460  0.278016  0.016002
1  3   0.908661  0.011952  0.263781  0.023395
2  4   0.910075  0.019965  0.261684  0.038314
3  5   0.910831  0.020143  0.260205  0.033850
4  6   0.910998  0.015782  0.260228  0.029504
5  7   0.912171  0.031418  0.257922  0.050143
6  8   0.909073  0.032058  0.262560  0.054154
7  9   0.913745  0.032252  0.255585  0.053398
```

This model is better than the simple Linear regression model, but it is not better than the Ridge Regression model.

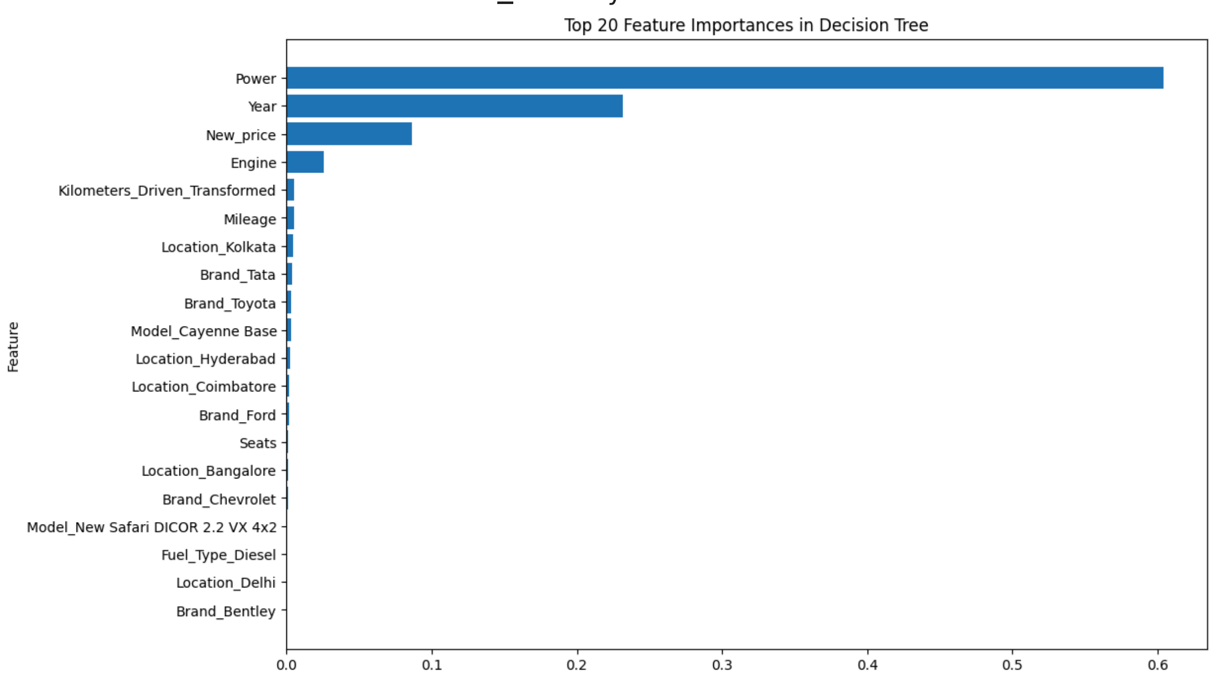
Feature Importance

In [188…

```
# Access feature importances
feature_importances = tree_model.feature_importances_

# Create a DataFrame for better readability
features = X.columns
importance_df = pd.DataFrame({
    'Feature': features,
    'Importance': feature_importances
})

# Sort the DataFrame by importance
importance_df = importance_df.sort_values(by='Importance', ascending=False)

# Select the top 20 features
top_20_features = importance_df.head(20)

# Display the DataFrame
print(top_20_features)
```

```
# Plot the feature importances of the top 20 features
plt.figure(figsize=(12, 8))
plt.barh(top_20_features['Feature'], top_20_features['Importance'])
plt.xlabel('Feature Importance')
plt.ylabel('Feature')
plt.title('Top 20 Feature Importances in Decision Tree')
plt.gca().invert_yaxis()
plt.show()
```

```
                                 Feature  Importance
3                                  Power    0.604303
0                                   Year    0.231673
5                              New_price    0.086054
2                                 Engine    0.025477
6          Kilometers_Driven_Transformed    0.005342
1                                Mileage    0.005087
14                      Location_Kolkata    0.004191
51                            Brand_Tata    0.003903
52                          Brand_Toyota    0.003087
352                   Model_Cayenne Base    0.002901
11                    Location_Hyderabad    0.002753
9                    Location_Coimbatore    0.001609
32                            Brand_Ford    0.001552
4                                  Seats    0.001341
7                     Location_Bangalore    0.001261
28                       Brand_Chevrolet    0.000846
1120  Model_New Safari DICOR 2.2 VX 4x2    0.000712
17                      Fuel_Type_Diesel    0.000685
10                        Location_Delhi    0.000647
27                          Brand_Bentley    0.000633
```



In this plot, it is shown that the most important features are:

- Power
- Year

- New_price
- Engine

Hyperparameter Tuning: Random Forest

In [189…
```
# Check the best parameters
params = {
    'n_estimators': [200, 300],
    'max_depth': [10, 20],
    'min_samples_split': [ 5,10],
    'min_samples_leaf': [ 2, 4]
}
folds = KFold(n_splits = 10, shuffle = True, random_state = 1)

forest = RandomForestRegressor()

forest_cv = GridSearchCV(estimator=forest, param_grid=params, cv=5, scoring=
forest_cv.fit(X_train, Y_train)
```

Out[189…
```
▸ GridSearchCV
▸ estimator: RandomForestRegressor
    ▸ RandomForestRegressor
```

In [190…
```
# Display best parameters
forest_cv.best_params_
```

Out[190…
```
{'max_depth': 20,
 'min_samples_leaf': 2,
 'min_samples_split': 5,
 'n_estimators': 200}
```

In [192…
```
# Build model using the previous parameters
forest_model_1 = RandomForestRegressor(max_depth= 20,
 min_samples_leaf= 2,
 min_samples_split= 5,
 n_estimators= 200)
forest_model_1.fit(X_train,Y_train)
```

Out[192…
```
▾ RandomForestRegressor
RandomForestRegressor(max_depth=20, min_samples_leaf=2, min_samples
_split=5,
                      n_estimators=200)
```

In [193…
```
# Evaluate model
model_performance(forest_model_1, X_train, X_test, Y_train, Y_test)
```

```
    Data      RMSE       MAE          MAPE        r2
0  Train  0.117591  0.076429  1.388294e+12  0.981768
1   Test  0.194037  0.138088  1.212597e+12  0.951807
```

In [194…

```python
# Compare with default parameters
forest_model_2 = RandomForestRegressor()
forest_model_2.fit(X_train, Y_train)
```

Out[194…

```
▾ RandomForestRegressor
RandomForestRegressor()
```

In [195…

```python
# Evaluate model
model_performance(forest_model_2, X_train, X_test, Y_train, Y_test)
```

```
    Data      RMSE       MAE          MAPE        r2
0  Train  0.079750  0.053362  1.095007e+12  0.991614
1   Test  0.189722  0.133691  1.523558e+12  0.953926
```

In [197…

```python
# Get parameters
forest_model_2.get_params()
```

Out[197…

```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'criterion': 'squared_error',
 'max_depth': None,
 'max_features': 1.0,
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

Observations

- Default parameters have a better score

In [198…

```python
# Function to cross validate Decision Tree
def cross_validate_forest(X, y, k=10, max_depth = None, min_samples_leaf = 1
    """
    Perform cross-validation for Ridge regression.
    X: Features
    y: Target variable
    alpha: Regularization strength
    k: Number of folds for cross-validation
    """
    kf = KFold(n_splits=k, shuffle=True, random_state=1)
```

```
    r2_scores = []
    rmse_scores = []

    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        model = RandomForestRegressor(max_depth = max_depth, min_samples_lea
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

        r2_scores.append(r2_score(y_test, y_pred))
        rmse_scores.append(mean_squared_error(y_test, y_pred, squared=False)

    mean_r2 = np.mean(r2_scores)
    std_r2 = np.std(r2_scores)
    mean_rmse = np.mean(rmse_scores)
    std_rmse = np.std(rmse_scores)

    return mean_r2, std_r2, mean_rmse, std_rmse
```

In [199…

```
# Use cross validation function for different values of k
results = []
for k in range(2, 10):
    mean_r2, std_r2, mean_mrse, std_mrse = cross_validate_forest(X.values, Y
    results.append((k, mean_r2, 2 * std_r2, mean_mrse, 2 * std_mrse))

results_df = pd.DataFrame(results, columns=['k', 'R-squared', ' +/-', 'MRSE'
print(results_df)
```

```
   k  R-squared       +/-      MRSE       +/-
0  2   0.934036  0.005822  0.224330  0.013445
1  3   0.939588  0.011709  0.214475  0.027168
2  4   0.941751  0.013736  0.210611  0.032783
3  5   0.943286  0.015306  0.207596  0.036372
4  6   0.943299  0.016442  0.207527  0.036172
5  7   0.942712  0.025025  0.208009  0.046738
6  8   0.943359  0.019398  0.207269  0.040944
7  9   0.943616  0.031622  0.205814  0.059160
```

This model is able to account for ~94% of variation.

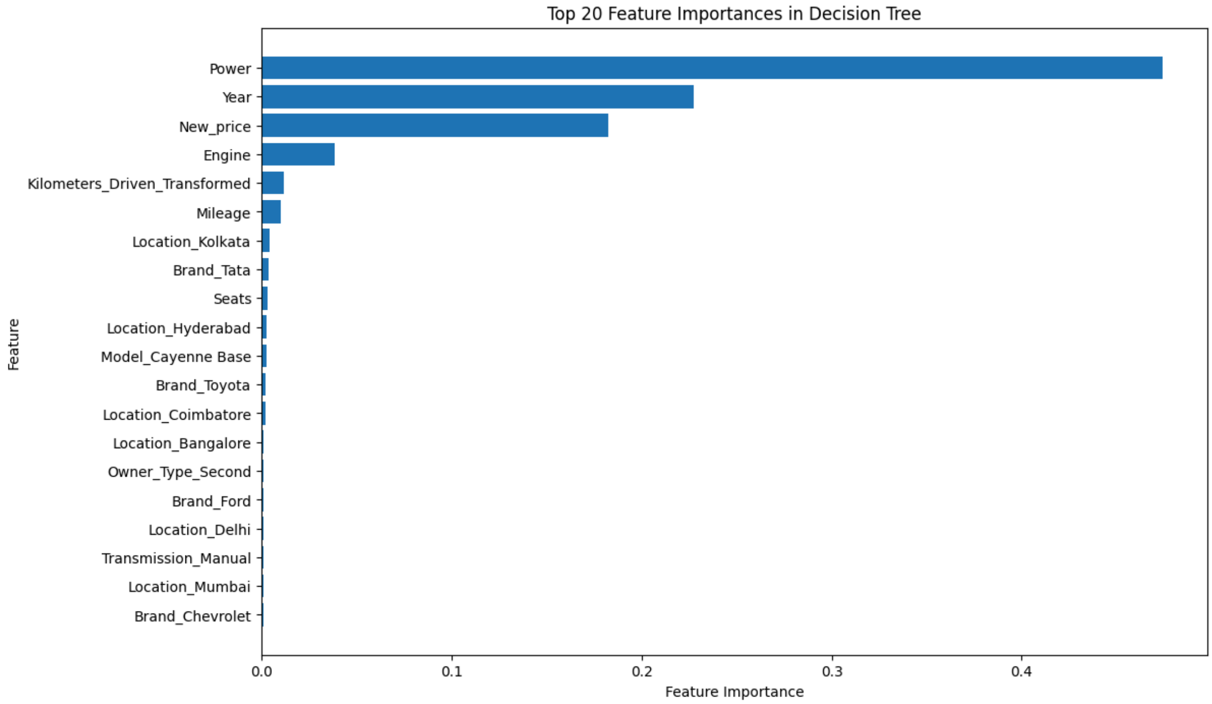
Feature Importance

In [200…

```
# Access feature importances
feature_importances = forest_model_2.feature_importances_

# Create a DataFrame for better readability
features = X.columns
importance_df = pd.DataFrame({
    'Feature': features,
    'Importance': feature_importances
})

# Sort the DataFrame by importance
```

```
importance_df = importance_df.sort_values(by='Importance', ascending=False)

# Select the top 20 features
top_20_features = importance_df.head(20)

# Display the DataFrame
print(top_20_features)

# Plot the feature importances of the top 20 features
plt.figure(figsize=(12, 8))
plt.barh(top_20_features['Feature'], top_20_features['Importance'])
plt.xlabel('Feature Importance')
plt.ylabel('Feature')
plt.title('Top 20 Feature Importances in Decision Tree')
plt.gca().invert_yaxis()
plt.show()
```

```
                          Feature  Importance
3                           Power    0.473914
0                            Year    0.227295
5                       New_price    0.182488
2                          Engine    0.038705
6     Kilometers_Driven_Transformed    0.011692
1                         Mileage    0.010077
14               Location_Kolkata    0.004439
51                     Brand_Tata    0.003874
4                           Seats    0.003037
11             Location_Hyderabad    0.002861
352            Model_Cayenne Base    0.002683
52                   Brand_Toyota    0.002424
9             Location_Coimbatore    0.001981
7              Location_Bangalore    0.001275
23              Owner_Type_Second    0.001268
32                     Brand_Ford    0.001238
10                 Location_Delhi    0.001128
21            Transmission_Manual    0.001125
15                Location_Mumbai    0.001082
28                Brand_Chevrolet    0.000965
```



The most important features for the random forest regression are:

- Power
- Year
- New price
- Engine

These are the same as in the decision tree regression.

Conclusions and Recommendations

1. Comparison of various techniques and their relative performance based on chosen Metric (Measure of success):

The techniques have similar performances. The primary metric used for evaluating performance was r2. The ranking is the following

- 1. Ridge Regression r2 of ~95%
- 2. Random forest r2 of ~94%
- 3. Decision Tree r2 of ~91%
- 4. Linear regression r2 of ~87%

2. Refined insights:

All the models are able to make predictions of the price of cars. However, the best models are Ridge Regression and Random forest regression.

The reason is that there is multicollinearity between variables and these two models are robust to these conditions.

The data had to be processed so it would be more effective for training models, a couple transformations were made (sqrt and log) to achieve normality in skewed variables.

3. Proposal for the final solution design:

The best model to adopt is the random forest regression, even though it is slightly worse than the ridge regression. It is far more customizable in the sense that there are more hyper parameters to adjust, the downside is that it takes more resources and time.

This model is also more interpretable since it has the 'Importance' feature, in the analysis made the top 4 features were Power, Engine, Year and New price.

Benefits

- Improved pricing accuracy.
- Increased sales.
- Improved customer satisfaction.

Costs

- Initial setup costs for model deployment and infrastructure.
- Ongoing costs for data maintenance and model retraining.