Linear Regression

If you are consulting an automobile company, you are trying to understand the factors that influence the sale price of the cars. Specifically, which factors drive the car prices up? And how accurately can you predict the sale price based on the car's features?

In this notebook, we will perform a simple linear regression analysis on a car price dataset, show how this prediction analysis is done and what are the important assumptions that must be satisfied for linear regression. We will also look at different ways to transform our data.

Objectives

After completing this lab you will be able to:

- Select the significant features based on the visual analysis
- Check the assumptions for Linear Regression model
- Apply the Linear Regression model and make the predictions
- Apply the pipelines to transform the data

Setup

For this lab, we will be using the following libraries:

- `pandas` for managing the data.
- `numpy` for mathematical operations.
- `seaborn` for visualizing the data.
- `matplotlib` for visualizing the data.
- `sklearn` for machine learning and machine-learning-pipeline related functions.
- `scipy` for statistical computations.

Import the required libraries

The following required modules are pre-installed in the Skills Network Labs environment. However if you run this notebook commands in a different Jupyter environment (e.g. Watson Studio or Ananconda) you will need to install these libraries by removing the `#` sign before `!mamba` in the code cell below.

In [1]:
```
!mamba install -qy pandas==1.3.4 numpy==1.21.4 seaborn==0.9.0 matplotlib==3.5.0 sci
# Note: If your environment doesn't support "!mamba install", use "!pip install"
```

```
Could not solve for environment specs
The following packages are incompatible
├─ matplotlib 3.5.0  is installable with the potential options
│  ├─ matplotlib [2.2.2|3.1.2|...|3.5.3] would require
│  │  └─ pyqt [ |>=5.6,<6.0a0 ] with the potential options
│  │     ├─ pyqt 5.6.0 would require
│  │     │  └─ qt 5.6.*  with the potential options
│  │     │     ├─ qt 5.6.2 would require
│  │     │     │  ├─ gst-plugins-base >=1.12.2,<1.13.0a0 , which requires
│  │     │     │  │  └─ gstreamer [>=1.12.2,<1.13.0a0 |>=1.12.4,<1.13.0a0 ], which r
equires
│  │     │     │  │     └─ glib >=2.53.6,<3.0a0 , which can be installed;
│  │     │     │  └─ openssl 1.0.* , which can be installed;
│  │     │     ├─ qt 5.6.2 would require
│  │     │     │  ├─ glib >=2.53.6,<3.0a0 , which can be installed;
│  │     │     │  └─ openssl >=1.0.2n,<1.0.3a , which can be installed;
│  │     │     ├─ qt 5.6.3 would require
│  │     │     │  ├─ glib >=2.56.1,<3.0a0 , which can be installed;
│  │     │     │  └─ openssl >=1.0.2o,<1.0.3a , which can be installed;
│  │     │     ├─ qt 5.6.3 would require
│  │     │     │  ├─ glib >=2.56.1,<3.0a0 , which can be installed;
│  │     │     │  └─ openssl >=1.0.2p,<1.0.3a , which can be installed;
│  │     │     └─ qt [5.6.3|5.9.7] would require
│  │     │        ├─ fontconfig >=2.13.0,<3.0a0  with the potential options
│  │     │        │  ├─ fontconfig 2.14.2 would require
│  │     │        │  │  └─ freetype >=2.12.1,<3.0a0 , which can be installed;
│  │     │        │  ├─ fontconfig [2.13.0|2.13.1] would require
│  │     │        │  │  └─ libuuid >=1.0.3,<2.0a0 , which can be installed;
│  │     │        │  ├─ fontconfig 2.14.1 would require
│  │     │        │  │  └─ libuuid >=1.41.5,<2.0a0 , which can be installed;
│  │     │        │  └─ fontconfig 2.14.1 would require
│  │     │        │     └─ freetype >=2.10.4,<3.0a0 , which can be installed;
│  │     │        └─ glib >=2.56.2,<3.0a0 , which can be installed;
│  │     ├─ pyqt [5.15.10|5.15.7|5.9.2] would require
│  │     │  └─ python >=3.10,<3.11.0a0 , which can be installed;
│  │     ├─ pyqt [5.15.10|5.15.7] would require
│  │     │  └─ python >=3.11,<3.12.0a0 , which can be installed;
│  │     ├─ pyqt 5.15.10 would require
│  │     │  └─ python >=3.12,<3.13.0a0 , which can be installed;
│  │     ├─ pyqt [5.15.10|5.15.7|5.9.2] would require
│  │     │  └─ python >=3.8,<3.9.0a0 , which can be installed;
│  │     ├─ pyqt [5.15.10|5.15.7|5.9.2] would require
│  │     │  └─ python >=3.9,<3.10.0a0 , which can be installed;
│  │     ├─ pyqt 5.15.7 would require
│  │     │  └─ qtwebkit 5.* , which requires
│  │     │     └─ glib >=2.69.1,<3.0a0 , which can be installed;
│  │     ├─ pyqt [5.6.0|5.9.2] would require
│  │     │  └─ python >=2.7,<2.8.0a0 , which can be installed;
│  │     ├─ pyqt [5.6.0|5.9.2] would require
│  │     │  └─ python >=3.5,<3.6.0a0 , which can be installed;
│  │     ├─ pyqt [5.6.0|5.9.2] would require
│  │     │  └─ python >=3.6,<3.7.0a0 , which can be installed;
│  │     └─ pyqt 5.9.2 would require
│  │        └─ qt [5.9.* |>=5.9.6,<5.10.0a0 ] with the potential options
│  │           ├─ qt [5.6.3|5.9.7], which can be installed (as previously explaine
d);
```

```
│  │           ├─ qt 5.9.6 would require
│  │           │  ├─ glib >=2.56.1,<3.0a0 , which can be installed;
│  │           │  └─ openssl 1.0.* , which can be installed;
│  │           └─ qt [5.9.4|5.9.5] would require
│  │              └─ openssl 1.0.* , which can be installed;
│  ├─ matplotlib [3.5.0|3.5.1|...|3.8.0] would require
│  │  └─ python >=3.10,<3.11.0a0 , which can be installed;
│  ├─ matplotlib [3.1.1|3.1.2|...|3.7.2] would require
│  │  └─ python >=3.8,<3.9.0a0 , which can be installed;
│  └─ matplotlib [3.3.4|3.4.2|...|3.8.0] would require
│     └─ python >=3.9,<3.10.0a0 , which can be installed;
├─ numpy 1.21.4  does not exist (perhaps a typo or a missing channel);
└─ seaborn 0.9.0  is installable with the potential options
   ├─ seaborn 0.9.0 would require
   │  └─ matplotlib >=1.4.3  with the potential options
   │     ├─ matplotlib [2.2.2|3.1.2|...|3.5.3], which can be installed (as previousl
y explained);
   │     ├─ matplotlib [3.5.0|3.5.1|...|3.8.0], which can be installed (as previousl
y explained);
   │     ├─ matplotlib [3.1.1|3.1.2|...|3.7.2], which can be installed (as previousl
y explained);
   │     ├─ matplotlib [3.3.4|3.4.2|...|3.8.0], which can be installed (as previousl
y explained);
   │     ├─ matplotlib [2.0.2|2.1.0|...|2.2.3] would require
   │     │  └─ python >=2.7,<2.8.0a0 , which can be installed;
   │     ├─ matplotlib [2.0.2|2.1.0|...|3.0.0] would require
   │     │  └─ python >=3.5,<3.6.0a0 , which can be installed;
   │     ├─ matplotlib [2.0.2|2.1.0|...|3.3.4] would require
   │     │  └─ python >=3.6,<3.7.0a0 , which can be installed;
   │     ├─ matplotlib [2.2.3|3.0.0|...|3.1.2] would require
   │     │  └─ pyqt 5.9.* , which can be installed (as previously explained);
   │     ├─ matplotlib [3.6.2|3.7.1|3.7.2|3.8.0] would require
   │     │  └─ python >=3.11,<3.12.0a0 , which can be installed;
   │     └─ matplotlib 3.8.0 would require
   │        └─ python >=3.12,<3.13.0a0 , which can be installed;
   ├─ seaborn 0.9.0 would require
   │  └─ python >=2.7,<2.8.0a0 , which can be installed;
   ├─ seaborn 0.9.0 would require
   │  └─ python >=3.5,<3.6.0a0 , which can be installed;
   └─ seaborn 0.9.0 would require
      └─ python >=3.6,<3.7.0a0 , which can be installed.
```

In [7]:
```
!pip install tqdm
```

```
Requirement already satisfied: tqdm in /home/jupyterlab/conda/envs/python/lib/python
3.7/site-packages (4.60.0)
```

In [8]:
```
!pip install -U scikit-learn
# import piplite
# await piplite.install(['tqdm', 'seaborn', 'skillsnetwork', 'pandas', 'numpy', 'sc
```

```
Requirement already satisfied: scikit-learn in /home/jupyterlab/conda/envs/python/li
b/python3.7/site-packages (1.0.2)
Requirement already satisfied: numpy>=1.14.6 in /home/jupyterlab/conda/envs/python/l
ib/python3.7/site-packages (from scikit-learn) (1.21.6)
Requirement already satisfied: scipy>=1.1.0 in /home/jupyterlab/conda/envs/python/li
b/python3.7/site-packages (from scikit-learn) (1.7.3)
Requirement already satisfied: joblib>=0.11 in /home/jupyterlab/conda/envs/python/li
b/python3.7/site-packages (from scikit-learn) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /home/jupyterlab/conda/envs/p
ython/lib/python3.7/site-packages (from scikit-learn) (3.1.0)
```

In [9]:
```
from tqdm import tqdm
import skillsnetwork
import numpy as np
import pandas as pd
from itertools import accumulate
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_digits, load_wine
from scipy.stats import boxcox
from scipy.stats.mstats import normaltest

from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import Pipeline
```

In [10]:
```
import sklearn; print("Scikit-Learn", sklearn.__version__)
```

```
Scikit-Learn 1.0.2
```

In [11]:
```
# Surpress warnings:
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

Reading and understanding our data

For this lab, we will be using the car sales dataset, hosted on IBM Cloud object storage. The dataset contains all the information about cars, the name of the manufacturer, the year it was launched, all car technical parameters, and the sale price.

Let's read the data into *pandas* data frame and look at the first 5 rows using the `head()` method.

In [ ]:
```
# That's the file path
""" https://cf-courses-data.s3.us.cloud-object-
  storage.appdomain.cloud/IBM-ML240EN-SkillsNetwork/labs/data/CarPrice_Assignment.c
```

In [12]:
```
URL = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-ML240

await skillsnetwork.download_dataset(URL)
print('file downloaded')

data = pd.read_csv('CarPrice_Assignment.csv')
data.head()
```

```
Downloading CarPrice_Assignment.csv:   0%|          | 0/26511 [00:00<?, ?it/s]
Saved as 'CarPrice_Assignment.csv'
file downloaded
```

Out[12]:

	car_ID	symboling	CarName	fueltype	aspiration	doornumber	carbody	drivewhe
0	1	3	alfa-romero giulia	gas	std	two	convertible	rv
1	2	3	alfa-romero stelvio	gas	std	two	convertible	rv
2	3	1	alfa-romero Quadrifoglio	gas	std	two	hatchback	rv
3	4	2	audi 100 ls	gas	std	four	sedan	fv
4	5	2	audi 100ls	gas	std	four	sedan	4v

5 rows × 26 columns

We can find more information about the features and types using the `info()` method.

In [13]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 26 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   car_ID            205 non-null    int64  
 1   symboling         205 non-null    int64  
 2   CarName           205 non-null    object 
 3   fueltype          205 non-null    object 
 4   aspiration        205 non-null    object 
 5   doornumber        205 non-null    object 
 6   carbody           205 non-null    object 
 7   drivewheel        205 non-null    object 
 8   enginelocation    205 non-null    object 
 9   wheelbase         205 non-null    float64
 10  carlength         205 non-null    float64
 11  carwidth          205 non-null    float64
 12  carheight         205 non-null    float64
 13  curbweight        205 non-null    int64  
 14  enginetype        205 non-null    object 
 15  cylindernumber    205 non-null    object 
 16  enginesize        205 non-null    int64  
 17  fuelsystem        205 non-null    object 
 18  boreratio         205 non-null    float64
 19  stroke            205 non-null    float64
 20  compressionratio  205 non-null    float64
 21  horsepower        205 non-null    int64  
 22  peakrpm           205 non-null    int64  
 23  citympg           205 non-null    int64  
 24  highwaympg        205 non-null    int64  
 25  price             205 non-null    float64
dtypes: float64(8), int64(8), object(10)
memory usage: 41.8+ KB
```

According to the output above, we have 205 entries or rows, as well as 26 features. The "Non-Null Count" column shows the number of non-null entries. If the count is 205 then there is no missing values for that particular feature. The 'price' is our target, or response variable, and the rest of the features are our predictor variables.

We also have a mix of numerical (8 int64 and 8 float64) and object data types (10 object).

The `describe()` function will provide the statistical information about all numeric values.

In [15]:
```
data.describe().T
```

Out[15]:

	count	mean	std	min	25%	50%	75%
car_ID	205.0	103.000000	59.322565	1.00	52.00	103.00	154.00
symboling	205.0	0.834146	1.245307	-2.00	0.00	1.00	2.00
wheelbase	205.0	98.756585	6.021776	86.60	94.50	97.00	102.40
carlength	205.0	174.049268	12.337289	141.10	166.30	173.20	183.10
carwidth	205.0	65.907805	2.145204	60.30	64.10	65.50	66.90
carheight	205.0	53.724878	2.443522	47.80	52.00	54.10	55.50
curbweight	205.0	2555.565854	520.680204	1488.00	2145.00	2414.00	2935.00
enginesize	205.0	126.907317	41.642693	61.00	97.00	120.00	141.00
boreratio	205.0	3.329756	0.270844	2.54	3.15	3.31	3.58
stroke	205.0	3.255415	0.313597	2.07	3.11	3.29	3.41
compressionratio	205.0	10.142537	3.972040	7.00	8.60	9.00	9.40
horsepower	205.0	104.117073	39.544167	48.00	70.00	95.00	116.00
peakrpm	205.0	5125.121951	476.985643	4150.00	4800.00	5200.00	5500.00
citympg	205.0	25.219512	6.542142	13.00	19.00	24.00	30.00
highwaympg	205.0	30.751220	6.886443	16.00	25.00	30.00	34.00
price	205.0	13276.710571	7988.852332	5118.00	7788.00	10295.00	16503.00

Data Cleaning and Wrangling

Here, we will check if we have any missing values.

In [16]:
```
data.isnull().sum()
```

Out[16]:
```
car_ID              0
symboling           0
CarName             0
fueltype            0
aspiration          0
doornumber          0
carbody             0
drivewheel          0
enginelocation      0
wheelbase           0
carlength           0
carwidth            0
carheight           0
curbweight          0
enginetype          0
cylindernumber      0
enginesize          0
fuelsystem          0
boreratio           0
stroke              0
compressionratio    0
horsepower          0
peakrpm             0
citympg             0
highwaympg          0
price               0
dtype: int64
```

Also, check for any duplicates by running `duplicated()` function through 'car_ID' records, since each row has a unique car ID value.

In [19]:
```
sum(data.duplicated(subset = 'car_ID')) == 0
```

Out[19]:
```
True
```

Next, let's look into some of our object variables first. Using `unique()` function, we will describe all categories of the 'CarName' attribute.

In [20]:
```
data["CarName"].unique()
```

Out[20]:
```
array(['alfa-romero giulia', 'alfa-romero stelvio',
       'alfa-romero Quadrifoglio', 'audi 100 ls', 'audi 100ls',
       'audi fox', 'audi 5000', 'audi 4000', 'audi 5000s (diesel)',
       'bmw 320i', 'bmw x1', 'bmw x3', 'bmw z4', 'bmw x4', 'bmw x5',
       'chevrolet impala', 'chevrolet monte carlo', 'chevrolet vega 2300',
       'dodge rampage', 'dodge challenger se', 'dodge d200',
       'dodge monaco (sw)', 'dodge colt hardtop', 'dodge colt (sw)',
       'dodge coronet custom', 'dodge dart custom',
       'dodge coronet custom (sw)', 'honda civic', 'honda civic cvcc',
       'honda accord cvcc', 'honda accord lx', 'honda civic 1500 gl',
       'honda accord', 'honda civic 1300', 'honda prelude',
       'honda civic (auto)', 'isuzu MU-X', 'isuzu D-Max ',
       'isuzu D-Max V-Cross', 'jaguar xj', 'jaguar xf', 'jaguar xk',
       'maxda rx3', 'maxda glc deluxe', 'mazda rx2 coupe', 'mazda rx-4',
       'mazda glc deluxe', 'mazda 626', 'mazda glc', 'mazda rx-7 gs',
       'mazda glc 4', 'mazda glc custom l', 'mazda glc custom',
       'buick electra 225 custom', 'buick century luxus (sw)',
       'buick century', 'buick skyhawk', 'buick opel isuzu deluxe',
       'buick skylark', 'buick century special',
       'buick regal sport coupe (turbo)', 'mercury cougar',
       'mitsubishi mirage', 'mitsubishi lancer', 'mitsubishi outlander',
       'mitsubishi g4', 'mitsubishi mirage g4', 'mitsubishi montero',
       'mitsubishi pajero', 'Nissan versa', 'nissan gt-r', 'nissan rogue',
       'nissan latio', 'nissan titan', 'nissan leaf', 'nissan juke',
       'nissan note', 'nissan clipper', 'nissan nv200', 'nissan dayz',
       'nissan fuga', 'nissan otti', 'nissan teana', 'nissan kicks',
       'peugeot 504', 'peugeot 304', 'peugeot 504 (sw)', 'peugeot 604sl',
       'peugeot 505s turbo diesel', 'plymouth fury iii',
       'plymouth cricket', 'plymouth satellite custom (sw)',
       'plymouth fury gran sedan', 'plymouth valiant', 'plymouth duster',
       'porsche macan', 'porcshce panamera', 'porsche cayenne',
       'porsche boxter', 'renault 12tl', 'renault 5 gtl', 'saab 99e',
       'saab 99le', 'saab 99gle', 'subaru', 'subaru dl', 'subaru brz',
       'subaru baja', 'subaru r1', 'subaru r2', 'subaru trezia',
       'subaru tribeca', 'toyota corona mark ii', 'toyota corona',
       'toyota corolla 1200', 'toyota corona hardtop',
       'toyota corolla 1600 (sw)', 'toyota carina', 'toyota mark ii',
       'toyota corolla', 'toyota corolla liftback',
       'toyota celica gt liftback', 'toyota corolla tercel',
       'toyota corona liftback', 'toyota starlet', 'toyota tercel',
       'toyota cressida', 'toyota celica gt', 'toyouta tercel',
       'vokswagen rabbit', 'volkswagen 1131 deluxe sedan',
       'volkswagen model 111', 'volkswagen type 3', 'volkswagen 411 (sw)',
       'volkswagen super beetle', 'volkswagen dasher', 'vw dasher',
       'vw rabbit', 'volkswagen rabbit', 'volkswagen rabbit custom',
       'volvo 145e (sw)', 'volvo 144ea', 'volvo 244dl', 'volvo 245',
       'volvo 264gl', 'volvo diesel', 'volvo 246'], dtype=object)
```

We can see that the 'CarName' includes both the company name (brand) and the car model. Next, we want to split a company name from the model of a car, as for our model building purpose, we will focus on a company name only.

In [21]:
```
data['brand'] = data.CarName.str.split(' ').str.get(0).str.lower()
```

Let's view all the `unique()` brands now.

In [22]:
```
data.brand.unique()
```

Out[22]:
```
array(['alfa-romero', 'audi', 'bmw', 'chevrolet', 'dodge', 'honda',
       'isuzu', 'jaguar', 'maxda', 'mazda', 'buick', 'mercury',
       'mitsubishi', 'nissan', 'peugeot', 'plymouth', 'porsche',
       'porcshce', 'renault', 'saab', 'subaru', 'toyota', 'toyouta',
       'vokswagen', 'volkswagen', 'vw', 'volvo'], dtype=object)
```

There are some typos in the names of the cars, so they should be corrected.

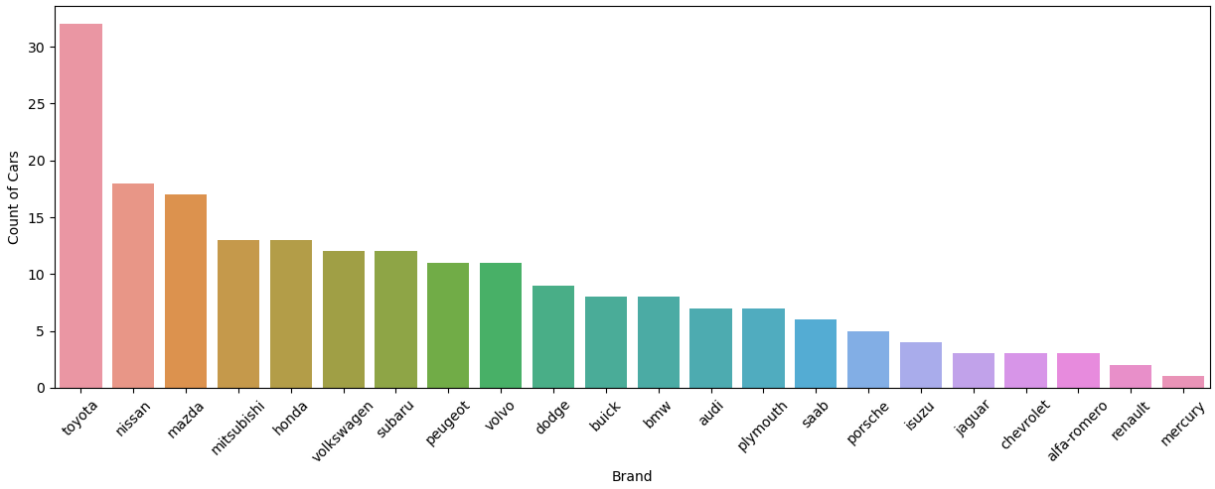
In [23]:
```
data['brand'] = data['brand'].replace(['vw', 'vokswagen'], 'volkswagen')
data['brand'] = data['brand'].replace(['maxda'], 'mazda')
data['brand'] = data['brand'].replace(['porcshce'], 'porsche')
data['brand'] = data['brand'].replace(['toyouta'], 'toyota')
```

In [24]:
```
data.brand.unique()
```

Out[24]:
```
array(['alfa-romero', 'audi', 'bmw', 'chevrolet', 'dodge', 'honda',
       'isuzu', 'jaguar', 'mazda', 'buick', 'mercury', 'mitsubishi',
       'nissan', 'peugeot', 'plymouth', 'porsche', 'renault', 'saab',
       'subaru', 'toyota', 'volkswagen', 'volvo'], dtype=object)
```

Let's plot and sort the total number of Brands.

In [27]:
```
fig, ax = plt.subplots(figsize = (15,5))
plt1 = sns.countplot(x=data['brand'], order=pd.value_counts(data['brand']).index)
plt1.set(xlabel = 'Brand', ylabel= 'Count of Cars')

# Setting angle for x-axis tick labels
plt.xticks(rotation=45)  # Adjust the rotation angle as needed

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




```
<Figure size 640x480 with 0 Axes>
```

We can drop `car_ID`, `symboling`, and `CarName` from our data frame, since they will no longer be needed.

In [ ]:
```
data.drop(['car_ID', 'symboling', 'CarName'],axis = 1, inplace = True)
```

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

In [ ]:
```
#If you need to save this partially processed data, uncomment the line below.
#data.to_csv('cleaned_car_data.csv', index=False)
```

Exercise 1

In this exercise, explore any (or all) object variables of your interest.

In [32]:
```
col_names = data.columns
```

In [34]:
```
# Loop through each object variable
for variable in col_names:
    # Display unique values
    unique_values = data[variable].unique()
    print(f"Unique values for {variable}:\n{unique_values}\n")
```

```
Unique values for car_ID:
[  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162
 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180
 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198
 199 200 201 202 203 204 205]

Unique values for symboling:
[ 3  1  2  0 -1 -2]

Unique values for CarName:
['alfa-romero giulia' 'alfa-romero stelvio' 'alfa-romero Quadrifoglio'
 'audi 100 ls' 'audi 100ls' 'audi fox' 'audi 5000' 'audi 4000'
 'audi 5000s (diesel)' 'bmw 320i' 'bmw x1' 'bmw x3' 'bmw z4' 'bmw x4'
 'bmw x5' 'chevrolet impala' 'chevrolet monte carlo' 'chevrolet vega 2300'
 'dodge rampage' 'dodge challenger se' 'dodge d200' 'dodge monaco (sw)'
 'dodge colt hardtop' 'dodge colt (sw)' 'dodge coronet custom'
 'dodge dart custom' 'dodge coronet custom (sw)' 'honda civic'
 'honda civic cvcc' 'honda accord cvcc' 'honda accord lx'
 'honda civic 1500 gl' 'honda accord' 'honda civic 1300' 'honda prelude'
 'honda civic (auto)' 'isuzu MU-X' 'isuzu D-Max ' 'isuzu D-Max V-Cross'
 'jaguar xj' 'jaguar xf' 'jaguar xk' 'maxda rx3' 'maxda glc deluxe'
 'mazda rx2 coupe' 'mazda rx-4' 'mazda glc deluxe' 'mazda 626' 'mazda glc'
 'mazda rx-7 gs' 'mazda glc 4' 'mazda glc custom l' 'mazda glc custom'
 'buick electra 225 custom' 'buick century luxus (sw)' 'buick century'
 'buick skyhawk' 'buick opel isuzu deluxe' 'buick skylark'
 'buick century special' 'buick regal sport coupe (turbo)'
 'mercury cougar' 'mitsubishi mirage' 'mitsubishi lancer'
 'mitsubishi outlander' 'mitsubishi g4' 'mitsubishi mirage g4'
 'mitsubishi montero' 'mitsubishi pajero' 'Nissan versa' 'nissan gt-r'
 'nissan rogue' 'nissan latio' 'nissan titan' 'nissan leaf' 'nissan juke'
 'nissan note' 'nissan clipper' 'nissan nv200' 'nissan dayz' 'nissan fuga'
 'nissan otti' 'nissan teana' 'nissan kicks' 'peugeot 504' 'peugeot 304'
 'peugeot 504 (sw)' 'peugeot 604sl' 'peugeot 505s turbo diesel'
 'plymouth fury iii' 'plymouth cricket' 'plymouth satellite custom (sw)'
 'plymouth fury gran sedan' 'plymouth valiant' 'plymouth duster'
 'porsche macan' 'porcshce panamera' 'porsche cayenne' 'porsche boxter'
 'renault 12tl' 'renault 5 gtl' 'saab 99e' 'saab 99le' 'saab 99gle'
 'subaru' 'subaru dl' 'subaru brz' 'subaru baja' 'subaru r1' 'subaru r2'
 'subaru trezia' 'subaru tribeca' 'toyota corona mark ii' 'toyota corona'
 'toyota corolla 1200' 'toyota corona hardtop' 'toyota corolla 1600 (sw)'
 'toyota carina' 'toyota mark ii' 'toyota corolla'
 'toyota corolla liftback' 'toyota celica gt liftback'
 'toyota corolla tercel' 'toyota corona liftback' 'toyota starlet'
 'toyota tercel' 'toyota cressida' 'toyota celica gt' 'toyouta tercel'
 'vokswagen rabbit' 'volkswagen 1131 deluxe sedan' 'volkswagen model 111'
 'volkswagen type 3' 'volkswagen 411 (sw)' 'volkswagen super beetle'
 'volkswagen dasher' 'vw dasher' 'vw rabbit' 'volkswagen rabbit'
 'volkswagen rabbit custom' 'volvo 145e (sw)' 'volvo 144ea' 'volvo 244dl'
```

```
 'volvo 245' 'volvo 264gl' 'volvo diesel' 'volvo 246']

Unique values for fueltype:
['gas' 'diesel']

Unique values for aspiration:
['std' 'turbo']

Unique values for doornumber:
['two' 'four']

Unique values for carbody:
['convertible' 'hatchback' 'sedan' 'wagon' 'hardtop']

Unique values for drivewheel:
['rwd' 'fwd' '4wd']

Unique values for enginelocation:
['front' 'rear']

Unique values for wheelbase:
[ 88.6  94.5  99.8  99.4 105.8  99.5 101.2 103.5 110.   88.4  93.7 103.3
  95.9  86.6  96.5  94.3  96.  113.  102.   93.1  95.3  98.8 104.9 106.7
 115.6  96.6 120.9 112.  102.7  93.   96.3  95.1  97.2 100.4  91.3  99.2
 107.9 114.2 108.   89.5  98.4  96.1  99.1  93.3  97.   96.9  95.7 102.4
 102.9 104.5  97.3 104.3 109.1]

Unique values for carlength:
[168.8 171.2 176.6 177.3 192.7 178.2 176.8 189.  193.8 197.  141.1 155.9
 158.8 157.3 174.6 173.2 144.6 150.  163.4 157.1 167.5 175.4 169.1 170.7
 172.6 199.6 191.7 159.1 166.8 169.  177.8 175.  190.9 187.5 202.6 180.3
 208.1 199.2 178.4 173.  172.4 165.3 170.2 165.6 162.4 173.4 181.7 184.6
 178.5 186.7 198.9 167.3 168.9 175.7 181.5 186.6 156.9 157.9 172.  173.5
 173.6 158.7 169.7 166.3 168.7 176.2 175.6 183.5 187.8 171.7 159.3 165.7
 180.2 183.1 188.8]

Unique values for carwidth:
[64.1 65.5 66.2 66.4 66.3 71.4 67.9 64.8 66.9 70.9 60.3 63.6 63.8 64.6
 63.9 64.  65.2 62.5 66.  61.8 69.6 70.6 64.2 65.7 66.5 66.1 70.3 71.7
 70.5 72.  68.  64.4 65.4 68.4 68.3 65.  72.3 66.6 63.4 65.6 67.7 67.2
 68.9 68.8]

Unique values for carheight:
[48.8 52.4 54.3 53.1 55.7 55.9 52.  53.7 56.3 53.2 50.8 50.6 59.8 50.2
 52.6 54.5 58.3 53.3 54.1 51.  53.5 51.4 52.8 47.8 49.6 55.5 54.4 56.5
 58.7 54.9 56.7 55.4 54.8 49.4 51.6 54.7 55.1 56.1 49.7 56.  50.5 55.2
 52.5 53.  59.1 53.9 55.6 56.2 57.5]

Unique values for curbweight:
[2548 2823 2337 2824 2507 2844 2954 3086 3053 2395 2710 2765 3055 3230
 3380 3505 1488 1874 1909 1876 2128 1967 1989 2191 2535 2811 1713 1819
 1837 1940 1956 2010 2024 2236 2289 2304 2372 2465 2293 2734 4066 3950
 1890 1900 1905 1945 1950 2380 2385 2500 2410 2443 2425 2670 2700 3515
 3750 3495 3770 3740 3685 3900 3715 2910 1918 1944 2004 2145 2370 2328
 2833 2921 2926 2365 2405 2403 1889 2017 1938 1951 2028 1971 2037 2008
 2324 2302 3095 3296 3060 3071 3139 3020 3197 3430 3075 3252 3285 3485
```

```
 3130 2818 2778 2756 2800 3366 2579 2460 2658 2695 2707 2758 2808 2847
 2050 2120 2240 2190 2340 2510 2290 2455 2420 2650 1985 2040 2015 2280
 3110 2081 2109 2275 2094 2122 2140 2169 2204 2265 2300 2540 2536 2551
 2679 2714 2975 2326 2480 2414 2458 2976 3016 3131 3151 2261 2209 2264
 2212 2319 2254 2221 2661 2563 2912 3034 2935 3042 3045 3157 2952 3049
 3012 3217 3062]

Unique values for enginetype:
['dohc' 'ohcv' 'ohc' 'l' 'rotor' 'ohcf' 'dohcv']

Unique values for cylindernumber:
['four' 'six' 'five' 'three' 'twelve' 'two' 'eight']

Unique values for enginesize:
[130 152 109 136 131 108 164 209  61  90  98 122 156  92  79 110 111 119
 258 326  91  70  80 140 134 183 234 308 304  97 103 120 181 151 194 203
 132 121 146 171 161 141 173 145]

Unique values for fuelsystem:
['mpfi' '2bbl' 'mfi' '1bbl' 'spfi' '4bbl' 'idi' 'spdi']

Unique values for boreratio:
[3.47 2.68 3.19 3.13 3.5  3.31 3.62 2.91 3.03 2.97 3.34 3.6  2.92 3.15
 3.43 3.63 3.54 3.08 3.33 3.39 3.76 3.58 3.46 3.8  3.78 3.17 3.35 3.59
 2.99 3.7  3.61 3.94 3.74 2.54 3.05 3.27 3.24 3.01]

Unique values for stroke:
[2.68  3.47  3.4   2.8   3.19  3.39  3.03  3.11  3.23  3.46  3.9   3.41
 3.07  3.58  4.17  2.76  3.15  3.255 3.16  3.64  3.1   3.35  3.12  3.86
 3.29  3.27  3.52  2.19  3.21  2.9   2.07  2.36  2.64  3.08  3.5   3.54
 2.87 ]

Unique values for compressionratio:
[ 9.   10.    8.    8.5   8.3   7.    8.8   9.5   9.6   9.41  9.4   7.6
  9.2  10.1   9.1   8.1  11.5   8.6  22.7  22.   21.5   7.5  21.9   7.8
  8.4  21.    8.7   9.31  9.3   7.7  22.5  23.  ]

Unique values for horsepower:
[111 154 102 115 110 140 160 101 121 182  48  70  68  88 145  58  76  60
  86 100  78  90 176 262 135  84  64 120  72 123 155 184 175 116  69  55
  97 152 200  95 142 143 207 288  73  82  94  62  56 112  92 161 156  52
  85 114 162 134 106]

Unique values for peakrpm:
[5000 5500 5800 4250 5400 5100 4800 6000 4750 4650 4200 4350 4500 5200
 4150 5600 5900 5750 5250 4900 4400 6600 5300]

Unique values for citympg:
[21 19 24 18 17 16 23 20 15 47 38 37 31 49 30 27 25 13 26 36 22 14 45 28
 32 35 34 29 33]

Unique values for highwaympg:
[27 26 30 22 25 20 29 28 53 43 41 38 24 54 42 34 33 31 19 17 23 32 39 18
 16 37 50 36 47 46]

Unique values for price:
```

```
[13495.    16500.    13950.    17450.    15250.    17710.    18920.
 23875.    17859.167 16430.    16925.    20970.    21105.    24565.
 30760.    41315.    36880.     5151.     6295.     6575.     5572.
  6377.     7957.     6229.     6692.     7609.     8558.     8921.
 12964.     6479.     6855.     5399.     6529.     7129.     7295.
  7895.     9095.     8845.    10295.    12945.    10345.     6785.
  8916.5   11048.    32250.    35550.    36000.     5195.     6095.
  6795.     6695.     7395.    10945.    11845.    13645.    15645.
  8495.    10595.    10245.    10795.    11245.    18280.    18344.
 25552.    28248.    28176.    31600.    34184.    35056.    40960.
 45400.    16503.     5389.     6189.     6669.     7689.     9959.
  8499.    12629.    14869.    14489.     6989.     8189.     9279.
  5499.     7099.     6649.     6849.     7349.     7299.     7799.
  7499.     7999.     8249.     8949.     9549.    13499.    14399.
 17199.    19699.    18399.    11900.    13200.    12440.    13860.
 15580.    16900.    16695.    17075.    16630.    17950.    18150.
 12764.    22018.    32528.    34028.    37028.    31400.5    9295.
  9895.    11850.    12170.    15040.    15510.    18620.     5118.
  7053.     7603.     7126.     7775.     9960.     9233.    11259.
  7463.    10198.     8013.    11694.     5348.     6338.     6488.
  6918.     7898.     8778.     6938.     7198.     7788.     7738.
  8358.     9258.     8058.     8238.     9298.     9538.     8449.
  9639.     9989.    11199.    11549.    17669.     8948.    10698.
  9988.    10898.    11248.    16558.    15998.    15690.    15750.
  7975.     7995.     8195.     9495.     9995.    11595.     9980.
 13295.    13845.    12290.    12940.    13415.    15985.    16515.
 18420.    18950.    16845.    19045.    21485.    22470.    22625.   ]

Unique values for brand:
['alfa-romero' 'audi' 'bmw' 'chevrolet' 'dodge' 'honda' 'isuzu' 'jaguar'
 'mazda' 'buick' 'mercury' 'mitsubishi' 'nissan' 'peugeot' 'plymouth'
 'porsche' 'renault' 'saab' 'subaru' 'toyota' 'volkswagen' 'volvo']
```

Next, we need to engineer some features, for better visualizations and analysis. We will group our data by 'brand', calculate the average price for each brand, and split these prices into 3 bins: `Budget`, `Mid-Range`, and `Luxury` cars, naming the newly created column - the `brand_category`.

In [35]:
```
data_comp_avg_price = data[['brand','price']].groupby('brand', as_index = False).me
```

In [36]:
```
data = data.merge(data_comp_avg_price, on = 'brand')
```

We will now check the statistics of our average car price per car brand.

In [37]:
```
data.brand_avg_price.describe()
```

Out[37]:
```
count      205.000000
mean     13276.710571
std       7154.179185
min       6007.000000
25%       9239.769231
50%      10077.500000
75%      15489.090909
max      34600.000000
Name: brand_avg_price, dtype: float64
```

In [38]:
```
data['brand_category'] = data['brand_avg_price'].apply(lambda x : "Budget" if x < 1
                                                       else ("Mid_Range" if 10000 <=
                                                             else "Luxury"))
```

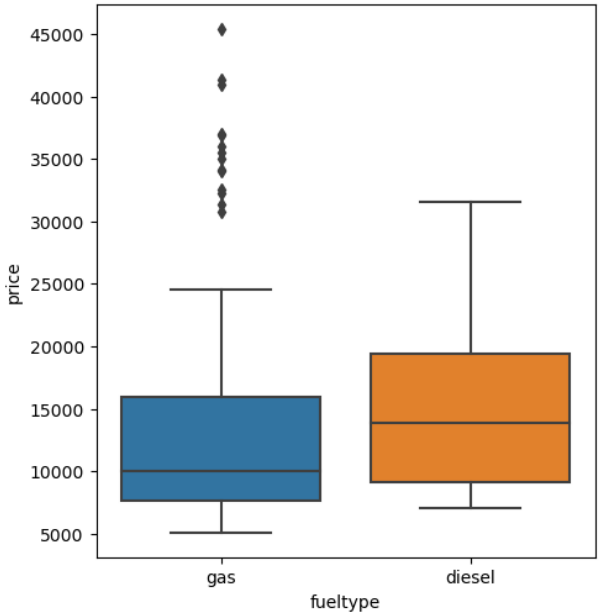
Exploratory Data Analysis

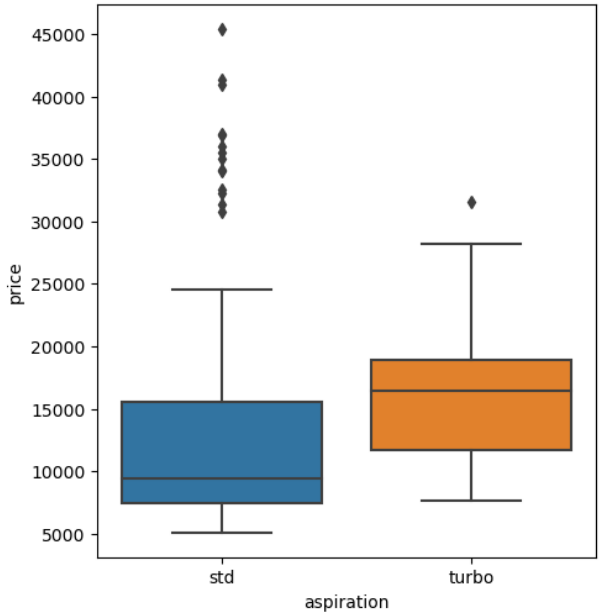
List of Categorical Variables:

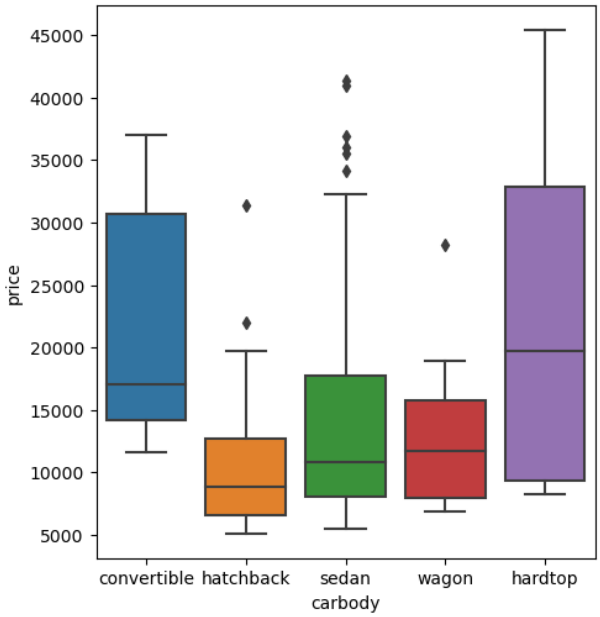
- brand_category
- fueltype
- enginetype
- carbody
- doornumber
- enginelocation
- fuelsystem
- cylindernumber
- aspiration
- drivewheel

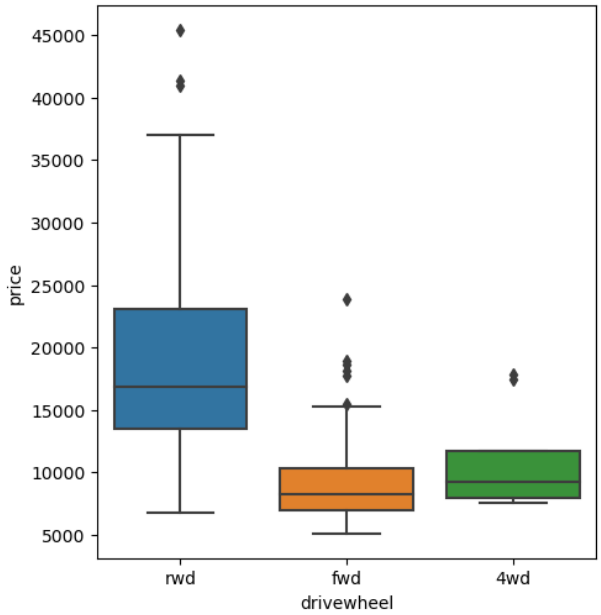
We will use the `boxplot()` function on the above mentioned categorical variables, to display the mean, variance, and possible outliers, with respect to the price.

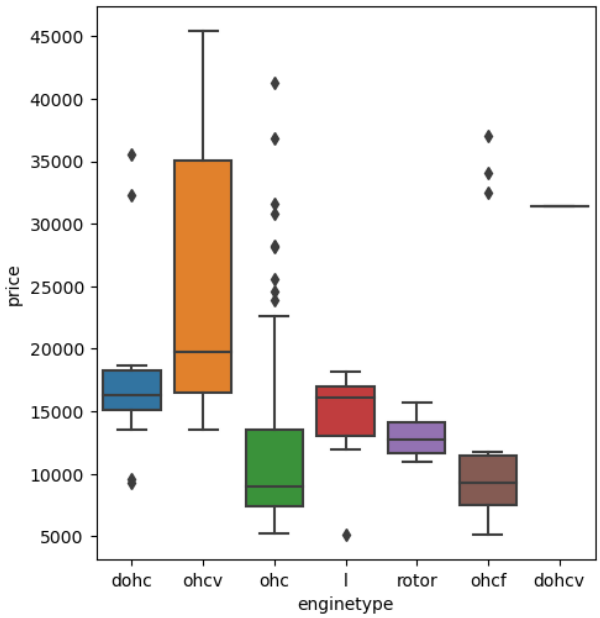
In [39]:
```
plt.figure(figsize=(10, 20))
plt.subplot(4,2,1)
sns.boxplot(x = 'fueltype', y = 'price', data = data)
plt.subplot(4,2,2)
sns.boxplot(x = 'aspiration', y = 'price', data = data)
plt.subplot(4,2,3)
sns.boxplot(x = 'carbody', y = 'price', data = data)
plt.subplot(4,2,4)
sns.boxplot(x = 'drivewheel', y = 'price', data = data)
plt.subplot(4,2,5)
sns.boxplot(x = 'enginetype', y = 'price', data = data)
plt.subplot(4,2,6)
sns.boxplot(x = 'brand_category', y = 'price', data = data)
plt.tight_layout()
plt.show()
```

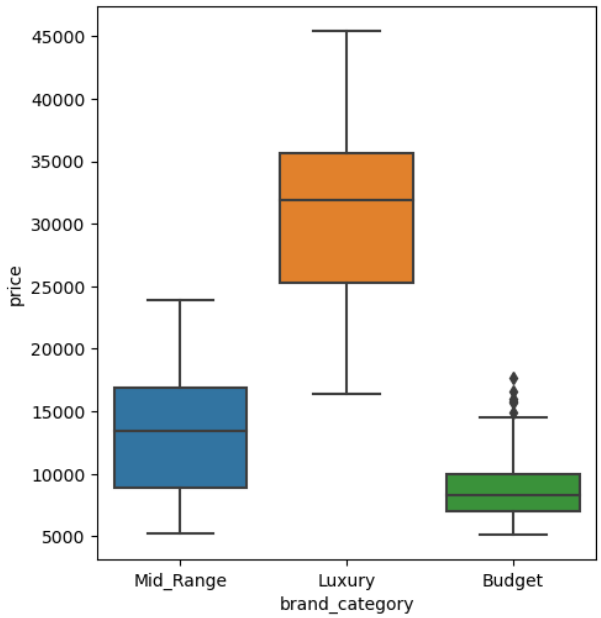

price
45000
40000
35000
30000
25000
20000
15000
10000
5000
gas
diesel
fueltype


price
45000
40000
35000
30000
25000
20000
15000
10000
5000
std
turbo
aspiration


price
45000
40000
35000
30000
25000
20000
15000
10000
5000
convertible
hatchback
sedan
wagon
hardtop
carbody


price
45000
40000
35000
30000
25000
20000
15000
10000
5000
rwd
fwd
4wd
drivewheel


price
45000
40000
35000
30000
25000
20000
15000
10000
5000
dohc
ohcv
ohc
l
rotor
ohcf
dohcv
enginetype


price
45000
40000
35000
30000
25000
20000
15000
10000
5000
Mid_Range
Luxury
Budget
brand_category

Next, let's view the list of top features that have high correlation coefficient. The `corr()` function calculates the Pearson's correlation coefficients with respect to the `price`.

In [40]:
```
corr_matrix = data.corr()
corr_matrix['price'].sort_values(ascending=False)
```

Out[40]:
```
price               1.000000
brand_avg_price     0.895520
enginesize          0.874145
curbweight          0.835305
horsepower          0.808139
carwidth            0.759325
carlength           0.682920
wheelbase           0.577816
boreratio           0.553173
carheight           0.119336
stroke              0.079443
compressionratio    0.067984
symboling          -0.079978
peakrpm            -0.085267
car_ID             -0.109093
citympg            -0.685751
highwaympg         -0.697599
Name: price, dtype: float64
```

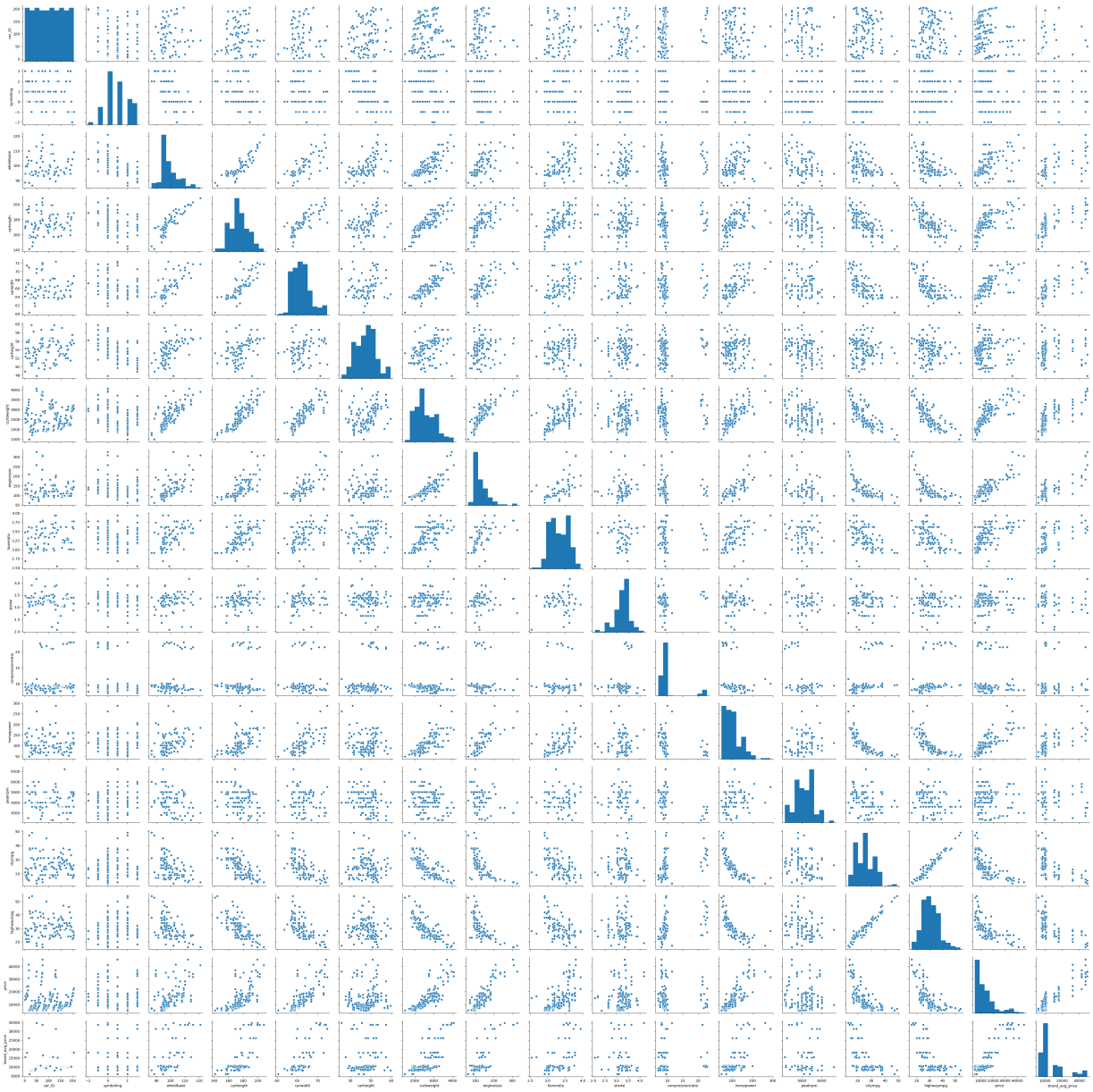
These are strongly correlated numerical features with Car Price.

We can also use the `heatmap()` or `pairplot()` to further explore the relationship between all features and the target variables.

Exercise 2

Use the `pairplot()` function to display the scatter plots of the relationships between the features. In output because of a lot of features it may give you a feeling of *ouch!!*

In [41]:
```
# Enter your code and run the cell
sns.pairplot(data)
plt.show()
```



▶ **Solution** (Click Here)

Testing Assumptions for Linear Regression

Since we fit a linear model, we assume that the relationship between the target (price) and other features is linear.

We also expect that the errors, or residuals, are pure random fluctuations around the true line, in other words, the variability in the response (dependent) variable doesn't increase as the value of the predictor (independent) variable increases. This is the assumption of equal variance, also known as *Homoscedasticity*.

We also assume that the observations are independent of one another (no *multicollinearity*), and there is no correlation between the sequential observations.

If we see one of these assumptions in the dataset are not met, it's more likely that the other ones, mentioned above, will also be violated. Luckily, we can check and fix these assumptions with a few unique techniques.

Now, let's briefly touch upon each of these assumptions in our example.

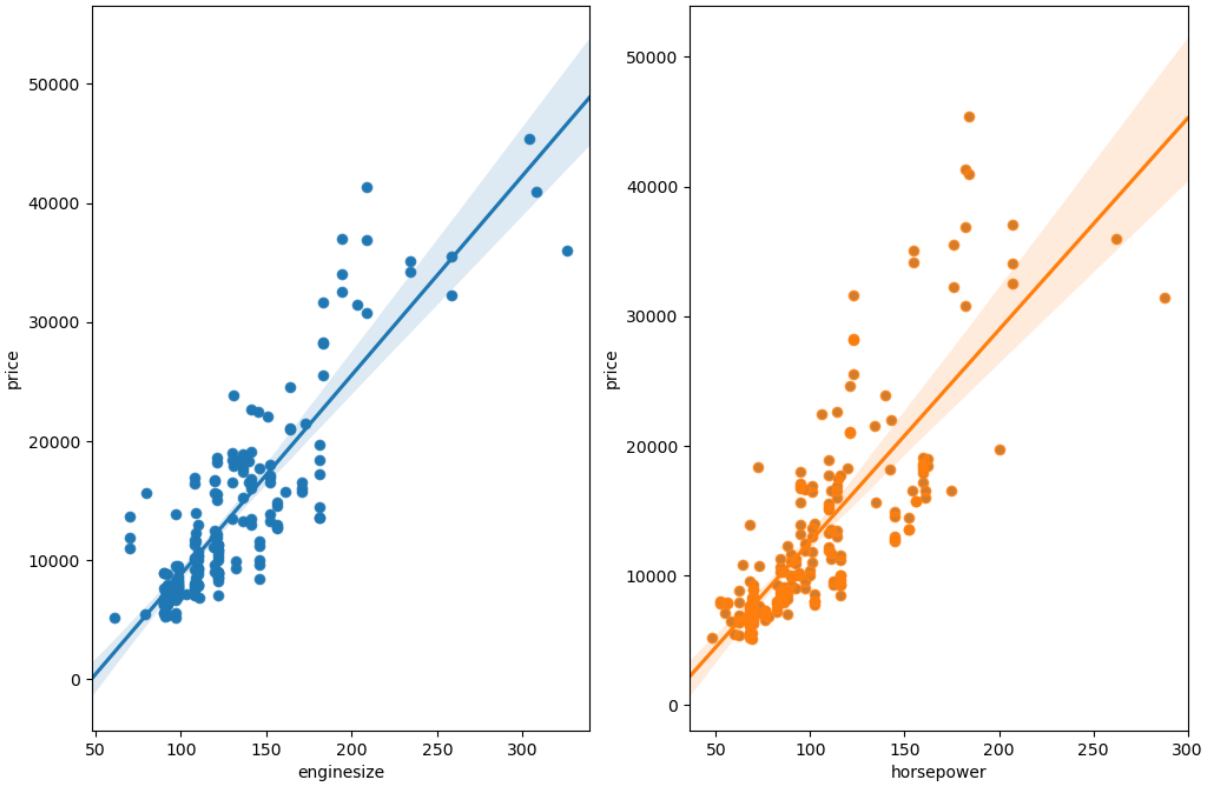
1. Linearity Assumption

Linear regression needs the relationship between independent variable and the dependent variable to be linear. We can test this assumption with some scatter plots and regression lines.

We will start with the 'enginesize' and 'horsepower' features.

In [42]:
```
fig, (ax1, ax2) = plt.subplots(figsize = (12,8), ncols=2,sharey=False)
sns.scatterplot( x = data.enginesize, y = data.price,  ax=ax1)
sns.regplot(x=data.enginesize, y=data.price, ax=ax1)

sns.scatterplot(x = data.horsepower,y = data.price, ax=ax2)
sns.regplot(x=data.horsepower, y=data.price, ax=ax2)
```

Out[42]:
```
<AxesSubplot:xlabel='horsepower', ylabel='price'>
```

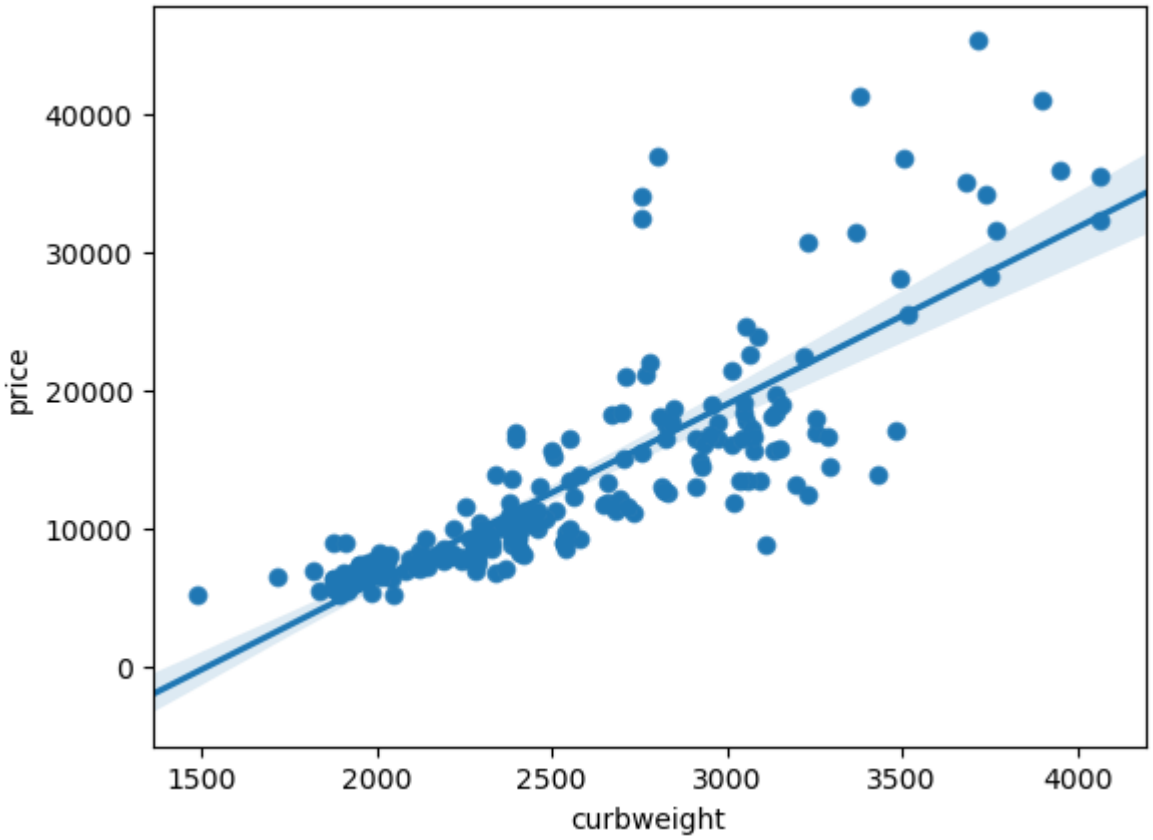


Exercise 3

In this Exercise, plot any other numeric features, using the *seaborn* `regplot()` function, to see whether there is any linear relationship between the feature and the 'price'.

In [51]:
```
# Enter your code and run the cell
sns.scatterplot(x=data.curbweight, y=data.price)
sns.regplot(x=data.curbweight, y=data.price)
```

Out[51]:
```
<AxesSubplot:xlabel='curbweight', ylabel='price'>
```



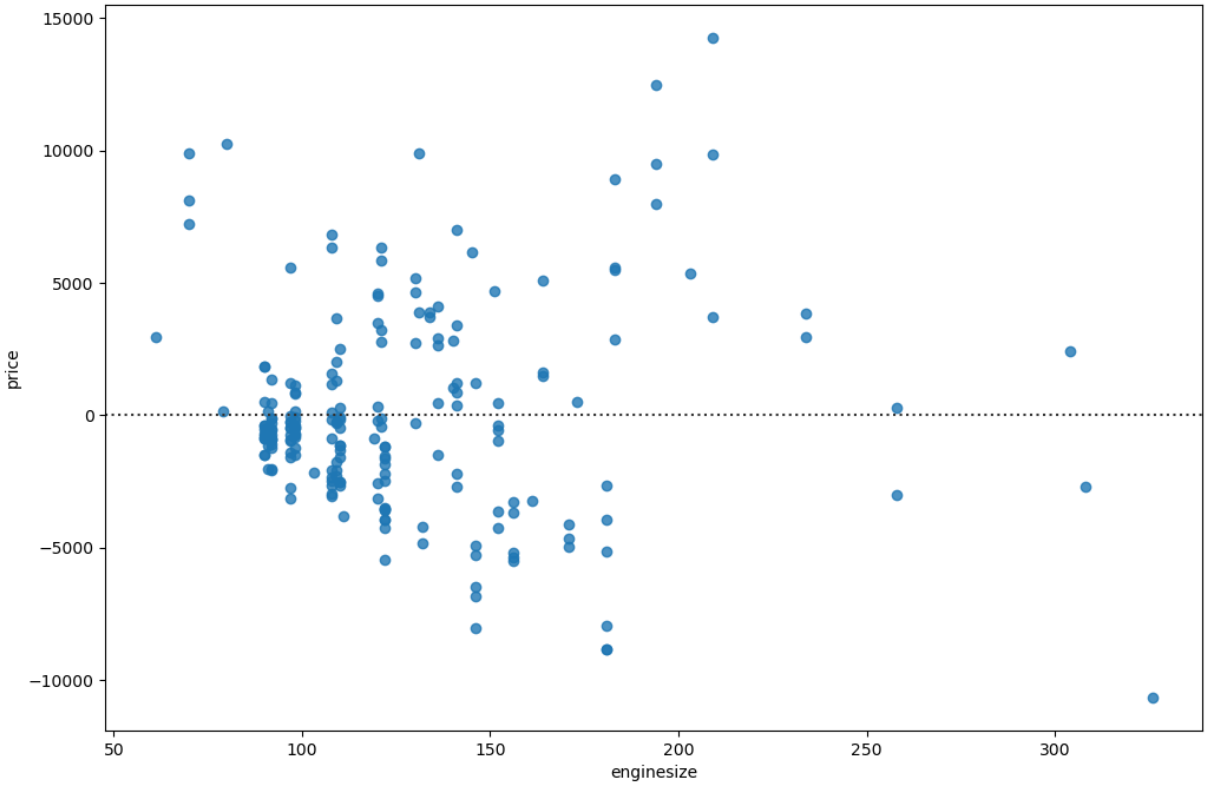
▶ **Solution** (Click Here)

2. *Homoscedasticity*

The assumption of *homoscedasticity* (constant variance), is crucial to linear regression models. *Homoscedasticity* describes a situation in which the error term or variance or the "noise" or random disturbance in the relationship between the independent variables and the dependent variable is the same across all values of the independent variable. In other words, there is a constant variance present in the response variable as the predictor variable increases. If the "noise" is not the same across the values of an independent variable, we call it *heteroscedasticity*, opposite of *homoscedasticity*.

In [52]:
```
plt.subplots(figsize = (12,8))
sns.residplot(x=data["enginesize"], y=data["price"])
```

Out[52]:
```
<AxesSubplot:xlabel='enginesize', ylabel='price'>
```



From the above plot, we can tell the error variance across the true line is dispersed somewhat not uniformly, but in a funnel like shape. So, the assumption of the *homoscedasticity* is more likely not met.

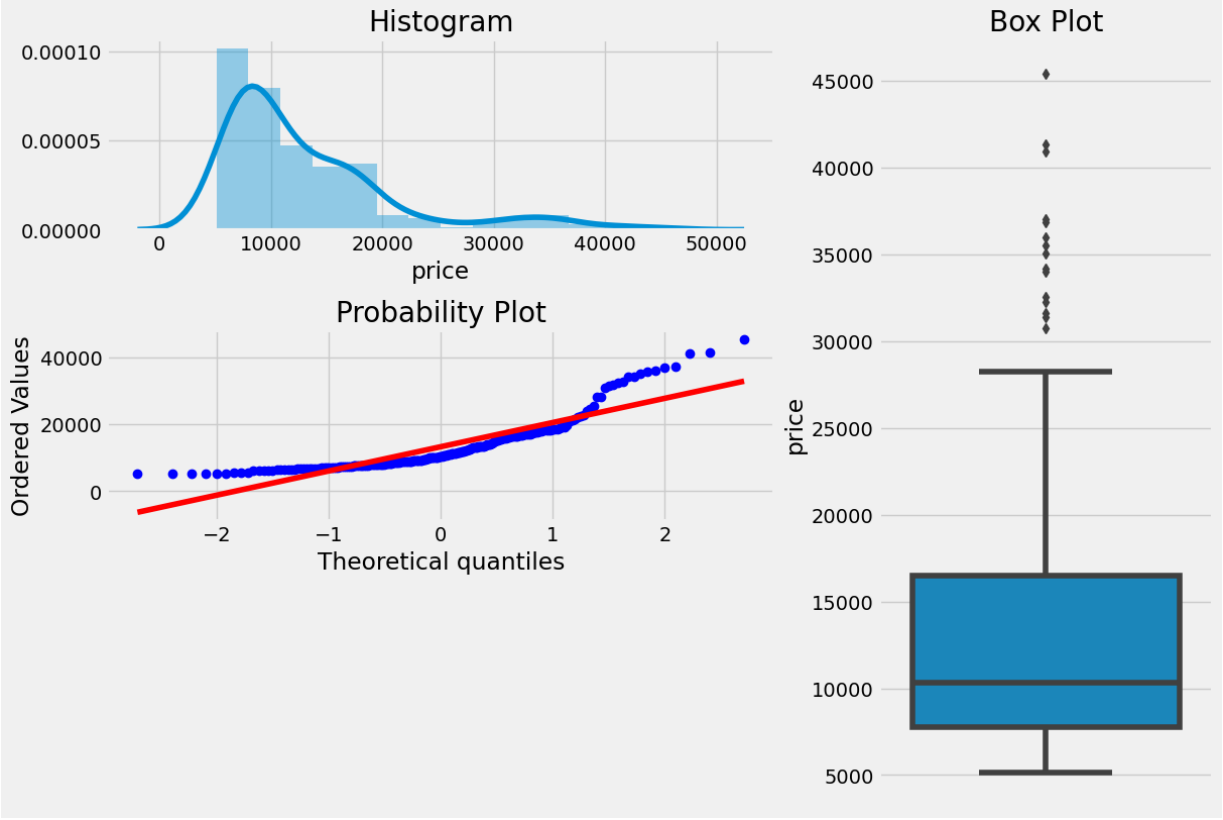
3. Normality

The linear regression analysis requires the dependent variable, 'price', to be normally distributed. A histogram, box plot, or a Q-Q-Plot can check if the target variable is normally distributed. The goodness of fit test, e.g., the Kolmogorov-Smirnov test can check for normality in the dependent variable. This documentation contains more information on the normality assumption.

Let's display all three charts to show how our target variable, 'price' behaves.

In [54]:
```
def plotting_3_chart(data, feature):
    ## Importing seaborn, matplotlab and scipy modules.
    import seaborn as sns
    import matplotlib.pyplot as plt
    import matplotlib.gridspec as gridspec
    from scipy import stats
    import matplotlib.style as style
    style.use('fivethirtyeight')

    ## Creating a customized chart. and giving in figsize and everything.
    fig = plt.figure(constrained_layout=True, figsize=(12,8))

    ## creating a grid of 3 cols and 3 rows.
    grid = gridspec.GridSpec(ncols=3, nrows=3, figure=fig)
```

```
    #gs = fig3.add_gridspec(3, 3)

    ## Customizing the histogram grid.
    ax1 = fig.add_subplot(grid[0, :2])

    ## Set the title.
    ax1.set_title('Histogram')

    ## plot the histogram.
    sns.distplot(data.loc[:,feature], norm_hist=True, ax = ax1)

    # customizing the QQ_plot.
    ax2 = fig.add_subplot(grid[1, :2])

    ## Set the title.
    ax2.set_title('QQ_plot')

    ## Plotting the QQ_Plot.
    stats.probplot(data.loc[:,feature], plot = ax2)

    ## Customizing the Box Plot.
    ax3 = fig.add_subplot(grid[:, 2])

    ## Set title.
    ax3.set_title('Box Plot')

    ## Plotting the box plot.
    sns.boxplot(data.loc[:,feature], orient='v', ax = ax3);

plotting_3_chart(data, 'price')
```



These three charts above can tell us a lot about our target variable:

- Our target variable, 'price' is not normally distributed
- Our target variable is right-skewed
- There are some outliers in the variable

The right-skewed plot means that most prices in the dataset are on the lower end (below 15,000). The 'max' value is very far from the '75%' quantile statistic. All these plots show that the assumption for accurate linear regression modeling is not met.

Next, we will perform the log transformation to correct our target variable and to make it more normally distributed.

But first, we will save our data that we have changed so far, in the 'previous_data' frame.

In [55]:
```
previous_data = data.copy()
```

Log Transformation

We can also check statistically if the target is normally distributed, using `normaltest()` function. If the p-value is large (>0.05), the target variable is normally distributed.

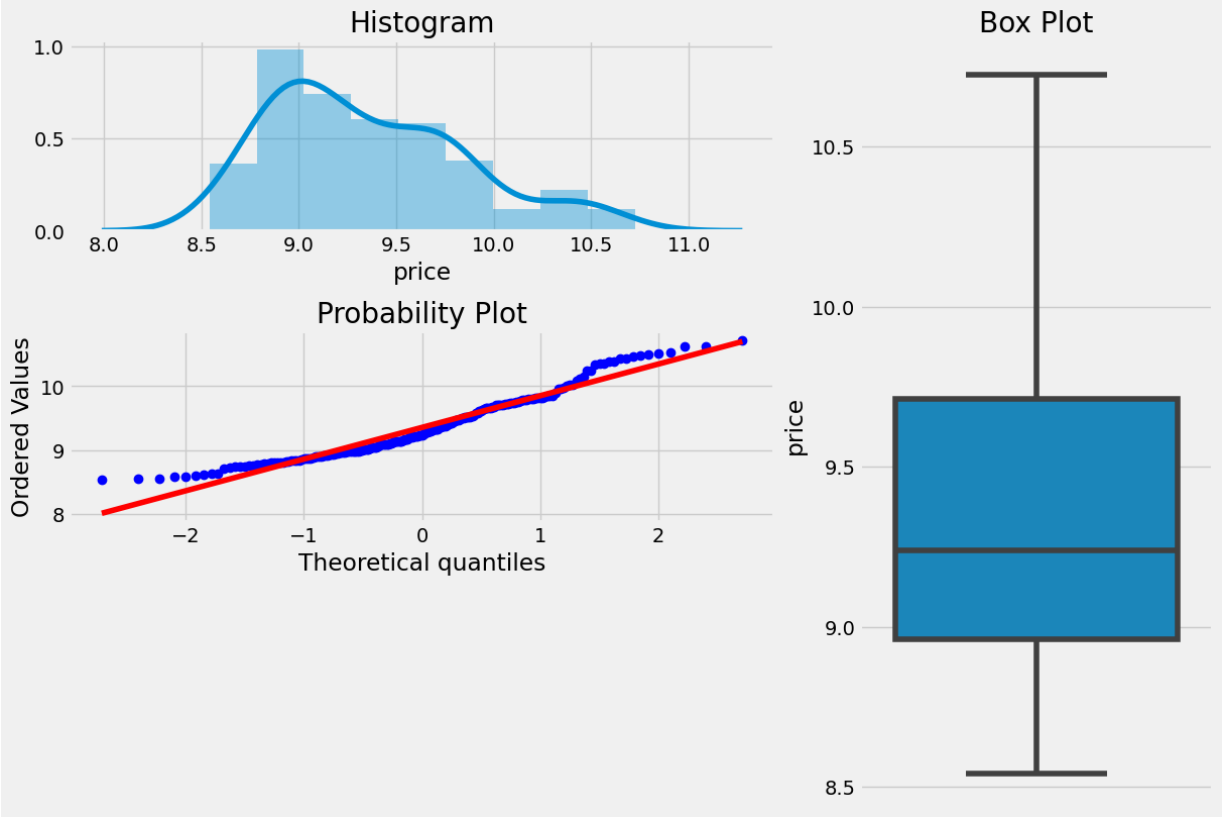
In [56]:
```
normaltest(data.price.values)
```

Out[56]:
```
NormaltestResult(statistic=77.37514013383584, pvalue=1.578344501676757e-17)
```

As we can see, the p-value is very small, so it is not normally distributed.

Now, we can try to transform our data, so it looks more normally distributed. We can use the `np.log()` or `np.log1p` functions from the `numpy` library to perform the log transformation. The `np.log1p` works better with smaller numbers and thus provides more accurate results. This documentation contains more information about the numpy log transform.

In [57]:
```
data['price'] = np.log(data['price'])
plotting_3_chart(data, 'price')
```



Let's check our p-value, after the transformation.

In [58]:
```
normaltest(data.price.values)
```

Out[58]:
```
NormaltestResult(statistic=14.103413457759457, pvalue=0.0008659297880185616)
```

As we can see, the log method transformed the car 'price' distribution into a more symmetrical bell curve. It is still not perfect, but it is much closer to being normally distributed.

There are other ways to correct the skewed data. For example, Square Root Transform (`np.sqrt`) and the Box-Cox Transform (`stats.boxcox` from the `scipy stats` library). To learn more about these two methods, please check out this article.

Exercise 4

Use the `boxcox()` function to do another transformation on the original, untransformed data (previous_data). Use the `normaltest()` function to check for statistics.

In [59]:
```
# Enter your code and run the cell
cp_result = boxcox(previous_data.price)
boxcox_price = cp_result[0]

normaltest(boxcox_price)
```

Out[59]:
```
NormaltestResult(statistic=16.727141721912407, pvalue=0.0002332100843764356)
```

▶ **Solution** (Click Here)

The higher the p-value is, the closer the distribution is to normal. In our case, pvalue=0.0002332100843764356, is very small, (<0.05), so the target variable is still not normally distributed)

4. *Multicollinearity*

Multicollinearity is when there is a strong correlation between the independent variables. Linear regression or multilinear regression requires independent variables to have little or no similar features. *Multicollinearity* can lead to a variety of problems, including:

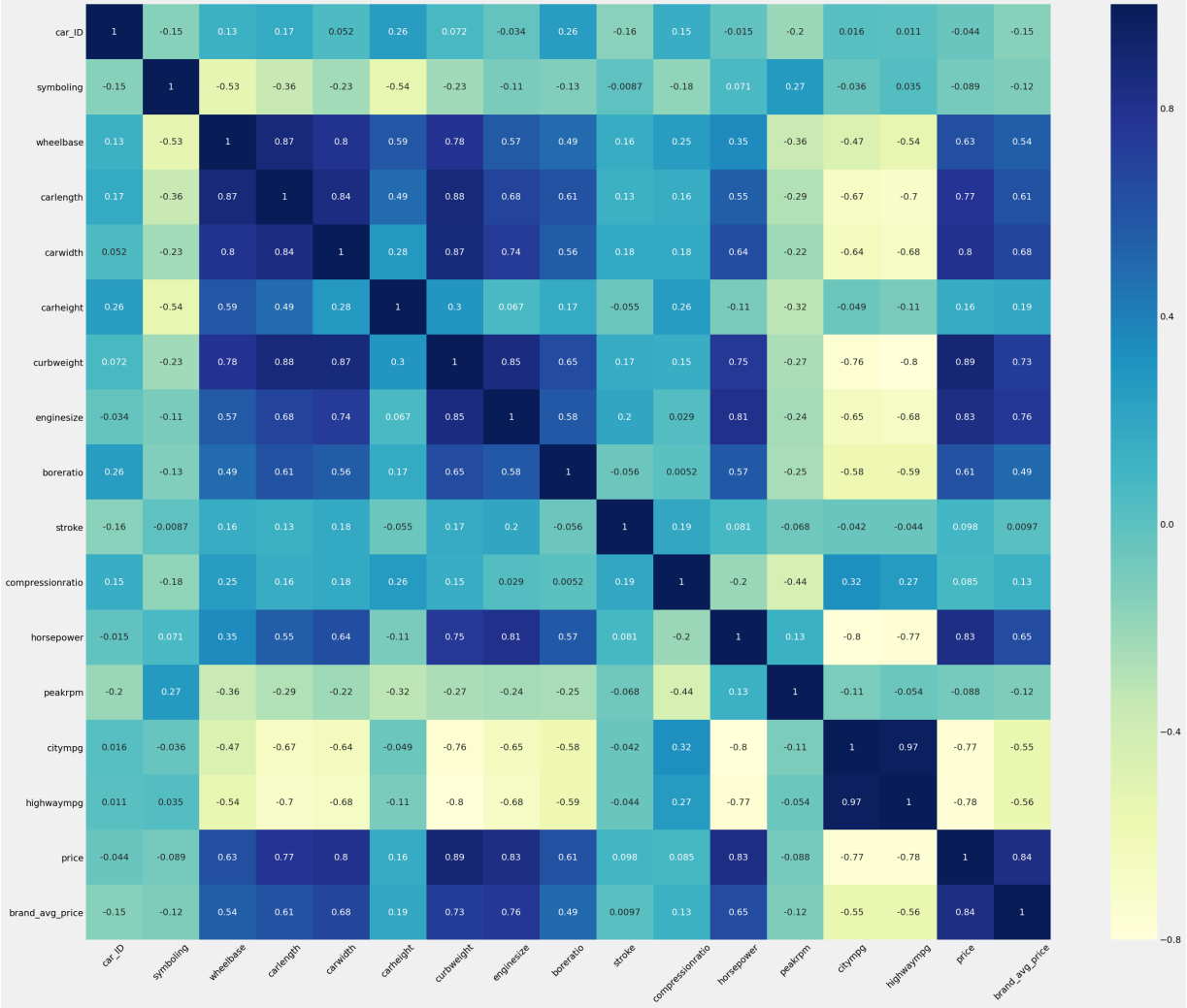
- The effect of predictor variables estimated by our regression will depend on what other variables are included in our model.
- Predictors can have widely different results depending on the observations in our sample, and small changes in samples can result in very different estimated effects.
- With very high multicollinearity, the inverse matrix, the computer calculations may not be accurate.
- We can no longer interpret a coefficient on a variable because there is no scenario in which one variable can change without a conditional change in another variable.

Using `heatmap()` function is an excellent way to identify whether there is *multicollinearity* present or not. The best way to solve for *multicollinearity* is to use the regularization methods like *Ridge* or *Lasso*, which we will introduce in the **Regularization** lab.

Exercise 5

Use the `heatmap()` do display all correlation factors of the numeric variables. Do you see any correlations between the independent features?

In [63]:
```
# Enter your code and run the cell
plt.figure(figsize = (30, 25))
sns.heatmap(data.corr(), annot = True, cmap="YlGnBu")
plt.xticks(rotation=45)
plt.show()
```



▶ **Solution** (Click Here)

Observation As we can see, the multicollinearity still exists in various features. However, we will keep them for now for the sake of learning and let the models (e.x. Regularization models such as Lasso, Ridge in the next lab) do the clean up later on.

Linear Regression Model

List of significant variables after Exploratory Data Analysis :

Numerical:

- Curbweight
- Car Length
- Car width
- Engine Size
- Boreratio
- Horse Power
- Wheel base

- City mpg (miles per gallon)
- Highway mpg (miles per gallon)

Categorical:

- Engine Type
- Fuel type
- Car Body
- Aspiration
- Cylinder Number
- Drivewheel
- Brand Category

We are going to put all the selected features into a data frame.

In [64]:
```
columns=['price', 'fueltype', 'aspiration','carbody', 'drivewheel','wheelbase', 'br
                  'curbweight', 'enginetype', 'cylindernumber', 'enginesize', 'bore



selected = data[columns]
selected.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 205 entries, 0 to 204
Data columns (total 17 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   price           205 non-null    float64
 1   fueltype        205 non-null    object 
 2   aspiration      205 non-null    object 
 3   carbody         205 non-null    object 
 4   drivewheel      205 non-null    object 
 5   wheelbase       205 non-null    float64
 6   brand_category  205 non-null    object 
 7   curbweight      205 non-null    int64  
 8   enginetype      205 non-null    object 
 9   cylindernumber  205 non-null    object 
 10  enginesize      205 non-null    int64  
 11  boreratio       205 non-null    float64
 12  horsepower      205 non-null    int64  
 13  carlength       205 non-null    float64
 14  carwidth        205 non-null    float64
 15  citympg         205 non-null    int64  
 16  highwaympg      205 non-null    int64  
dtypes: float64(5), int64(5), object(7)
memory usage: 36.9+ KB
```

We can find the categorical columns by iterating through the `dtypes` attribute.

In [65]:
```
categorical_columns=[key for key, value in selected.dtypes.iteritems()  if value=='
categorical_columns
```

Out[65]:
```
['fueltype',
 'aspiration',
 'carbody',
 'drivewheel',
 'brand_category',
 'enginetype',
 'cylindernumber']
```

Exercise 6

Find the names of the numeric columns using the list `columns` and assign them to the list `numeric_columns`.

In [66]:
```
# Enter your code and run the cell
numeric_columns=list(set(columns)-set(categorical_columns))
numeric_columns
```

Out[66]:
```
['boreratio',
 'enginesize',
 'carwidth',
 'curbweight',
 'wheelbase',
 'price',
 'citympg',
 'carlength',
 'horsepower',
 'highwaympg']
```

▶ **Solution** (Click Here)

We can split the data into the features `X` and target `y`.

In [67]:
```
X = selected.drop("price", axis=1)
X.head()
```

Out[67]:

	fueltype	aspiration	carbody	drivewheel	wheelbase	brand_category	curbweight	en
0	gas	std	convertible	rwd	88.6	Mid_Range	2548	
1	gas	std	convertible	rwd	88.6	Mid_Range	2548	
2	gas	std	hatchback	rwd	94.5	Mid_Range	2823	
3	gas	std	sedan	fwd	99.8	Mid_Range	2337	
4	gas	std	sedan	4wd	99.4	Mid_Range	2824	

In [68]:
```
y = selected["price"].copy()
y.head()
```

Out[68]:
```
0    9.510075
1    9.711116
2    9.711116
3    9.543235
4    9.767095
Name: price, dtype: float64
```

Before we used one-hot encoding to deal with the categorical data, let's examine the distribution of the categorical variables:

In [69]:
```
for column in  categorical_columns:
    print("column name:", column)
    print("value_count:")
    print( X[column].value_counts())
```

```
column name: fueltype
value_count:
gas       185
diesel     20
Name: fueltype, dtype: int64
column name: aspiration
value_count:
std      168
turbo     37
Name: aspiration, dtype: int64
column name: carbody
value_count:
sedan          96
hatchback      70
wagon          25
hardtop         8
convertible     6
Name: carbody, dtype: int64
column name: drivewheel
value_count:
fwd    120
rwd     76
4wd      9
Name: drivewheel, dtype: int64
column name: brand_category
value_count:
Budget       95
Mid_Range    86
Luxury       24
Name: brand_category, dtype: int64
column name: enginetype
value_count:
ohc      148
ohcf      15
ohcv      13
dohc      12
l         12
rotor      4
dohcv      1
Name: enginetype, dtype: int64
column name: cylindernumber
value_count:
four      159
six        24
five       11
eight       5
two         4
three       1
twelve      1
Name: cylindernumber, dtype: int64
```

We see many categorical features have few or one occurrence. For example, we see `three`, `twelve` only occur once in the column `cylindernumber`. Therefore, if the components for the one-hot encoding are constructed using the training data, and the sample in the column `cylindernumber` does not include three or twelve, we will get an error. As a result,

we must split the data before the transformation. This is fine as one-hot encoding is a deterministic transform, but for other transforms, for example standardization, the parameters should be estimated using the training data, then applied to the test data.

OneHotEncoder

We will use the following modules:

In [70]:
```
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
```

To perform one-hot encoding, we use the `ColumnTransformer` class, this allows different columns or column subsets to be transformed separately.

The input is as follows:

The `transformerslist` is the number of tuples. The list of `(name, transformer, columns)` tuples specify the transformer objects to be applied to the subsets of the data.

- name: name of the operation that can be used later
- `transformer` : estimator must support fit and transform, in this case we will use `OneHotEncoder()`
- `'drop'` : to drop the columns
- `'passthrough'` : to pass them through untransformed data
- `remainder` : specifies the columns that are not transformed are being set to `passthrough` . They are combined in the output, and the non-specified columns are dropped.

We apply `fit_transform()` to transform the data.

In [71]:
```
one_hot = ColumnTransformer(transformers=[("one_hot", OneHotEncoder(), categorical_
X=one_hot.fit_transform(X)
type(X)
```

Out[71]:
```
numpy.ndarray
```

We see the output is a NumPy array, so let's get the feature names from the `one_hot` object using `get_feature_names_out()` method. The output will be the feature name with the prefix of the name of the transformer. For one-hot encoding, the prefix will also include the name of the column that generated that feature.

In [72]:
```
names=one_hot.get_feature_names_out()
names
```

Out[72]:
```
array(['one_hot__fueltype_diesel', 'one_hot__fueltype_gas',
       'one_hot__aspiration_std', 'one_hot__aspiration_turbo',
       'one_hot__carbody_convertible', 'one_hot__carbody_hardtop',
       'one_hot__carbody_hatchback', 'one_hot__carbody_sedan',
       'one_hot__carbody_wagon', 'one_hot__drivewheel_4wd',
       'one_hot__drivewheel_fwd', 'one_hot__drivewheel_rwd',
       'one_hot__brand_category_Budget', 'one_hot__brand_category_Luxury',
       'one_hot__brand_category_Mid_Range', 'one_hot__enginetype_dohc',
       'one_hot__enginetype_dohcv', 'one_hot__enginetype_l',
       'one_hot__enginetype_ohc', 'one_hot__enginetype_ohcf',
       'one_hot__enginetype_ohcv', 'one_hot__enginetype_rotor',
       'one_hot__cylindernumber_eight', 'one_hot__cylindernumber_five',
       'one_hot__cylindernumber_four', 'one_hot__cylindernumber_six',
       'one_hot__cylindernumber_three', 'one_hot__cylindernumber_twelve',
       'one_hot__cylindernumber_two', 'remainder__wheelbase',
       'remainder__curbweight', 'remainder__enginesize',
       'remainder__boreratio', 'remainder__horsepower',
       'remainder__carlength', 'remainder__carwidth',
       'remainder__citympg', 'remainder__highwaympg'], dtype=object)
```

Let's strip out the prefix of the string.

In [73]:
```
colunm_names=[name[name.find("_")+1:] for name in  [name[name.find("__")+2:] for na
colunm_names
```

Out[73]:
```
['diesel',
 'gas',
 'std',
 'turbo',
 'convertible',
 'hardtop',
 'hatchback',
 'sedan',
 'wagon',
 '4wd',
 'fwd',
 'rwd',
 'category_Budget',
 'category_Luxury',
 'category_Mid_Range',
 'dohc',
 'dohcv',
 'l',
 'ohc',
 'ohcf',
 'ohcv',
 'rotor',
 'eight',
 'five',
 'four',
 'six',
 'three',
 'twelve',
 'two',
 'wheelbase',
 'curbweight',
 'enginesize',
 'boreratio',
 'horsepower',
 'carlength',
 'carwidth',
 'citympg',
 'highwaympg']
```

We can save the result as a dataframe to be used in other labs.

In [74]:
```
df=pd.DataFrame(data=X,columns=colunm_names)
#df.to_csv('cleaned_car_data.csv', index=False)
```

Exercise 7

Write the lines of code that performs same task as `ColumnTransformer` using `OneHotEncoder()`.

In [76]:
```
# Enter your code and run the cellX_ = selected[categorical_columns+numeric_columns
X_ = selected[categorical_columns+numeric_columns]
```

```
X_numeric = X_[numeric_columns].to_numpy()
X_categorical = OneHotEncoder().fit_transform(X_[categorical_columns]).toarray()
X_= np.concatenate((X_categorical,X_numeric), axis = 1)
```

▶ **Solution** (Click Here)

Exercise 8

Write the lines of code that performs same task as `ColumnTransformer` using `pd.get_dummies`.

In [79]:
```
# Enter your code and run the cell
def dummies(x, data):
    temp = pd.get_dummies(data[x], drop_first=True)
    data = pd.concat([data, temp], axis=1)
    data.drop([x], axis=1, inplace=True)
    return data

X_ = selected[categorical_columns + numeric_columns]

for column in categorical_columns:
    print(pd.unique(data[column]))

for column in categorical_columns:
    X_ = dummies(column, X_)
```

```
['gas' 'diesel']
['std' 'turbo']
['convertible' 'hatchback' 'sedan' 'wagon' 'hardtop']
['rwd' 'fwd' '4wd']
['Mid_Range' 'Luxury' 'Budget']
['dohc' 'ohcv' 'ohc' 'l' 'rotor' 'ohcf' 'dohcv']
['four' 'six' 'five' 'three' 'twelve' 'two' 'eight']
```

▶ **Solution** (Click Here)

Train Test Split

We split our data into training and testing sets, using 30% of the data for testing.

In [80]:
```
from sklearn.model_selection import train_test_split
```

In [81]:
```
X_train, X_test, y_train, y_test = train_test_split( df, y, test_size=0.30, random_
```

Standardize the Data

We standardize features by removing the mean and scaling to unit variance using `StandardScaler`, we create a `StandardScaler` object:

In [82]:
```
from sklearn.preprocessing import StandardScaler
```

In [83]:
```
ss=StandardScaler()
ss
```

Out[83]:
```
StandardScaler()
```

We `fit` our training data, then we `transform` it.

In [84]:
```
X_train=ss.fit_transform(X_train)
```

Linear Regression

Finally, we apply the `LinearRegression()` model and `fit()` our `X` and `y` data.

In [85]:
```
lm = LinearRegression()
lm.fit(X_train,y_train)
```

Out[85]:
```
LinearRegression()
```

Making Prediction

We will select some random data and apply the `predict()` function.

In [86]:
```
X_test=ss.transform(X_test)
car_price_predictions = lm.predict(X_test)
car_price_predictions
```

Out[86]:
```
array([ 8.87538854e+00,  9.77477195e+00,  9.33322270e+00,  2.64988987e+12,
        9.23757613e+00,  9.35534969e+00,  8.70207306e+00,  8.79050026e+00,
        9.67110567e+00,  8.84205359e+00,  9.85361792e+00, -4.89173764e+12,
        9.46990796e+00,  9.53240118e+00,  8.71816340e+00,  9.29843677e+00,
        9.09823376e+00,  9.58629548e+00,  9.01091540e+00,  8.78957724e+00,
        9.23950808e+00,  9.55350991e+00,  9.12538573e+00,  9.34230922e+00,
        9.81241433e+00,  8.86571687e+00,  8.88265864e+00,  9.57776021e+00,
        8.89161179e+00,  8.86274837e+00,  9.12725758e+00,  9.24418297e+00,
        9.99595666e+00,  9.10570845e+00,  8.87593702e+00,  1.03734306e+01,
        9.43619977e+00,  9.68481023e+00,  8.77785585e+00,  1.03956326e+01,
        8.70874293e+00,  9.40225843e+00,  1.04558394e+01,  9.40141643e+00,
        9.29406051e+00,  8.83627669e+00,  8.82975487e+00,  9.47258948e+00,
        9.23765824e+00,  9.14502188e+00,  9.84524354e+00,  8.88929265e+00,
        8.96307323e+00,  9.15555354e+00,  9.97768726e+00,  9.74791123e+00,
        9.24082018e+00,  9.98489278e+00,  9.20493247e+00,  8.80234906e+00,
        8.38097033e+00,  9.48391382e+00])
```

Model Evaluation

Let's evaluate this model with some statistics. We will use *Scikit_Learn's* `mean_squared_error()` function for this evaluation. MSE measures the average of the

squares of the errors, that is, the average squared difference between the estimated values and the actual values using the test data. For more information on MSE, please visit this wikipedia site.

In [87]:
```
mse = mean_squared_error(y_test, car_price_predictions)
mse
```

Out[87]:
```
4.992098953340692e+23
```

Checking the R squared, the coefficient of determination, which is the proportion of the variation in the dependent variable that is predictable from the independent variables. The closer is R squared to 1, the better is the fit of the model.

The `score()` method returns the coefficient of determination of the prediction.

In [88]:
```
lm.score(X_test,y_test)
```

Out[88]:
```
-1.9399530214719651e+24
```

The `r2_score` method returns the same statistic, also known as the goodness of fit of the model.

In [89]:
```
from sklearn.metrics import r2_score
```

In [90]:
```
r2_score(y_test,car_price_predictions)
```

Out[90]:
```
-1.9399530214719651e+24
```

If the R squared is negative, it suggests the overfitting, when a statistical model fits exactly against its training data.

Pipeline Object

We can also create a Pipeline object and apply a set of transforms sequentially. Then, we can apply linear regression. Data Pipelines simplify the steps of processing the data. We use the module Pipeline to create a pipeline. We also use `StandardScaler` as a step in our pipeline.

We create the pipeline, by creating a list of tuples including the name of the model or estimator and its corresponding constructor.

In [91]:
```
steps=[('scaler', StandardScaler()), ('lm',  LinearRegression())]
```

We input the list as an argument to the pipeline constructor.

In [92]:
```
pipe = Pipeline(steps=steps)
```

We `fit` the constructor.

In [93]:
```
pipe.fit(X_train,y_train)
```

Out[93]:
```
Pipeline(steps=[('scaler', StandardScaler()), ('lm', LinearRegression())])
```

We make a prediction and perform model evaluation.

In [94]:
```
car_price_predictions = pipe.predict(X_test)
mse = mean_squared_error(y_test, car_price_predictions)
rmse = np.sqrt(mse)
rmse
```

Out[94]:
```
3611239073507.6885
```

In [95]:
```
r2_score(car_price_predictions, y_test)
```

Out[95]:
```
-0.0047288164225374185
```

Exercise 9

Use the `ColumnTransformer` in the pipeline, then train the model using **all** the data, make a prediction and calculate all the metrics.

In [96]:
```
# Enter your code and run the cell
X = selected[categorical_columns+numeric_columns]
one_hot = ColumnTransformer(transformers=[("one_hot", OneHotEncoder(), categorical_
steps=[('one_hot',one_hot), ('scaler', StandardScaler()), ('lm',  LinearRegression(

pipe = Pipeline(steps=steps)
pipe.fit(X,y)
car_price_predictions=pipe.predict(X)
r2_score(car_price_predictions, y)
```

Out[96]:
```
1.0
```