Car Price Prediction Model

Based on Real Life Problem of car sales, Models predicts the car price on the basis of different inputs such as brand, engine volume etc. It involves the following

- Model: Multivariable Linear Regression
- Data Cleaning
- Data visualisation
- Taking Assumptions
- Log Transformation
- Creation of dummies
- Creation of a model

Project By:

Chaman Patel
21121017

Background:

The automotive industry is highly competitive, with car prices being influenced by a multitude of factors. Accurate car price predictions can significantly benefit various stakeholders, including manufacturers, dealers, and buyers. By leveraging historical data, we aim to build a predictive model that can estimate the price of a car based on key attributes.

Objective:

The objective of this project is to develop a data science model that can predict car prices using the provided dataset. The model will help in understanding the relationship between various features and the car prices, ultimately aiding in making informed pricing decisions.

Problem Statement:

Create a predictive model to estimate the price of a car based on the provided dataset. The dataset includes various features that potentially influence car prices.

Data Exploration:

Data Preview

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	Year	Model
0	BMW	4200.0	sedan	277	2.0	Petrol	yes	1991	320
1	Mercedes-Benz	7900.0	van	427	2.9	Diesel	yes	1999	Sprinter 212
2	Mercedes-Benz	13300.0	sedan	358	5.0	Gas	yes	2003	S 500
3	Audi	23000.0	crossover	240	4.2	Petrol	yes	2007	Q7
4	Toyota	18300.0	crossover	120	2.0	Petrol	yes	2011	Rav 4

Code:

```
raw_data = pd.read_csv('1.04. Real-life example.csv')
raw_data.head()
```

Description of the Dataset

The provided dataset contains information about various cars and their attributes, which can be used to predict car prices. Here's a detailed description of the dataset:

Dataset Overview

- **Total Entries:** 4345
- **Total Columns:** 9

Columns and Data Types

1. **Brand (object):** The manufacturer of the car (e.g., BMW, Mercedes-Benz, Audi).
2. **Price (float64):** The price of the car. Note that there are some missing values (4173 non-null entries).
3. **Body (object):** The body type of the car (e.g., sedan, van, crossover).
4. **Mileage (int64):** The total distance the car has been driven, in kilometers.
5. **EngineV (float64):** The volume of the car's engine in liters. There are some missing values (4195 non-null entries).
6. **Engine Type (object):** The type of engine (e.g., Petrol, Diesel, Gas).
7. **Registration (object):** Whether the car is registered or not (yes or no).
8. **Year (int64):** The year the car was manufactured.
9. **Model (object):** The specific model of the car.

Data Summary

- **Brand:** This categorical feature indicates the car manufacturer. Examples include BMW, Mercedes-Benz, and Audi.
- **Price:** This is the target variable representing the car's price. Some values are missing and need to be addressed.
- **Body:** This categorical feature indicates the body type of the car. Examples include sedan, van, and crossover.
- **Mileage:** This numeric feature represents the car's mileage in kilometers.
- **EngineV:** This numeric feature represents the engine volume in liters. Some values are missing and need to be addressed.
- **Engine Type:** This categorical feature indicates the type of engine, such as Petrol, Diesel, or Gas.
- **Registration:** This binary categorical feature indicates whether the car is registered (yes or no).
- **Year:** This numeric feature represents the manufacturing year of the car.
- **Model:** This categorical feature represents the specific model of the car.

Python libraries and their purpose in project

NumPy: Numerical Operations such as for calculating log and exponential etc.

Pandas: Dataset retrieval, creation and manipulation

Statsmodel: Check for multicollinearity by variance inflation factor (VIF)

Matplotlib: Outliers detection and data Visualization

SkLearn: Model Creation (Multivariable Linear Regression)

Seaborn: Betterment of the visualized data

Importing the relevant libraries

code:

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
import seaborn as sns
sns.set()
```

Loading Data

```
raw_data = pd.read_csv('1.04. Real-life example.csv')
raw_data.head()
```

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	Year	Model
0	BMW	4200.0	sedan	277	2.0	Petrol	yes	1991	320
1	Mercedes-Benz	7900.0	van	427	2.9	Diesel	yes	1999	Sprinter 212
2	Mercedes-Benz	13300.0	sedan	358	5.0	Gas	yes	2003	S 500
3	Audi	23000.0	crossover	240	4.2	Petrol	yes	2007	Q7
4	Toyota	18300.0	crossover	120	2.0	Petrol	yes	2011	Rav 4

Preprocessing

Exploration of Descriptive Statistics of the variable

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	Year	Model
count	4345	4173.000000	4345	4345.000000	4195.000000	4345	4345	4345.000000	4345
unique	7	NaN	6	NaN	NaN	4	2	NaN	312
top	Volkswagen	NaN	sedan	NaN	NaN	Diesel	yes	NaN	E-Class
freq	936	NaN	1649	NaN	NaN	2019	3947	NaN	199
mean	NaN	19418.746935	NaN	161.237284	2.790734	NaN	NaN	2006.550058	NaN
std	NaN	25584.242620	NaN	105.705797	5.066437	NaN	NaN	6.719097	NaN
min	NaN	600.000000	NaN	0.000000	0.600000	NaN	NaN	1969.000000	NaN
25%	NaN	6999.000000	NaN	86.000000	1.800000	NaN	NaN	2003.000000	NaN
50%	NaN	11500.000000	NaN	155.000000	2.200000	NaN	NaN	2008.000000	NaN
75%	NaN	21700.000000	NaN	230.000000	3.000000	NaN	NaN	2012.000000	NaN
max	NaN	300000.000000	NaN	980.000000	99.990000	NaN	NaN	2016.000000	NaN

Code: `raw_data.describe(include='all')`

Note: Categorical variables don't have some types of numerical descriptives and numerical variables

Outcomes on analysing descriptive statistics

- 4345 number of rows in model column may have lots of unique model require lots of input columns for dummy. I decided to drop it as it will not impact my model much.

- There is missing values in Price and EngineV column.

  I will be dealing with it by dropping the rows with missing values. Since, the rule of thumb says that dropping nearly 5% of data will effect none and it’s completely ok.

Data Cleaning

Determining the variable of interest

I have dropped column model because it require large number of dummy variable for each model there will be 1 column.

Code:

```
data = raw_data.drop(['Model'],axis=1)
data.describe(include='all')
```

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	Year
count	4345	4173.000000	4345	4345.000000	4195.000000	4345	4345	4345.000000
unique	7	NaN	6	NaN	NaN	4	2	NaN
top	Volkswagen	NaN	sedan	NaN	NaN	Diesel	yes	NaN

Dealing with missing values

data.isnull() shows a df with the information whether a data point is null. Since True = the data point is missing, while False = the data point is not missing, we can sum them. This will give us the total number of missing values feature-wise

Code: `data.isnull().sum()`

```
Brand             0
Price           172
Body              0
Mileage           0
EngineV         150
Engine Type       0
Registration      0
Year              0
dtype: int64
```

Descriptive without missing values

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	Year
count	4025	4025.000000	4025	4025.000000	4025.000000	4025	4025	4025.000000
unique	7	NaN	6	NaN	NaN	4	2	NaN
top	Volkswagen	NaN	sedan	NaN	NaN	Diesel	yes	NaN
freq	880	NaN	1534	NaN	NaN	1861	3654	NaN
mean	NaN	19552.308065	NaN	163.572174	2.764586	NaN	NaN	2006.379627
std	NaN	25815.734988	NaN	103.394703	4.935941	NaN	NaN	6.695595
min	NaN	600.000000	NaN	0.000000	0.600000	NaN	NaN	1969.000000
25%	NaN	6999.000000	NaN	90.000000	1.800000	NaN	NaN	2003.000000
50%	NaN	11500.000000	NaN	158.000000	2.200000	NaN	NaN	2007.000000
75%	NaN	21900.000000	NaN	230.000000	3.000000	NaN	NaN	2012.000000
max	NaN	300000.000000	NaN	980.000000	99.990000	NaN	NaN	2016.000000

Plotting Probability Distribution Function (PDF)

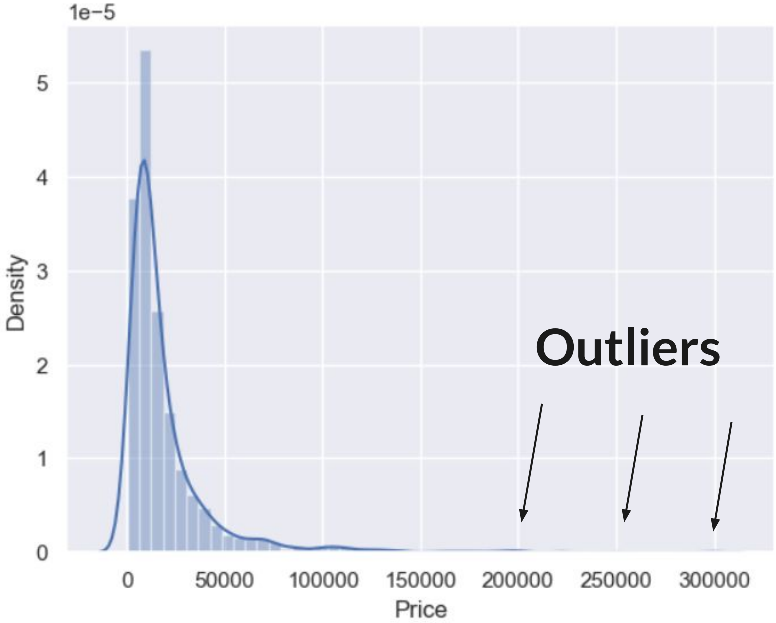
Purpose:

- The PDF will show us how that variable is distributed.
- This makes it very easy to spot anomalies, such as outliers.
- The PDF is often the basis on which we decide whether we want to transform a feature.

Code:

```
sns.distplot(data_no_mv['Price'])
```

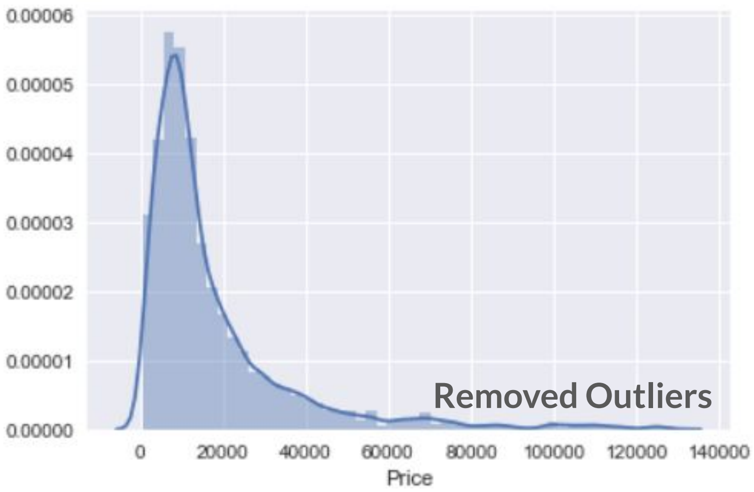
- Here, the outliers are situated around the higher prices (right side of the graph)
- Logic should also be applied
- This is a dataset about used cars, therefore one can imagine how $300,000 is an excessive price



Dealing with outliers

- We will only take values of price below 99 percentile.
- Quantile method of Pandas will help to deal get value of 99 percentile.

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	Year
count	3984	3984.000000	3984	3984.000000	3984.000000	3984	3984	3984.000000
unique	7	NaN	6	NaN	NaN	4	2	NaN
top	Volkswagen	NaN	sedan	NaN	NaN	Diesel	yes	NaN
freq	880	NaN	1528	NaN	NaN	1853	3613	NaN
mean	NaN	17837.117460	NaN	165.116466	2.743770	NaN	NaN	2006.292922
std	NaN	18976.268315	NaN	102.766126	4.956057	NaN	NaN	6.672745
min	NaN	600.000000	NaN	0.000000	0.600000	NaN	NaN	1969.000000
25%	NaN	6980.000000	NaN	93.000000	1.800000	NaN	NaN	2002.750000
50%	NaN	11400.000000	NaN	160.000000	2.200000	NaN	NaN	2007.000000
75%	NaN	21000.000000	NaN	230.000000	3.000000	NaN	NaN	2011.000000
max	NaN	129222.000000	NaN	980.000000	99.990000	NaN	NaN	2016.000000



Code:

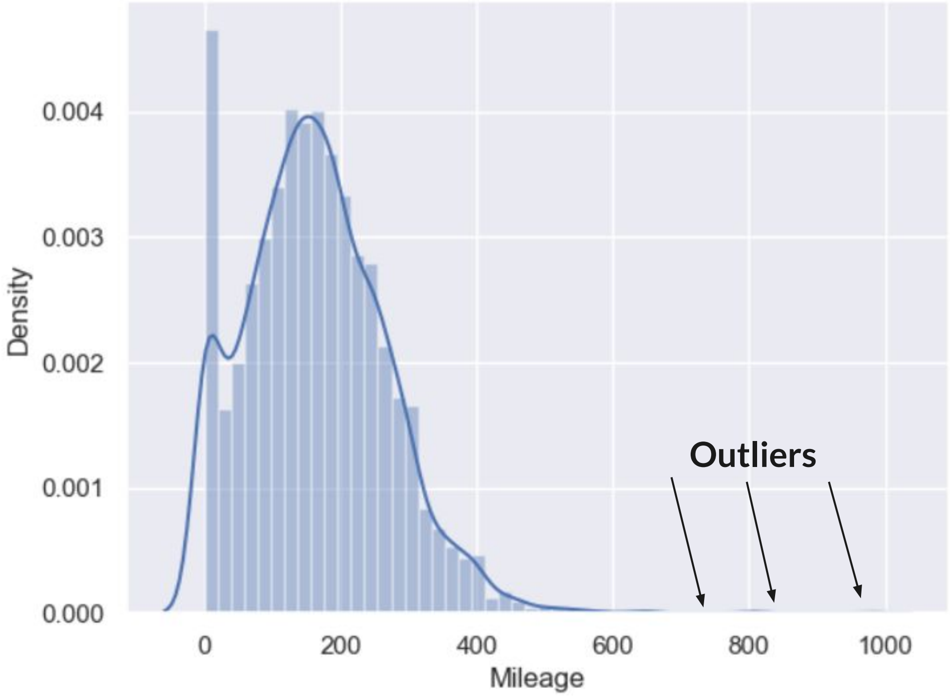
```
q = data_no_mv['Price'].quantile(0.99)
data_1 = data_no_mv[data_no_mv['Price']<q]
data_1.describe(include='all')
```

Code: `sns.distplot(data_1['Price'])`

Checking for other variable

Mileage

Code: `sns.distplot(data_no_mv['Mileage'])`



This Outliers will affect the variable weight. Since, we want to remove this outliers because we want our model for majority of the population. Therefore, we want maximum occurrences.

Dealing with it

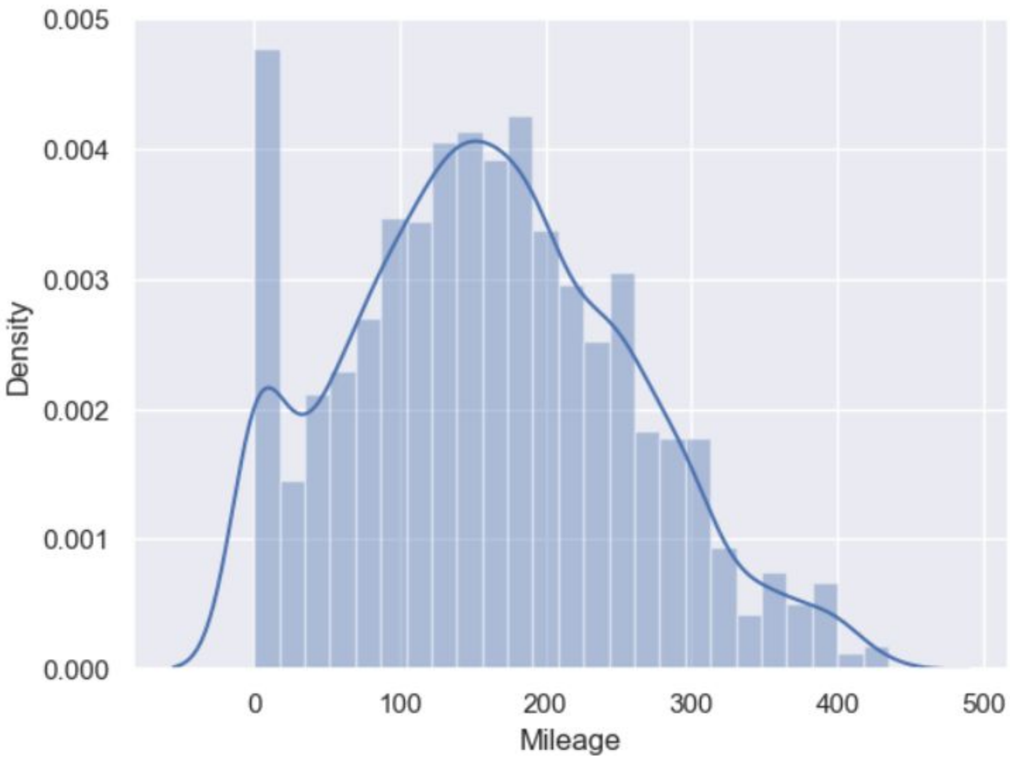
Taking 99 Percentile of the mileage data

Code:

```
q = data_1['Mileage'].quantile(0.99)

data_2 = data_1[data_1['Mileage']<q]

sns.distplot(data_2['Mileage'])
```

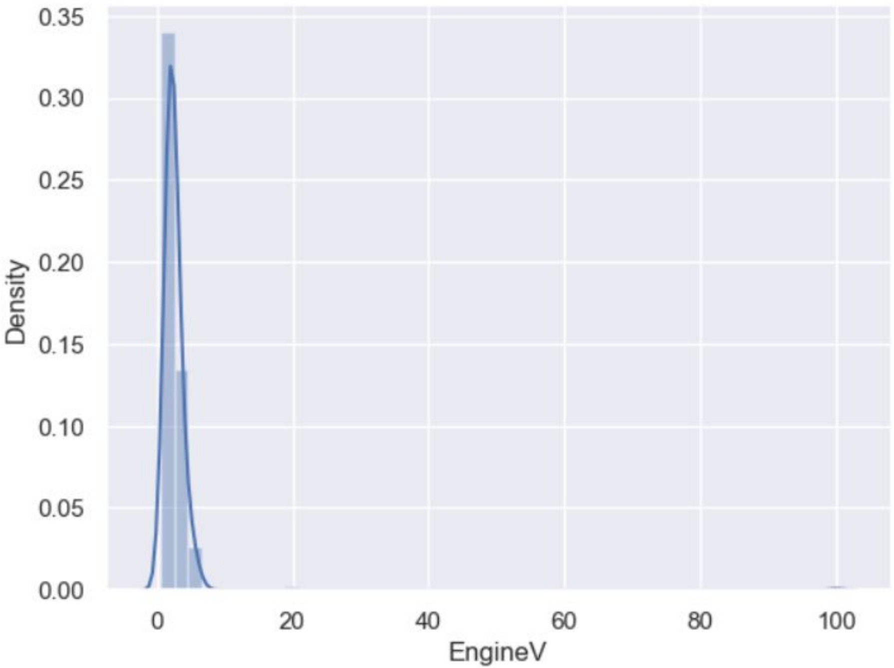


Engine Volume

Code: sns.distplot(data_no_mv['EngineV'])

OBSERVATION

- The situation with engine volume is very strange.
- In such cases it makes sense to manually check what may be causing the problem.
- In our case the issue comes from the fact that most missing values are indicated with 99.99 or 99.
- There are also some incorrect entries like 75.



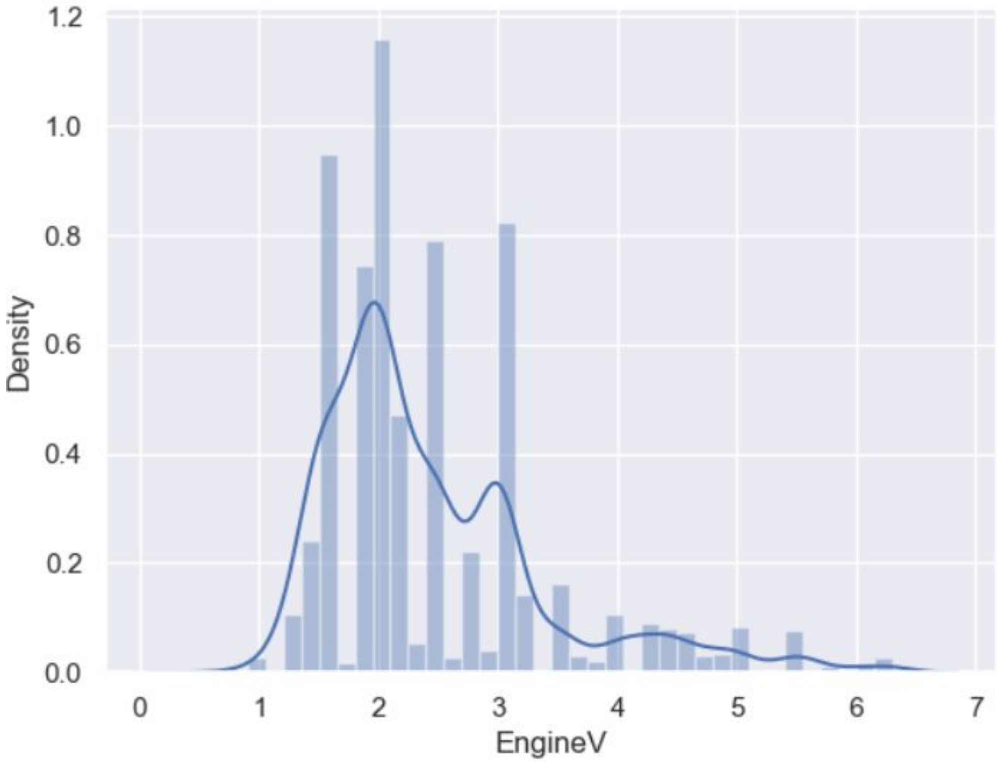
Dealing With incorrect entries in EngineV

Fact

- A simple Google search can indicate the natural domain of this variable
- Car engine volumes are usually (always?) below 6.5l
- This is a prime example of the fact that a domain expert (a person working in the car industry)
- may find it much easier to determine problems with the data than an outsider

Code:

```
data_3 = data_2[data_2['EngineV']<6.5]
sns.distplot(data_3['EngineV'])
```

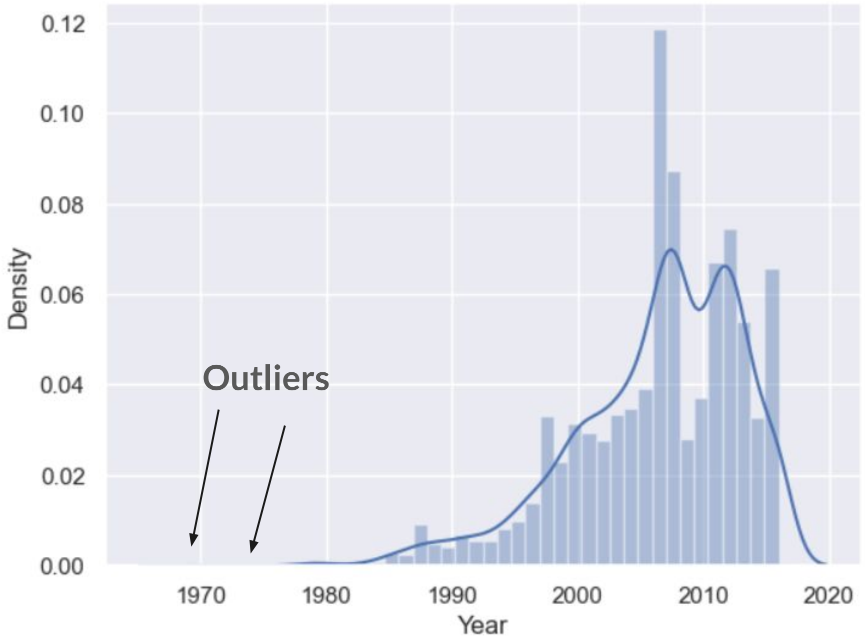


Year

Code: `sns.distplot(data_no_mv['Year'])`

Observation

- the situation with 'Year' is similar to 'Price' and 'Mileage'.
- However, the outliers are on the low end.

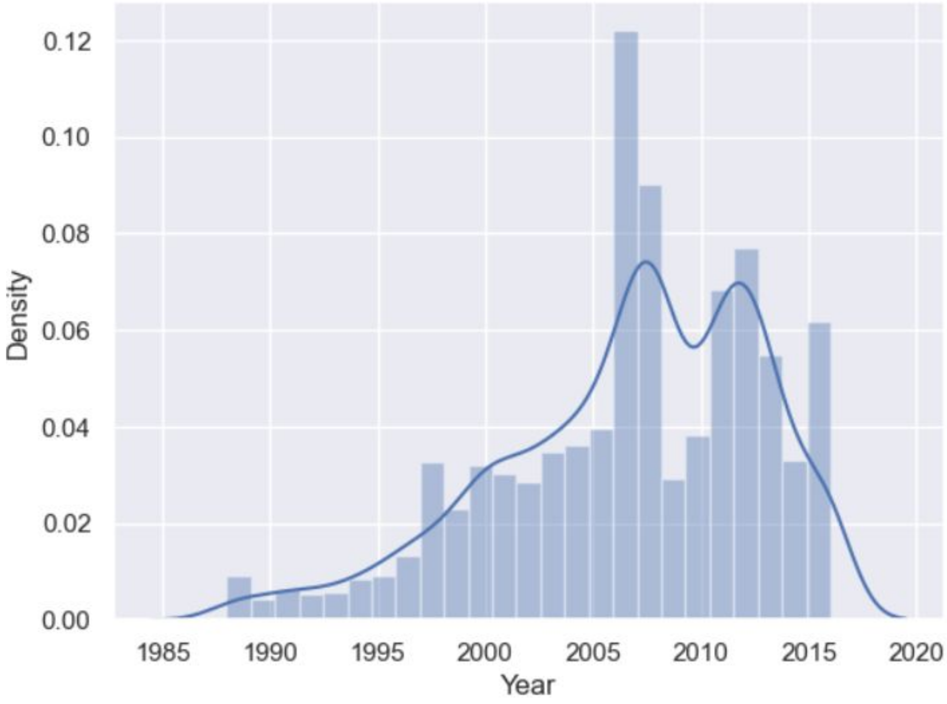


Dealing with year

Code:

```
q = data_3['Year'].quantile(0.01)

data_4 = data_3[data_3['Year']>q]
```



Descriptive of Cleaned Data

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	Year
count	3867	3867.000000	3867	3867.000000	3867.000000	3867	3867	3867.000000
unique	7	NaN	6	NaN	NaN	4	2	NaN
top	Volkswagen	NaN	sedan	NaN	NaN	Diesel	yes	NaN
freq	848	NaN	1467	NaN	NaN	1807	3505	NaN
mean	NaN	18194.455679	NaN	160.542539	2.450440	NaN	NaN	2006.709853
std	NaN	19085.855165	NaN	95.633291	0.949366	NaN	NaN	6.103870
min	NaN	800.000000	NaN	0.000000	0.600000	NaN	NaN	1988.000000
25%	NaN	7200.000000	NaN	91.000000	1.800000	NaN	NaN	2003.000000
50%	NaN	11700.000000	NaN	157.000000	2.200000	NaN	NaN	2008.000000
75%	NaN	21700.000000	NaN	225.000000	3.000000	NaN	NaN	2012.000000
max	NaN	129222.000000	NaN	435.000000	6.300000	NaN	NaN	2016.000000

Code: `data_cleaned.describe(include='all')`

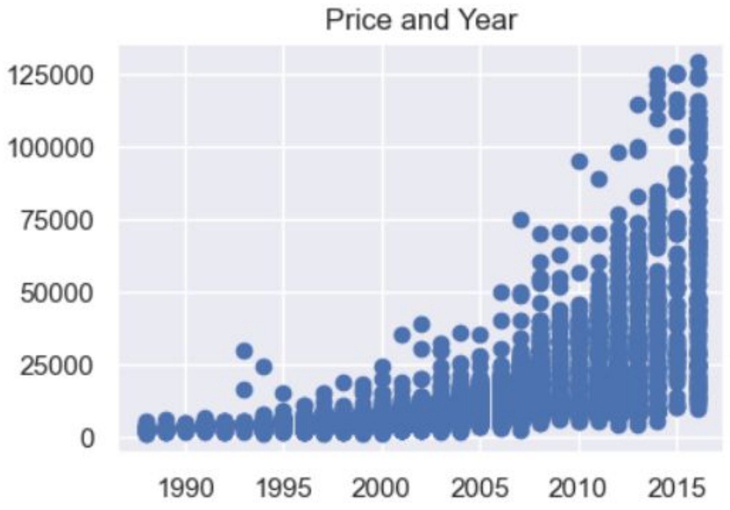
Checking OLS Assumption

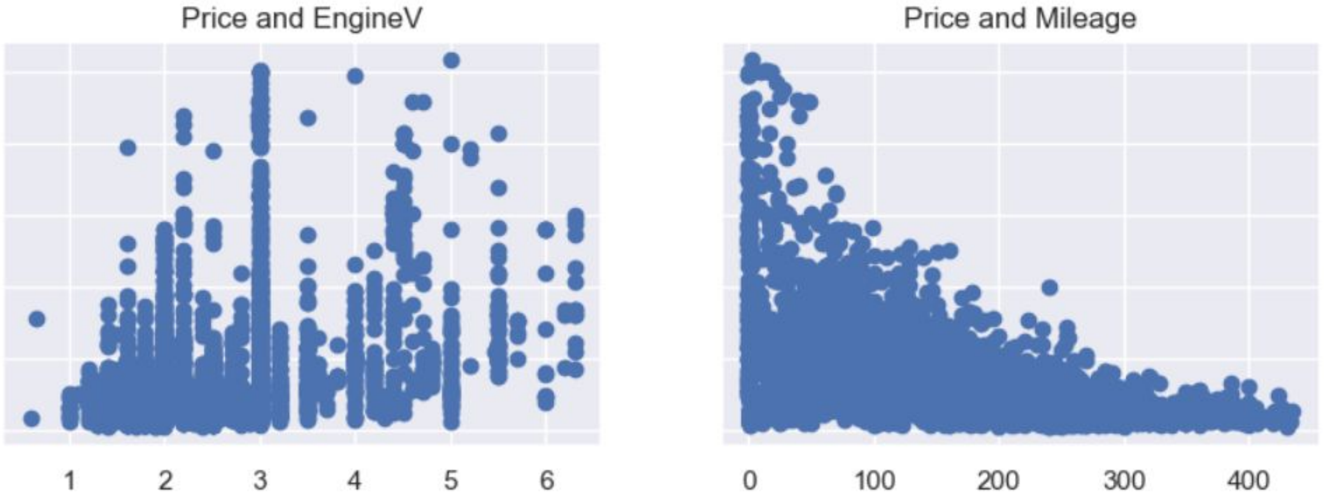
Scatter plot can help

Code:

```
f, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=True, figsize =(15,3))
#sharey -> share 'Price' as y
ax1.scatter(data_cleaned['Year'],data_cleaned['Price'])
ax1.set_title('Price and Year')
ax2.scatter(data_cleaned['EngineV'],data_cleaned['Price'])
ax2.set_title('Price and EngineV')
ax3.scatter(data_cleaned['Mileage'],data_cleaned['Price'])
ax3.set_title('Price and Mileage')


plt.show()
```





Observation:

- Year and Mileage are exponentially dependent on Price.
- Since, we need linear distribution we will take **log Transformation** of Price.

Linear regression requires linear dependence of variables

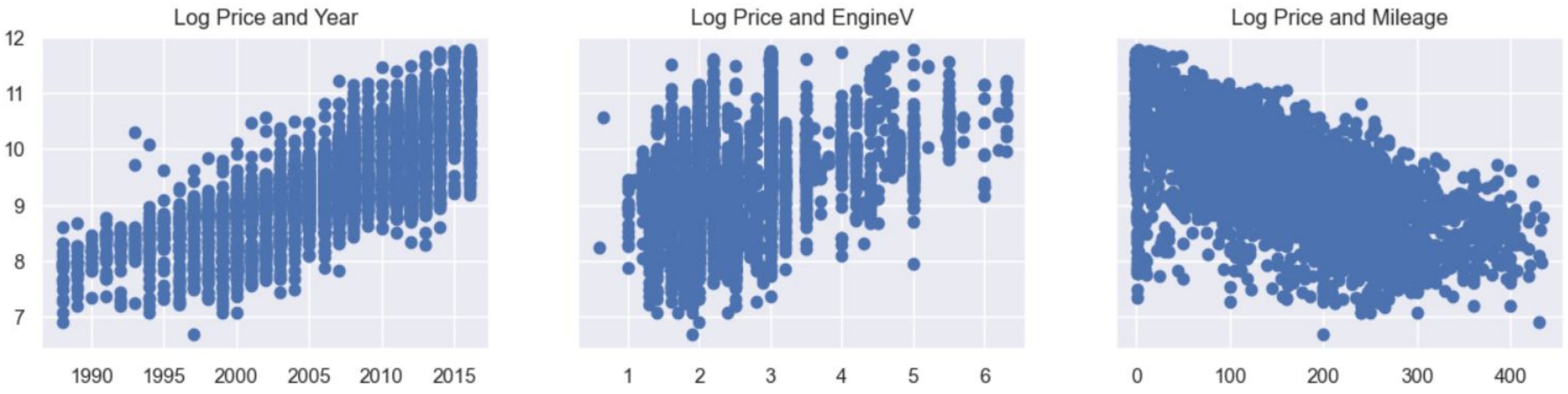
Transformations

Code:

```
log_price = np.log(data_cleaned['Price'])
data_cleaned['log_price'] = log_price
data_cleaned
```

	Brand	Price	Body	Mileage	EngineV	Engine Type	Registration	Year	log_price
0	BMW	4200.0	sedan	277	2.0	Petrol	yes	1991	8.342840
1	Mercedes-Benz	7900.0	van	427	2.9	Diesel	yes	1999	8.974618
2	Mercedes-Benz	13300.0	sedan	358	5.0	Gas	yes	2003	9.495519
3	Audi	23000.0	crossover	240	4.2	Petrol	yes	2007	10.043249
4	Toyota	18300.0	crossover	120	2.0	Petrol	yes	2011	9.814656
...	...	...	...	...	...	...	...	...	...
3862	Volkswagen	11500.0	van	163	2.5	Diesel	yes	2008	9.350102
3863	Toyota	17900.0	sedan	35	1.6	Petrol	yes	2014	9.792556
3864	Mercedes-Benz	125000.0	sedan	9	3.0	Diesel	yes	2014	11.736069
3865	BMW	6500.0	sedan	1	3.5	Petrol	yes	1999	8.779557
3866	Volkswagen	13500.0	van	124	2.0	Diesel	yes	2013	9.510445

Scatter plot after Transformation



- The relationships show a clear linear relationship.
- This is some good linear regression material.

Code:

```
f, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=True, figsize =(15,3))
ax1.scatter(data_cleaned['Year'],data_cleaned['log_price'])
ax1.set_title('Log Price and Year')
ax2.scatter(data_cleaned['EngineV'],data_cleaned['log_price'])
ax2.set_title('Log Price and EngineV')
ax3.scatter(data_cleaned['Mileage'],data_cleaned['log_price'])
ax3.set_title('Log Price and Mileage')
```

Multicollinearity

- To make this as easy as possible to use, we declare a variable where we put.
- All features where we want to check for multicollinearity.
- Since our categorical data is not yet preprocessed, we will only take the numerical ones.
- We create a new data frame which will include all the VIFs.
- Note that each variable has its own variance inflation factor as this measure is variable specific (not model specific).
- Here we make use of the variance_inflation_factor, which will basically output the respective VIFs.
- Finally, I like to include names so it is easier to explore the result.
- I will use stats model library to calculate VIF.

Code:

```
from statsmodels.stats.outliers_influence import variance_inflation_factor
variables = data_cleaned[['Mileage','Year','EngineV']]
vif = pd.DataFrame()
vif["VIF"] = [variance_inflation_factor(variables.values, i) for i in range(variables.shape[1])]
vif["Features"] = variables.columns
vif
```

	VIF	Features
0	3.791584	Mileage
1	10.354854	Year
2	7.662068	EngineV

Dealing with multicollinearity

- Since Year has the highest VIF, I will remove it from the model.
- This will drive the VIF of other variables down.
- So even if EngineV seems with a high VIF, too, once 'Year' is gone that will no longer be the case.

Code:

```
data_no_multicollinearity = data_cleaned.drop(['Year'],axis=1)
```

Create Dummy Variable

- To include the categorical data in the regression, let's create dummies.
- It is extremely important that we drop one of the dummies, alternatively we will introduce multicollinearity.

Code:

```
data_with_dummies = pd.get_dummies(data_no_multicollinearity, drop_first=True)
data_with_dummies.head()
```

	Mileage	EngineV	log_price	Brand_BMW	Brand_Mercedes-Benz	Brand_Mitsubishi	Brand_Renault	Brand_Toyota	Brand_Volkswagen	Body_hatch	Body_other	Body_sedan	Body_vagon	Body_van	Engine Type_Gas	Engine Type_Other	Engine Type_Petrol	Registration_yes
0	277	2.0	8.342840	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1
1	427	2.9	8.974618	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1
2	358	5.0	9.495519	0	1	0	0	0	0	0	0	1	0	0	1	0	0	1
3	240	4.2	10.043249	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
4	120	2.0	9.814656	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1

Linear Regression Model

Declare inputs and target

- The target(s) (dependent variable) is 'log price'.
- The inputs are everything BUT the dependent variable, so we can simply drop it.

Code:

```
targets = data_preprocessed['log_price']
inputs = data_preprocessed.drop(['log_price'],axis=1)
```

Scaling of Data

Code:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(inputs)
inputs_scaled = scaler.transform(inputs)
```

Train Test Split

Split the variables with an 80-20 split and some random state.

Code:

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(inputs_scaled, targets, test_size=0.2, random_state=365)
```

Create the Regression

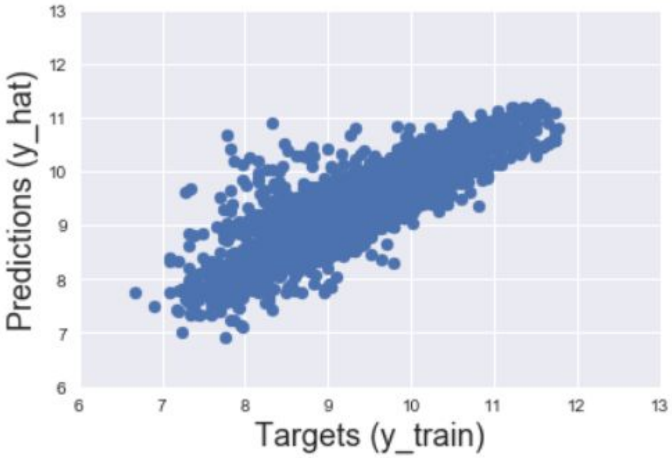
Code:

```
reg = LinearRegression()
reg.fit(x_train,y_train)
y_hat = reg.predict(x_train)
```

- The simplest way to compare the targets (y_train) and the predictions (y_hat) is to plot them on a scatter plot.
- The closer the points to the 45-degree line, the better the prediction.
- Let's also name the axes.
- Sometimes the plot will have different scales of the x-axis and the y-axis.
- This is an issue as we won't be able to interpret the '45-degree line'.
- We want the x-axis and the y-axis to be the same.

Code:

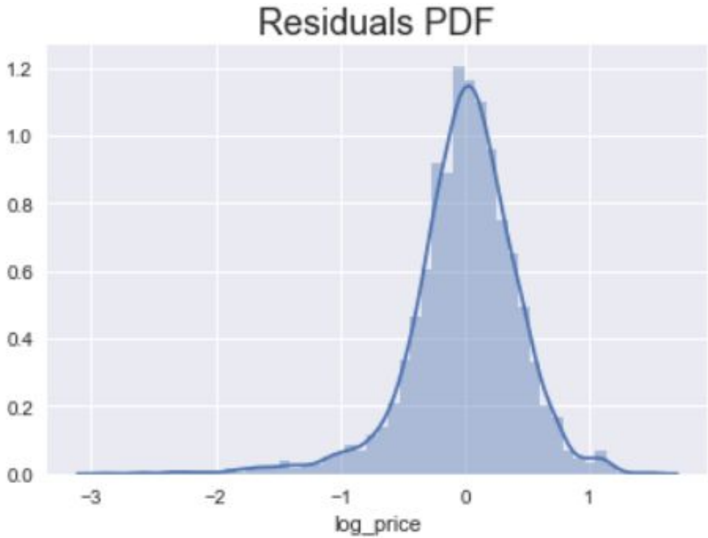
```
plt.scatter(y_train, y_hat)
plt.xlabel('Targets (y_train)',size=18)
plt.ylabel('Predictions (y_hat)',size=18)
plt.xlim(6,13)
plt.ylim(6,13)
plt.show()
```



- Another useful check of our model is a residual plot.
- We can plot the PDF of the residuals and check for anomalies.
- In the best case scenario this plot should be normally distributed.
- In our case we notice that there are many negative residuals (far away from the mean).
- Given the definition of the residuals (y_train - y_hat), negative values imply, that y_hat (predictions) are much higher than y_train (the targets).
- This is food for thought to improve our model.

Code:

```
sns.distplot(y_train - y_hat)
plt.title("Residuals PDF", size=18)
```



Getting R^2 (Score)

Code: `reg.score(x_train,y_train)`

Output: `0.744996578792662`

Finding Weights

Code:

```
reg_summary = pd.DataFrame(inputs.columns.values, columns=['Features'])
reg_summary['Weights'] = reg.coef_
reg_summary
```

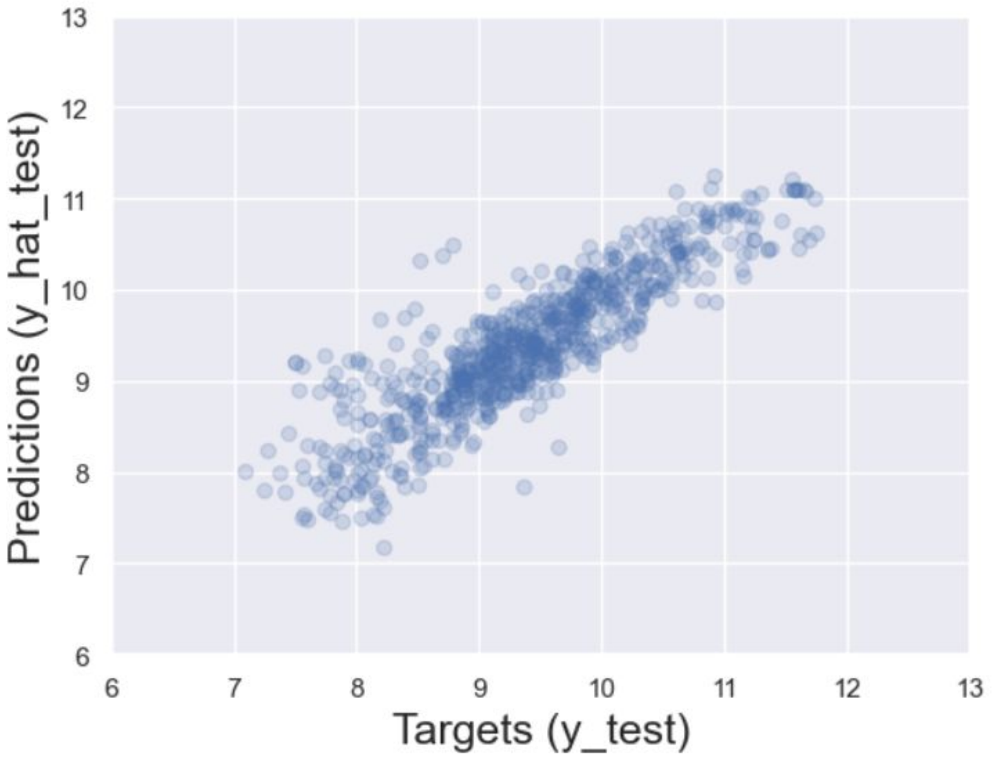
	Features	Weights
0	Mileage	-0.448713
1	EngineV	0.209035
2	Brand_BMW	0.014250
3	Brand_Mercedes-Benz	0.012882
4	Brand_Mitsubishi	-0.140552
5	Brand_Renault	-0.179909
6	Brand_Toyota	-0.060550
7	Brand_Volkswagen	-0.089924
8	Body_hatch	-0.145469
9	Body_other	-0.101444
10	Body_sedan	-0.200630
11	Body_vagon	-0.129887
12	Body_van	-0.168597
13	Engine Type_Gas	-0.121490
14	Engine Type_Other	-0.033368
15	Engine Type_Petrol	-0.146909
16	Registration_yes	0.320473

Testing

- Once we have trained and fine-tuned our model, we can proceed to testing it.
- Testing is done on a dataset that the algorithm has never seen.
- Luckily we have prepared such a dataset.
- Our test inputs are 'x_test', while the outputs: 'y_test'.
- We SHOULD NOT TRAIN THE MODEL ON THEM, we just feed them and find the predictions.
- If the predictions are far off, we will know that our model overfitted.
- Create a scatter plot with the test targets and the test predictions.
- You can include the argument 'alpha' which will introduce opacity to the graph.

Code:

```
y_hat_test = reg.predict(x_test)
plt.scatter(y_test, y_hat_test, alpha=0.2)
plt.xlabel('Targets (y_test)',size=18)
plt.ylabel('Predictions (y_hat_test)',size=18)
plt.xlim(6,13)
plt.ylim(6,13)
plt.show()
```



- Finally, let's manually check these predictions
- To obtain the actual prices, we take the exponential of the log_price
- We can also include the test targets in that data frame (so we can manually compare them)

Code:

```
df_pf['Target'] = np.exp(y_test)
df_pf
```

	Prediction	Target
0	10685.501696	NaN
1	3499.255242	7900.0
2	7553.285218	NaN
3	7463.963017	NaN
4	11353.490075	NaN
...	...	...
769	29651.726363	6950.0
770	10732.071179	NaN
771	13922.446953	NaN
772	27487.751303	NaN
773	13491.163043	NaN

- ❖ Note that we have a lot of missing values.
- ❖ There is no reason to have ANY missing values, though.
- ❖ This suggests that something is wrong with the data frame / indexing.

- After displaying y_test, we find what the issue is
- The old indexes are preserved (recall earlier in that code we made a note on that)
- The code was: data_cleaned = data_4.reset_index(drop=True)
- Therefore, to get a proper result, we must reset the index and drop the old indexing
- Check the result

Code:

```
y_test = y_test.reset_index(drop=True)
y_test.head()
```

```
0    7.740664
1    7.937375
2    7.824046
3    8.764053
4    9.121509
Name: log_price, dtype: float64
```

- Let's overwrite the 'Target' column with the appropriate values
- Again, we need the exponential of the test log price

Code:

```
df_pf['Target'] = np.exp(y_test)

df_pf
```

	Prediction	Target
0	10685.501696	2300.0
1	3499.255242	2800.0
2	7553.285218	2500.0
3	7463.963017	6400.0
4	11353.490075	9150.0
...	...	...
769	29651.726363	29500.0
770	10732.071179	9600.0
771	13922.446953	18300.0
772	27487.751303	68500.0
773	13491.163043	10800.0

- Additionally, we can calculate the difference between the targets and the predictions.
- Note that this is actually the residual (we already plotted the residuals).
- Since OLS is basically an algorithm which minimizes the total sum of squared errors (residuals).
- This comparison makes a lot of sense.

Code:

```
df_pf['Residual'] = df_pf['Target'] - df_pf['Prediction']
df_pf['Difference%'] = np.absolute(df_pf['Residual']/df_pf['Target']*100)
df_pf
```

	Prediction	Target	Residual	Difference%
0	10685.501696	2300.0	-8385.501696	364.587030
1	3499.255242	2800.0	-699.255242	24.973402
2	7553.285218	2500.0	-5053.285218	202.131409
3	7463.963017	6400.0	-1063.963017	16.624422
4	11353.490075	9150.0	-2203.490075	24.081859
...	...	...	...	...
769	29651.726363	29500.0	-151.726363	0.514327
770	10732.071179	9600.0	-1132.071179	11.792408
771	13922.446953	18300.0	4377.553047	23.921055
772	27487.751303	68500.0	41012.248697	59.871896
773	13491.163043	10800.0	-2691.163043	24.918176

Exploring the descriptives here gives us additional insights

Code:

```
df_pf.describe()
```

It is showing max difference percentage as 512.68.

No issues, since majority has desired difference percentage value.

	Prediction	Target	Residual	Difference%
count	774.000000	774.000000	774.000000	774.000000
mean	15946.760167	18165.817106	2219.056939	36.256693
std	13133.197604	19967.858908	10871.218143	55.066507
min	1320.562768	1200.000000	-29456.498331	0.062794
25%	7413.644234	6900.000000	-2044.191251	12.108022
50%	11568.168859	11600.000000	142.518577	23.467728
75%	20162.408805	20500.000000	3147.343497	39.563570
max	77403.055224	126000.000000	85106.162329	512.688080

- Sometimes it is useful to check these outputs manually
- To see all rows, we use the relevant pandas syntax

Code:

```
pd.options.display.max_rows = 999
```

- Moreover, to make the dataset clear, we can display the result with only 2 digits after the dot

Code:

```
pd.set_option('display.float_format', lambda x: '%.2f' % x)
```

- Finally, we sort by difference in % and manually check the model

Code:

```
df_pf.sort_values(by=['Difference%'])
```

It is not possible to put snapshot of entire table. Since, there are 774 rows.

(Please check jupyter notebook file)

	Prediction	Target	Residual	Difference%
698	30480.85	30500.00	19.15	0.06
742	16960.31	16999.00	38.69	0.23
60	12469.21	12500.00	30.79	0.25
110	25614.14	25500.00	-114.14	0.45
367	42703.68	42500.00	-203.68	0.48
369	3084.69	3100.00	15.31	0.49
769	29651.73	29500.00	-151.73	0.51
272	9749.53	9800.00	50.47	0.52
714	23118.07	22999.00	-119.07	0.52
630	8734.58	8800.00	65.42	0.74
380	3473.79	3500.00	26.21	0.75
648	21174.10	21335.00	160.90	0.75
308	8967.74	8900.00	-67.74	0.76
665	17858.02	18000.00	141.98	0.79
379	17654.84	17800.00	145.16	0.82
719	11391.95	11500.00	108.05	0.94
102	28625.56	28900.00	274.44	0.95
94	7724.17	7800.00	75.83	0.97
561	6429.03	6500.00	70.97	1.09

Thank You