B.Sc. (Hons.) Computer Science

Data Analysis and Visualization Project

Name : Khushal Rastogi

Roll No. : 19013570016 (2k19/CS/60)

Year & Sem : 3rd Year & 5th Sem

>>Problem Statement:

Project is based on the problem of "**Predicting prices of used cars?"**.

Real Life Scenario:

- Assume you have a friend Sam, who wants to sell his car.
- But the problem is, he doesn't know how much he should sell his car for?
- He wants to sell it for as much as he can; but he also wants to set the price reasonably so someone would buy it.
- The Big Question is "How can we help him determine the best price for his car?"
- Let us try to clearly define some of his problems.
  - *Is there any data on the prices of some cars & their characteristics?*
  - *What features of cars affect their prices?*
    Color? Brand? Horsepower? Something Else…?
  - *How much can a particular feature affect the price relative to others?*
- These are some of the Questions we can start thinking about at the start!

>>Understanding the Data:

- The Dataset used is an open dataset, by Jeffrey C. Schlemmer.

- This dataset is in CSV (**C**omma **S**eparated **V**alues) format, where each of the values are separated by commas.

- There are a total of **201 different entries** (rows) in the dataset.
- There are **26 different features** (columns) in the dataset.
  - Some of them are numerical variables like wheel-base, length, engine-size, horsepower, price, etc.
  - While some are categorical variables like fuel-type, body-style, brand, engine location, etc.

- **Description of Attributes** is given in the table below:

No.	Attribute name	attribute range	No.	Attribute name	attribute range
1	symboling	-3, -2, -1, 0, 1, 2, 3.	14	curb-weight	continuous from 1488 to 4066.
2	normalized-losses	continuous from 65 to 256.	15	engine-type	dohc, dohcv, l, ohc, ohcf, ohcv, rotor.
3	make	audi, bmw, etc.	16	num-of-cylinders	eight, five, four, six, three, twelve, two.
4	fuel-type	diesel, gas.	17	engine-size	continuous from 61 to 326.
5	aspiration	std, turbo.	18	fuel-system	1bbl, 2bbl, 4bbl, idi, mfi, mpfi, spdi, spfi.
6	num-of-doors	four, two.	19	bore	continuous from 2.54 to 3.94.
7	body-style	hardtop, wagon, etc.	20	stroke	continuous from 2.07 to 4.17.
8	drive-wheels	4wd, fwd, rwd.	21	compression-ratio	continuous from 7 to 23.
9	engine-location	front, rear.	22	horsepower	continuous from 48 to 288.
10	wheel-base	continuous from 86.6 120.9.	23	peak-rpm	continuous from 4150 to 6600.
11	length	continuous from 141.1 to 208.1.	24	city-mpg	continuous from 13 to 49.
12	width	continuous from 60.3 to 72.3.	25	highway-mpg	continuous from 16 to 54.
13	height	continuous from 47.8 to 59.8.	26	price	continuous from 5118 to 45400.

```
>> symboling
False    205
Name: symboling, dtype: int64
--------------------------------------------------
>> normalized-losses
False    164
True      41
Name: normalized-losses, dtype: int64
--------------------------------------------------
>> make
False    205
Name: make, dtype: int64
--------------------------------------------------
>> fuel-type
False    205
Name: fuel-type, dtype: int64
--------------------------------------------------
>> aspiration
False    205
Name: aspiration, dtype: int64
--------------------------------------------------
>> num-of-doors
False    203
True       2
Name: num-of-doors, dtype: int64
--------------------------------------------------
>> body-style
False    205
Name: body-style, dtype: int64
--------------------------------------------------
>> drive-wheels
False    205
Name: drive-wheels, dtype: int64
--------------------------------------------------
>> engine-location
False    205
Name: engine-location, dtype: int64
--------------------------------------------------
>> wheel-base
False    205
Name: wheel-base, dtype: int64
--------------------------------------------------
>> length
False    205
Name: length, dtype: int64
--------------------------------------------------
>> width
False    205
Name: width, dtype: int64
--------------------------------------------------
>> height
False    205
Name: height, dtype: int64
--------------------------------------------------
>> curb-weight
False    205
Name: curb-weight, dtype: int64
--------------------------------------------------
>> engine-type
False    205
Name: engine-type, dtype: int64
--------------------------------------------------
>> num-of-cylinders
False    205
Name: num-of-cylinders, dtype: int64
--------------------------------------------------
>> engine-size
False    205
Name: engine-size, dtype: int64
--------------------------------------------------
>> fuel-system
False    205
Name: fuel-system, dtype: int64
--------------------------------------------------
>> bore
False    201
True       4
Name: bore, dtype: int64
--------------------------------------------------
>> stroke
False    201
True       4
Name: stroke, dtype: int64
--------------------------------------------------
>> compression-ratio
False    205
Name: compression-ratio, dtype: int64
--------------------------------------------------
>> horsepower
False    203
True       2
Name: horsepower, dtype: int64
--------------------------------------------------
>> peak-rpm
False    203
True       2
Name: peak-rpm, dtype: int64
--------------------------------------------------
>> city-mpg
False    205
Name: city-mpg, dtype: int64
--------------------------------------------------
>> highway-mpg
False    205
Name: highway-mpg, dtype: int64
--------------------------------------------------
>> price
False    201
True       4
Name: price, dtype: int64
--------------------------------------------------
```

Based on the summary above, each column has 205 rows of data and seven of the columns containing missing data:

1. "normalized-losses": 41 missing data
2. "num-of-doors": 2 missing data
3. "bore": 4 missing data
4. "stroke" : 4 missing data
5. "horsepower": 2 missing data
6. "peak-rpm": 2 missing data
7. "price": 4 missing data

Deal with missing data

How to deal with missing data?

1. Drop data
   a. Drop the whole row
   b. Drop the whole column
2. Replace data
   a. Replace it by mean
   b. Replace it by frequency
   c. Replace it based on other functions

Whole columns should be dropped only if most entries in the column are empty. In our dataset, none of the columns are empty enough to drop entirely. We have some freedom in choosing which method to replace data; however, some methods may seem more reasonable than others. We will apply each method to many different columns:

Replace by mean:

- "normalized-losses": 41 missing data, replace them with mean
- "stroke" : 4 missing data, replace them with mean
- "bore" : 4 missing data, replace them with mean
- "horsepower" : 2 missing data, replace them with mean
- "peak-rpm" : 2 missing data, replace them with mean

Replace by frequency:

- "num-of-doors": 2 missing data, replace them with "four".
  - Reason: 84% sedans is four doors. Since four doors is most frequent, it is most likely to occur.

Drop the whole row:

- "price": 4 missing data, simply delete the whole row
  - Reason: price is what we want to predict. Any data entry without price data cannot be used for prediction; therefore any row now without price data is not useful to us.

Calculate the mean value for the "normalized-losses" column

In [11]:
```python
avg_norm_loss = df["normalized-losses"].astype("float").mean(axis=0)
print("Average of normalized-losses:", avg_norm_loss)
```
```
Average of normalized-losses: 122.0
```

Replace "NaN" with mean value in "normalized-losses" column

In [12]:
```python
df["normalized-losses"].replace(np.nan, avg_norm_loss, inplace=True)
```

Calculate the mean value for the "bore" column

In [13]:
```python
avg_bore=df['bore'].astype('float').mean(axis=0)
print("Average of bore:", avg_bore)
```
```
Average of bore: 3.3297512437810957
```

Replace "NaN" with the mean value in the "bore" column

In [14]:
```python
df["bore"].replace(np.nan, avg_bore, inplace=True)
```

Calculate the mean value for the "stroke" column

In [15]:
```python
avg_stroke = df["stroke"].astype("float").mean(axis = 0)
print("Average of stroke:", avg_stroke)
```
```
Average of stroke: 3.2554228855721337
```

Replace "NaN" with the mean value in the "stroke" column

In [16]:
```python
df["stroke"].replace(np.nan, avg_stroke, inplace = True)
```

Calculate the mean value for the "horsepower" column

In [17]:
```python
avg_horsepower = df['horsepower'].astype('float').mean(axis=0)
print("Average horsepower:", avg_horsepower)
```
```
Average horsepower: 104.25615763546799
```

Replace "NaN" with the mean value in the "horsepower" column

In [18]:
```python
df['horsepower'].replace(np.nan, avg_horsepower, inplace=True)
```

Calculate the mean value for "peak-rpm" column

In [19]:
```python
avg_peakrpm=df['peak-rpm'].astype('float').mean(axis=0)
print("Average peak rpm:", avg_peakrpm)
```
```
Average peak rpm: 5125.369458128079
```

Replace "NaN" with the mean value in the "peak-rpm" column

In [20]:
```python
df['peak-rpm'].replace(np.nan, avg_peakrpm, inplace=True)
```

To see which values are present in a particular column, we can use the ".value_counts()" method:

In [21]:
```python
df['num-of-doors'].value_counts()
```
Out[21]:
```
four    114
two      89
Name: num-of-doors, dtype: int64
```

We can see that four doors are the most common type.
We can also use the ".idxmax()" method to calculate the most common type automatically:

In [22]:
```python
df['num-of-doors'].value_counts().idxmax()
```
Out[22]:
```
'four'
```

The replacement procedure is very similar to what we have seen previously:

In [23]:
```python
#replace the missing 'num-of-doors' values by the most frequent
df["num-of-doors"].replace(np.nan, "four", inplace=True)
```

Finally, let's drop all rows that do not have price data:

In [24]:
```python
# simply drop whole row with NaN in "price" column
df.dropna(subset=["price"], axis=0, inplace=True)

# reset index, because we droped two rows
df.reset_index(drop=True, inplace=True)
```

In [25]:
```python
df.head()
```
Out[25]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke	compre
0	3	122.0	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	
1	3	122.0	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	
2	1	122.0	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	

5 rows × 26 columns

Good! Now, we have a dataset with no missing values.

Correct data format

We are almost there!

The last step in data cleaning is checking and making sure that all data is in the correct format (int, float, text or other).

In Pandas, we use:

.dtype() to check the data type
.astype() to change the data type

Let's list the data types for each column

In [26]:
```python
df.dtypes
```
Out[26]:
```
symboling             int64
normalized-losses    object
make                 object
fuel-type            object
aspiration           object
num-of-doors         object
body-style           object
drive-wheels         object
engine-location      object
wheel-base          float64
length              float64
width               float64
height              float64
curb-weight           int64
engine-type          object
num-of-cylinders     object
engine-size           int64
fuel-system          object
bore                 object
stroke               object
compression-ratio   float64
horsepower           object
peak-rpm             object
city-mpg              int64
highway-mpg           int64
price                object
dtype: object
```

As we can see above, some columns are not of the correct data type.
Numerical variables should have type 'float' or 'int', and variables with strings such as categories should have type 'object'
For example, 'bore' and 'stroke' variables are numerical values that describe the engines, so we should expect them to be of the type 'float' or 'int'; however, they are shown as type 'object'.
We have to convert data types into a proper format for each column using the `astype()` method.

Convert data types to proper format

In [27]:
```python
df[["bore", "stroke"]] = df[["bore", "stroke"]].astype("float")
df[["normalized-losses"]] = df[["normalized-losses"]].astype("int")
df[["price"]] = df[["price"]].astype("float")
df[["peak-rpm"]] = df[["peak-rpm"]].astype("float")
df[["horsepower"]] = df[["horsepower"]].astype("int64")
```

Let us list the columns after the conversion

In [28]:
```python
df.dtypes
```
Out[28]:
```
symboling             int64
normalized-losses     int32
make                 object
fuel-type            object
aspiration           object
num-of-doors         object
body-style           object
drive-wheels         object
engine-location      object
wheel-base          float64
length              float64
width               float64
height              float64
curb-weight           int64
engine-type          object
num-of-cylinders     object
engine-size           int64
fuel-system          object
bore                float64
stroke              float64
compression-ratio   float64
horsepower            int64
peak-rpm            float64
city-mpg              int64
highway-mpg           int64
price               float64
dtype: object
```

Wonderful!

Now we have finally obtained the cleaned dataset with no missing values with all data in its proper format.

Basic Insight of Dataset

After reading data into Pandas dataframe, it is time for us to explore the dataset.
There are several ways to obtain essential insights of the data to help us better understand our dataset.

Data Types

Data has a variety of types.
The main types stored in Pandas dataframes are **object**, **float**, **int**, **bool** and **datetime64**.
In order to better learn about each attribute, it is always good for us to know the data type of each column.

In [29]:
```python
# check the data type of data frame "df" by .dtypes
print(df.dtypes)
```
```
symboling             int64
normalized-losses     int32
make                 object
fuel-type            object
aspiration           object
num-of-doors         object
body-style           object
drive-wheels         object
engine-location      object
wheel-base          float64
length              float64
width               float64
height              float64
curb-weight           int64
engine-type          object
num-of-cylinders     object
engine-size           int64
fuel-system          object
bore                float64
stroke              float64
compression-ratio   float64
horsepower            int64
peak-rpm            float64
city-mpg              int64
highway-mpg           int64
price               float64
dtype: object
```

As shown above, it is clear to see that the data type of "symboling" and "curb-weight" are `int64`, "normalized-losses" is `object`, and "wheel-base" is `float64`, etc.

Describe

If we would like to get a statistical summary of each column e.g. count, column mean value, column standard deviation, etc., we use the describe method:

This method will provide various summary statistics, excluding `NaN` (Not a Number) values.

For example, the attribute "symboling" has 205 counts, the mean value of this column is 0.83, the standard deviation is 1.25, the minimum value is -2, 25th percentile is 0, 50th percentile is 1, 75th percentile is 2, and the maximum value is 3.

We can add an argument `include = "all"` inside the bracket to check all the columns including those that are of type object.
Let's try it again.

In [30]:
```python
# Describe all the columns in "df"
df.describe(include = "all").T
```
Out[30]:

	count	unique	top	freq	mean	std	min	25%	50%	75%	max
symboling	201.0	NaN	NaN	NaN	0.840796	1.254802	-2.0	0.0	1.0	2.0	3.0
normalized-losses	201.0	NaN	NaN	NaN	122.0	31.99625	65.0	101.0	122.0	137.0	256.0
make	201	22	toyota	32	NaN	NaN	NaN	NaN	NaN	NaN	NaN
fuel-type	201	2	gas	181	NaN	NaN	NaN	NaN	NaN	NaN	NaN
aspiration	201	2	std	165	NaN	NaN	NaN	NaN	NaN	NaN	NaN
num-of-doors	201	2	four	115	NaN	NaN	NaN	NaN	NaN	NaN	NaN
body-style	201	5	sedan	94	NaN	NaN	NaN	NaN	NaN	NaN	NaN
drive-wheels	201	3	fwd	118	NaN	NaN	NaN	NaN	NaN	NaN	NaN
engine-location	201	2	front	198	NaN	NaN	NaN	NaN	NaN	NaN	NaN
wheel-base	201.0	NaN	NaN	NaN	98.797015	6.066366	86.6	94.5	97.0	102.4	120.9
length	201.0	NaN	NaN	NaN	174.200995	12.322175	141.1	166.8	173.2	183.5	208.1
width	201.0	NaN	NaN	NaN	65.889055	2.101471	60.3	64.1	65.5	66.6	72.0
height	201.0	NaN	NaN	NaN	53.766667	2.447822	47.8	52.0	54.1	55.5	59.8
curb-weight	201.0	NaN	NaN	NaN	2555.666667	517.296727	1488.0	2169.0	2414.0	2926.0	4066.0
engine-type	201	6	ohc	145	NaN	NaN	NaN	NaN	NaN	NaN	NaN
num-of-cylinders	201	7	four	157	NaN	NaN	NaN	NaN	NaN	NaN	NaN
engine-size	201.0	NaN	NaN	NaN	126.875622	41.546834	61.0	98.0	120.0	141.0	326.0
fuel-system	201	8	mpfi	92	NaN	NaN	NaN	NaN	NaN	NaN	NaN
bore	201.0	NaN	NaN	NaN	3.330692	0.268072	2.54	3.15	3.31	3.58	3.94
stroke	201.0	NaN	NaN	NaN	3.256874	0.316048	2.07	3.11	3.29	3.41	4.17
compression-ratio	201.0	NaN	NaN	NaN	10.164279	4.004965	7.0	8.6	9.0	9.4	23.0
horsepower	201.0	NaN	NaN	NaN	103.402985	37.36565	48.0	70.0	95.0	116.0	262.0
peak-rpm	201.0	NaN	NaN	NaN	5117.665368	478.113805	4150.0	4800.0	5125.369458	5500.0	6600.0
city-mpg	201.0	NaN	NaN	NaN	25.179104	6.42322	13.0	19.0	24.0	30.0	49.0
highway-mpg	201.0	NaN	NaN	NaN	30.686567	6.81515	16.0	25.0	30.0	34.0	54.0
price	201.0	NaN	NaN	NaN	13207.129353	7947.066342	5118.0	7775.0	10295.0	16500.0	45400.0

Now it provides the statistical summary of all the columns, including object-typed attributes.
We can now see how many unique values there, which one is the top value and the frequency of top value in the object-typed columns.
Some values in the table above show as "NaN". This is because those numbers are not available regarding a particular column type.

Info

Another method we can use to check your dataset is:

In [31]:
```python
#look at the info of dataframe "df".
df.info()
```
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 201 entries, 0 to 200
Data columns (total 26 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   symboling          201 non-null    int64  
 1   normalized-losses  201 non-null    int32  
 2   make               201 non-null    object 
 3   fuel-type          201 non-null    object 
 4   aspiration         201 non-null    object 
 5   num-of-doors       201 non-null    object 
 6   body-style         201 non-null    object 
 7   drive-wheels       201 non-null    object 
 8   engine-location    201 non-null    object 
 9   wheel-base         201 non-null    float64
 10  length             201 non-null    float64
 11  width              201 non-null    float64
 12  height             201 non-null    float64
 13  curb-weight        201 non-null    int64  
 14  engine-type        201 non-null    object 
 15  num-of-cylinders   201 non-null    object 
 16  engine-size        201 non-null    int64  
 17  fuel-system        201 non-null    object 
 18  bore               201 non-null    float64
 19  stroke             201 non-null    float64
 20  compression-ratio  201 non-null    float64
 21  horsepower         201 non-null    int64  
 22  peak-rpm           201 non-null    float64
 23  city-mpg           201 non-null    int64  
 24  highway-mpg        201 non-null    int64  
 25  price              201 non-null    float64
dtypes: float64(9), int32(1), int64(6), object(10)
memory usage: 40.2+ KB
```

It provides a concise summary of your DataFrame.
This method prints information about a DataFrame including the index, dtype and columns, non-null values and memory usage.

Data Normalization

Why normalization?

Normalization is the process of transforming values of several variables into a similar range.
Typical normalizations include scaling the variable so the variable average is 0, scaling the variable so the variance is 1, or scaling the variable so the variable values range from 0 to 1.

Example

Target: would like to normalize "length", "width" and "height" so their value ranges from 0 to 1.

Approach: replace original value by (original value)/(maximum value).

In [32]:
```python
# replace (original value) by (original value)/(maximum value)
df['length'] = df['length']/df['length'].max()
df['width'] = df['width']/df['width'].max()
df['height'] = df['height']/df['height'].max()

# show the scaled columns
df[["length","width","height"]].head()
```

	length	width	height
0	0.811148	0.890278	0.816054
1	0.811148	0.890278	0.816054
2	0.822681	0.909722	0.876254
3	0.848630	0.919444	0.908027
4	0.848630	0.922222	0.908027

Here we can see we've normalized "length", "width" and "height" in the range of [0,1].

Binning

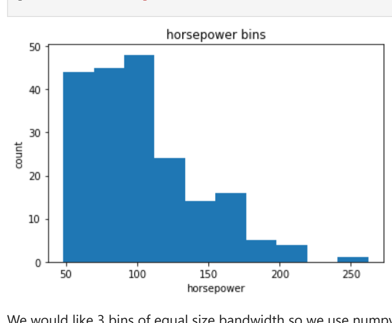
Why binning?

Binning is a process of transforming continuous numerical variables into discrete categorical 'bins' for grouped analysis.

In our dataset, "horsepower" is a real valued variable ranging from 48 to 288 and it has 57 unique values. What if we only care about the price difference between cars with high horsepower, medium horsepower, and little horsepower (3 types)? Can we rearrange them into three 'bins' to simplify analysis?

Let's Plot the Histogram of horsepower to see what the distribution of horsepower looks like.

```
import matplotlib.pyplot as plt  #For Visualizing Data
plt.hist(df["horsepower"])

# set x/y labels and plot title
plt.xlabel("horsepower")
plt.ylabel("count")
plt.title("horsepower bins");
```



We would like 3 bins of equal size bandwidth so we use numpy's `linspace(start_value, end_value, numbers_generated)` function.
Since we want to include the minimum value of horsepower, we want to set start_value = min(df["horsepower"]).
Since we want to include the maximum value of horsepower, we want to set end_value = max(df["horsepower"]).
Since we are building 3 bins of equal length, there should be 4 dividers, so numbers_generated = 4.

We build a bin array with a minimum value to a maximum value by using the bandwidth calculated above
The values will determine when one bin ends and another begins.

```
bins = np.linspace(min(df["horsepower"]), max(df["horsepower"]), 4)
bins
```

```
array([ 48.        , 119.33333333, 190.66666667, 262.        ])
```

```
#We set group  names:
group_names = ['Low', 'Medium', 'High']
```

We apply the function "cut" to determine what each value of `df['horsepower']` belongs to.

```
df['horsepower-binned'] = pd.cut(df['horsepower'], bins, labels=group_names, include_lowest=True )
df[['horsepower','horsepower-binned']].head(15).T
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
horsepower	111	111	154	102	115	110	110	110	140	101	101	121	121	121	182
horsepower-binned	Low	Low	Medium	Low	Low	Low	Low	Low	Medium	Low	Low	Medium	Medium	Medium	Medium

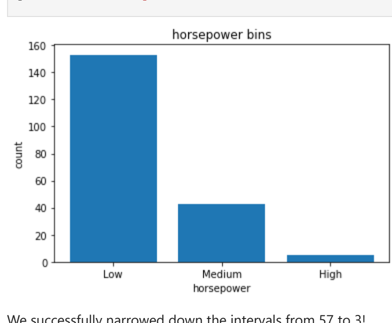
Let's see the number of vehicles in each bin:

```
df["horsepower-binned"].value_counts()
```

```
Low       153
Medium     43
High        5
Name: horsepower-binned, dtype: int64
```

Let's plot the distribution of each bin:

```
plt.bar(group_names, df["horsepower-binned"].value_counts())

# set x/y labels and plot title
plt.xlabel("horsepower")
plt.ylabel("count")
plt.title("horsepower bins");
```



We successfully narrowed down the intervals from 57 to 3!

Indicator Variable (or Dummy Variable)

An indicator variable (or dummy variable) is a numerical variable used to label categories. They are called 'dummies' because the numbers themselves don't have inherent meaning.
We use indicator variables so we can use categorical variables for regression analysis.

Creating an indicator variable for "fuel-type".

We see the column "fuel-type" has two unique values: "gas" or "diesel". Regression doesn't understand words, only numbers. To use this attribute in regression analysis, we convert "fuel-type" to indicator variables.

We will use pandas' method 'get_dummies' to assign numerical values to different categories of fuel type.

```
dummy_variable_1 = pd.get_dummies(df["fuel-type"])
dummy_variable_1.head()
```

	diesel	gas
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1

Change the column names for clarity:

```
#change the column names for clarity
dummy_variable_1.rename(columns={'gas':'fuel-type-gas', 'diesel':'fuel-type-diesel'}, inplace=True)
dummy_variable_1.head()
```

	fuel-type-diesel	fuel-type-gas
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1

In the dataframe, column 'fuel-type' has values for 'gas' and 'diesel' as 0s and 1s now.

```
# merge data frame "df" and "dummy_variable_1"
df = pd.concat([df, dummy_variable_1], axis=1)

# drop original column "fuel-type" from "df"
df.drop("fuel-type", axis = 1, inplace=True)
```

```
df.head()
```

	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	length	...	stroke	compression-ratio	horsepower
0	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	2.68	9.0	111
1	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	2.68	9.0	111
2	1	122	alfa-romero	std	two	hatchback	rwd	front	94.5	0.822681	...	3.47	9.0	154
3	2	164	audi	std	four	sedan	fwd	front	99.8	0.848630	...	3.40	10.0	102
4	2	164	audi	std	four	sedan	4wd	front	99.4	0.848630	...	3.40	8.0	115

5 rows × 28 columns

The last two columns are now the indicator variable representation of the fuel-type variable. They're all 0s and 1s now.

Nice, The Data is all cleaned.
Now, We Can use it efficiently for further analysis and model development.

Exploratory Data Analysis

Table of Contents

What are the main characteristics that have the most impact on the car price?

Cleaned Data

```
df.head()
```

	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	length	...	stroke	compression-ratio	horsepower
0	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	2.68	9.0	111
1	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	2.68	9.0	111
2	1	122	alfa-romero	std	two	hatchback	rwd	front	94.5	0.822681	...	3.47	9.0	154
3	2	164	audi	std	four	sedan	fwd	front	99.8	0.848630	...	3.40	10.0	102
4	2	164	audi	std	four	sedan	4wd	front	99.4	0.848630	...	3.40	8.0	115

5 rows × 28 columns

Analyzing Individual Feature Patterns Using Visualization

Importing visualization packages "Matplotlib" and "Seaborn".

```
import matplotlib.pyplot as plt
import seaborn as sns
```

When visualizing individual variables, it is important to first understand what type of variable you are dealing with. This will help us find the right visualization method for that variable.

```
# list the data types for each column
print(df.dtypes)
```

```
symboling              int64
normalized-losses      int32
make                  object
aspiration            object
num-of-doors          object
body-style            object
drive-wheels          object
engine-location       object
wheel-base           float64
length               float64
width                float64
height               float64
curb-weight            int64
engine-type           object
num-of-cylinders      object
engine-size            int64
fuel-system           object
bore                 float64
stroke               float64
compression-ratio    float64
horsepower             int32
peak-rpm             float64
city-mpg               int64
highway-mpg            int64
price                float64
horsepower-binned   category
fuel-type-diesel       uint8
fuel-type-gas          uint8
dtype: object
```

We can calculate the correlation between variables of type "int64" or "float64" using the method "corr":

```
df.corr()
```

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore	stroke	compression-ratio
symboling	1.000000	0.466264	-0.535987	-0.365404	-0.242423	-0.550160	-0.233118	-0.110581	-0.140019	-0.008245	-0.182196
normalized-losses	0.466264	1.000000	-0.056661	0.019424	0.086802	-0.373737	0.099404	0.112360	-0.029862	0.055563	-0.114713
wheel-base	-0.535987	-0.056661	1.000000	0.876024	0.814507	0.590742	0.782097	0.572027	0.493244	0.158502	0.250313
length	-0.365404	0.019424	0.876024	1.000000	0.857170	0.492063	0.880665	0.685025	0.608971	0.124139	0.159733
width	-0.242423	0.086802	0.814507	0.857170	1.000000	0.306002	0.866201	0.729436	0.544885	0.188829	0.189867
height	-0.550160	-0.373737	0.590742	0.492063	0.306002	1.000000	0.307581	0.074694	0.180449	-0.062704	0.259737
curb-weight	-0.233118	0.099404	0.782097	0.880665	0.866201	0.307581	1.000000	0.849072	0.644060	0.167562	0.156433
engine-size	-0.110581	0.112360	0.572027	0.685025	0.729436	0.074694	0.849072	1.000000	0.572609	0.209523	0.028889
bore	-0.140019	-0.029862	0.493244	0.608971	0.544885	0.180449	0.644060	0.572609	1.000000	-0.055390	0.001263
stroke	-0.008245	0.055563	0.158502	0.124139	0.188829	-0.062704	0.167562	0.209523	-0.055390	1.000000	0.187923
compression-ratio	-0.182196	-0.114713	0.250313	0.159733	0.189867	0.259737	0.156433	0.028889	0.001263	0.187923	1.000000
horsepower	0.075819	0.217299	0.371147	0.579821	0.615077	-0.087027	0.757976	0.822676	0.566936	0.098462	-0.214514
peak-rpm	0.279740	0.239543	-0.360305	-0.285970	-0.245800	-0.309974	-0.279361	-0.256733	-0.267392	-0.065713	-0.435780
city-mpg	-0.035527	-0.225016	-0.470606	-0.665192	-0.633531	-0.049800	-0.749543	-0.650546	-0.582027	-0.034696	0.331425
highway-mpg	0.036233	-0.181877	-0.543304	-0.698142	-0.680635	-0.104812	-0.794889	-0.679571	-0.591309	-0.035201	0.268465
price	-0.082391	0.133999	0.584642	0.690628	0.751265	0.135486	0.834415	0.872335	0.543155	0.082310	0.071107
fuel-type-diesel	-0.196735	-0.101546	0.307237	0.211187	0.244356	0.281578	0.221046	0.070779	0.054458	0.241303	0.985231
fuel-type-gas	0.196735	0.101546	-0.307237	-0.211187	-0.244356	-0.281578	-0.221046	-0.070779	-0.054458	-0.241303	-0.985231

The diagonal elements are always one.

```
#Find the correlation between the following columns: bore, stroke, compression-ratio, and horsepower.
df[['bore', 'stroke', 'compression-ratio', 'horsepower']].corr()
```

	bore	stroke	compression-ratio	horsepower
bore	1.000000	-0.055390	0.001263	0.566936
stroke	-0.055390	1.000000	0.187923	0.098462
compression-ratio	0.001263	0.187923	1.000000	-0.214514
horsepower	0.566936	0.098462	-0.214514	1.000000

Continuous Numerical Variables:

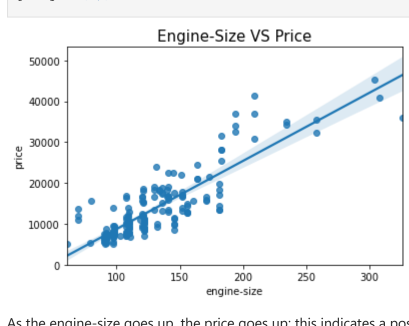
Continuous numerical variables are variables that may contain any value within some range. They can be of type "int64" or "float64". A great way to visualize these variables is by using scatterplots with fitted lines.

In order to start understanding the (linear) relationship between an individual variable and the price, we can use "regplot" which plots the scatterplot plus the fitted regression line for the data.

Positive Linear Relationship

Let's find the **scatterplot** of "engine-size" and "price".

```
# Engine size as potential predictor variable of price
sns.regplot(x="engine-size", y="price", data=df)
plt.title("Engine-Size VS Price",size=15)
plt.ylim(0,)
```



As the engine-size goes up, the price goes up: this indicates a positive direct correlation between these two variables. Engine size seems like a pretty good predictor of price since the regression line is almost a perfect diagonal line.

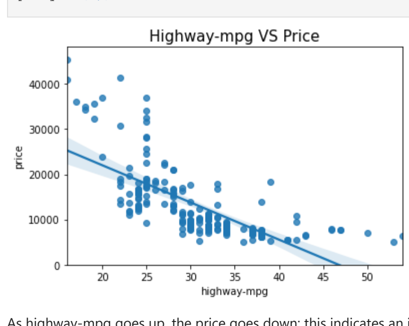
We can examine the correlation between 'engine-size' and 'price' and see that it's approximately 0.87.

```
df[["engine-size", "price"]].corr()
```

	engine-size	price
engine-size	1.000000	0.872335
price	0.872335	1.000000

Highway-mpg is a potential predictor variable of price.
Let's find the scatterplot of "highway-mpg" and "price".

```
sns.regplot(x="highway-mpg", y="price", data=df)
plt.title("Highway-mpg VS Price",size=15)
plt.ylim(0,)
```



As highway-mpg goes up, the price goes down: this indicates an inverse/negative relationship between these two variables. Highway-mpg could potentially be a predictor of price.

We can examine the correlation between 'highway-mpg' and 'price' and see it's approximately -0.704.

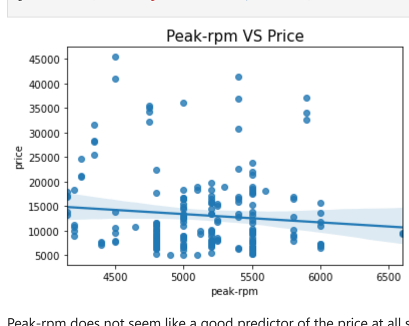
```
df[['highway-mpg', 'price']].corr()
```

	highway-mpg	price
highway-mpg	1.000000	-0.704692
price	-0.704692	1.000000

Weak Linear Relationship

Let's see if "peak-rpm" is a predictor variable of "price".

```
sns.regplot(x="peak-rpm", y="price", data=df)
plt.title("Peak-rpm VS Price",size=15);
```



Peak-rpm does not seem like a good predictor of the price at all since the regression line is close to horizontal. Also, the data points are very scattered and far from the fitted line, showing lots of variability. Therefore, it's not a reliable variable.

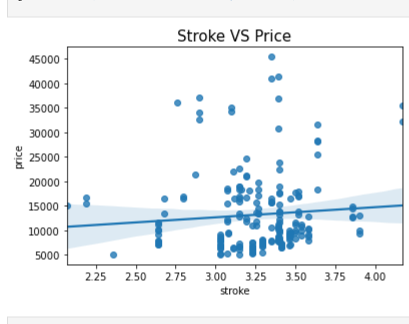
We can examine the correlation between 'peak-rpm' and 'price' and see it's approximately -0.101616.

```
df[['peak-rpm','price']].corr()
```

	peak-rpm	price
peak-rpm	1.000000	-0.101616
price	-0.101616	1.000000

Let's see if "stroke" is a predictor variable of "price".

```
#There is a weak correlation between the variable 'stroke' and 'price.'
sns.regplot(x="stroke", y="price", data=df)
plt.title("Stroke VS Price",size=15);
```



199	rwd	sedan	22470.0
200	rwd	sedan	22625.0

201 rows × 3 columns

To see if different types of 'drive-wheels' impact 'price', we group the data.

```
In [80]: grouped_test2=df_gptest[['drive-wheels', 'price']].groupby(['drive-wheels'])
grouped_test2.head(2)
```

Out[80]:

	drive-wheels	price
0	rwd	13495.0
1	rwd	16500.0
3	fwd	13950.0
4	4wd	17450.0
5	fwd	15250.0
136	4wd	7603.0

We can obtain the values of the method group using the method "get_group".

```
In [81]: grouped_test2.get_group('4wd')['price']
```

```
Out[81]: 4      17450.0
136     7603.0
140     9233.0
141    11259.0
144     8013.0
145    11694.0
150     7898.0
151     8778.0
Name: price, dtype: float64
```

We can use the function 'f_oneway' in the module 'stats' to obtain the **F-test score** and **P-value**.

```
In [82]: # ANOVA
f_val, p_val = stats.f_oneway(grouped_test2.get_group('fwd')['price'], grouped_test2.get_group('rwd')['price'],

print( "ANOVA results: F=", f_val, ", P =", p_val)
```

```
ANOVA results: F= 67.95406500780399 , P = 3.3945443577151245e-23
```

This is a great result with a **large F-test score** showing a strong correlation and a P-value < 0.001 implying almost certain statistical significance. But does this mean all three tested groups are all this highly correlated?

Let's examine them separately.

fwd and rwd

```
In [83]: f_val, p_val = stats.f_oneway(grouped_test2.get_group('fwd')['price'], grouped_test2.get_group('rwd')['price'])

print( "ANOVA results: F=", f_val, ", P =", p_val )
```

```
ANOVA results: F= 130.5533160959111 , P = 2.2355306355677845e-23
```

Conclusion:
They are highly correlated meaning, prices between "fwd" and "rwd" is significantly different as the F-score is large (130) and the p-value is < 0.001.

4wd and rwd

```
In [84]: f_val, p_val = stats.f_oneway(grouped_test2.get_group('4wd')['price'], grouped_test2.get_group('rwd')['price'])

print( "ANOVA results: F=", f_val, ", P =", p_val)
```

```
ANOVA results: F= 8.580681368924756 , P = 0.004411492211225333
```

Conclusion:
They are moderately correlated meaning, prices between "4wd" and "rwd" is moderately different as the F-score is 8 and the p-value is > 0.001 and <0.05.

4wd and fwd

```
In [85]: f_val, p_val = stats.f_oneway(grouped_test2.get_group('4wd')['price'], grouped_test2.get_group('fwd')['price'])

print("ANOVA results: F=", f_val, ", P =", p_val)
```

```
ANOVA results: F= 0.665465750252303 , P = 0.41620116697845666
```

Conclusion:
They are weakly correlated meaning, prices between "4wd" and "fwd" are comparable as the F-score is small (0.6) and the p-value is > 0.05.

Main Conclusion: Important Variables

We now have a better idea of our data and which variables are important to take into account when predicting the car price.
We have narrowed it down to the following variables:

Continuous numerical variables:

- Length
- Width
- Curb-weight
- Engine-size
- Horsepower
- City-mpg
- Highway-mpg
- Wheel-base
- Bore

Categorical variables:

- Drive-wheels

As we now move into building machine learning models to automate our analysis, feeding the model with variables that meaningfully affect our target variable will improve our model's prediction performance.

Model Development

Table of Contents

Now, We will develop several models that will predict the price of the car using the variables or features. This is just an estimate but should give us an objective idea of how much the car should cost.

Some questions we want to ask now are:

- Do I know if the dealer is offering fair value for my trade-in?
- Do I know if I put a fair value on my car?

In data analytics, we use **Model Development** to help us predict future observations from the data we have.

A model will help us understand the exact relationship between different variables and how these variables are used to predict the result.

Linear Regression and Multiple Linear Regression

Linear Regression

One example of a Data Model that we will be using is:

Simple Linear Regression
Simple Linear Regression is a method to help us understand the relationship between two variables:

- The predictor/independent variable (X)
- The response/dependent variable (that we want to predict)(Y)

The result of Linear Regression is a **linear function** that predicts the response (dependent) variable as a function of the predictor (independent) variable.

$$Y: Response\ Variable$$
$$X: Predictor\ Variables$$

Linear Function

$$Yhat = a + bX$$

- **"a"** refers to the **intercept** of the regression line, in other words: the value of Y when X is 0
- **"b"** refers to the **slope** of the regression line, in other words: the value with which Y changes when X increases by 1 unit

Let's load the modules for linear regression:

```
In [86]: from sklearn.linear_model import LinearRegression
```

Creating the linear regression object:

```
In [87]: lm = LinearRegression()
lm
```

```
Out[87]: LinearRegression()
```

Let's look at how highway-mpg can help us predict car price. Using simple linear regression, we will create a linear function with "highway-mpg" as the predictor variable and the "price" as the response variable.

```
In [88]: X = df[['highway-mpg']]
Y = df['price']
```

Let Fit (Train) the linear model using highway-mpg:

```
In [89]: lm.fit(X,Y)
```

```
Out[89]: LinearRegression()
```

Let's output a prediction:

```
In [90]: Yhat=lm.predict(X)
Yhat[0:5]
```

```
Out[90]: array([16236.50464347, 16236.50464347, 17058.23802179, 13771.3045085 ,
       20345.17153508])
```

Value of the intercept (a)?

```
In [91]: lm.intercept_
```

```
Out[91]: 38423.305858157386
```

Value of the slope (b)?

```
In [92]: lm.coef_
```

```
Out[92]: array([-821.73337832])
```

What is the final estimated linear model we get?

As we saw above, we should get a final linear model with the structure:

$$Yhat = a + bX$$

Plugging in the actual values we get:
Price = 38423.31 - 821.73 x **highway-mpg**

Creating another linear regression object called "lm1".

```
In [93]: lm1 = LinearRegression()
lm1
```

```
Out[93]: LinearRegression()
```

Train the model using "engine-size" as the independent variable and "price" as the dependent variable.

```
In [94]: lm1.fit(df[['engine-size']], df['price'])
lm1
```

```
Out[94]: LinearRegression()
```

Finding the slope and intercept of the model.

Slope

```
In [95]: # Slope
lm1.coef_
```

```
Out[95]: array([166.86001569])
```

Intercept

```
In [96]: # Intercept
lm1.intercept_
```

```
Out[96]: -7963.338906281049
```

Equation of the predicted line.

Yhat=-7963.34 + 166.86*X*
*Price=-7963.34 + 166.86*engine-size

Multiple Linear Regression

What if we want to predict car price using more than one variable?

If we want to use more variables in our model to predict car price, we can use **Multiple Linear Regression**.
Multiple Linear Regression is very similar to Simple Linear Regression, but this method is used to explain the relationship between one continuous response (dependent) variable and **two or more** predictor (independent) variables.
Most of the real-world regression models involve multiple predictors. We will illustrate the structure by using four predictor variables:

$$Y: Response\ Variable$$
$$X\_1: Predictor\ Variable\ 1$$
$$X\_2: Predictor\ Variable\ 2$$
$$X\_3: Predictor\ Variable\ 3$$
$$X\_4: Predictor\ Variable\ 4$$

$$a: intercept$$
$$b\_1: coefficients\ of\ Variable\ 1$$
$$b\_2: coefficients\ of\ Variable\ 2$$
$$b\_3: coefficients\ of\ Variable\ 3$$
$$b\_4: coefficients\ of\ Variable\ 4$$

The equation is given by:

$$Yhat = a + b\_1X\_1 + b\_2X\_2 + b\_3X\_3 + b\_4X\_4$$

From the previous section we know that other good predictors of price could be:

- Horsepower
- Curb-weight
- Engine-size
- Highway-mpg

Let's develop a model using these variables as the predictor variables.

```
In [97]: Z = df[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']]
```

Fit the linear model using the four above-mentioned variables.

```
In [98]: lm.fit(Z, df['price'])
```

```
Out[98]: LinearRegression()
```

value of the intercept(a)?

```
In [99]: lm.intercept_
```

```
Out[99]: -15811.863767729243
```

values of the coefficients (b1, b2, b3, b4)?

```
In [100…: lm.coef_
```

```
Out[100…: array([53.53022809,  4.70805253, 81.51280006, 36.1593925 ])
```

What is the final estimated linear model that we get?

As we saw above, we should get a final linear function with the structure:

$$Yhat = a + b\_1X\_1 + b\_2X\_2 + b\_3X\_3 + b\_4X\_4$$

Price = -15678.742628061467 + (52.65851272 x **horsepower**) + (4.69878948 x **curb-weight**) + (81.95906216 x **engine-size**) + (33.58258185 x **highway-mpg**)

Model Evaluation Using Visualization

Now that we've developed some models, how do we evaluate our models and choose the best one?
One way to do this is by using a visualization.

Import the visualization package, seaborn:

```
In [101…: # import the visualization package: seaborn
import seaborn as sns
```

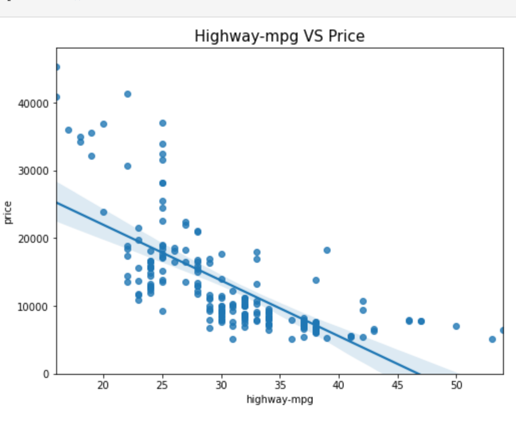
Regression Plot

When it comes to simple linear regression, an excellent way to visualize the fit of our model is by using **regression plots**.

This plot will show a combination of a scattered data points (a **scatterplot**), as well as the fitted **linear regression** line going through the data. This will give us a reasonable estimate of the relationship between the two variables, the strength of the correlation, as well as the direction (positive or negative correlation).

Let's visualize **highway-mpg** as potential predictor variable of price:

```
In [102…: plt.figure(figsize=(8, 6))
sns.regplot(x="highway-mpg", y="price", data=df)
plt.title("Highway-mpg VS Price",size=15)
plt.ylim(0,)
plt.show()
```

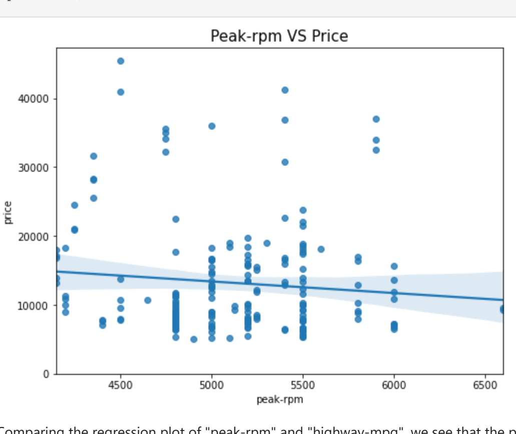


We can see from this plot that price is negatively correlated to highway-mpg since the regression slope is negative.

One thing to keep in mind when looking at a regression plot is to pay attention to how scattered the data points are around the regression line. This will give you a good indication of the variance of the data and whether a linear model would be the best fit or not. If the data is too far off from the line, this linear model might not be the best model for this data.

Let's compare this plot to the regression plot of "peak-rpm".</p>

```
In [103…: plt.figure(figsize=(8, 6))
sns.regplot(x="peak-rpm", y="price", data=df)
plt.title("Peak-rpm VS Price",size=15)
plt.ylim(0,)
plt.show()
```



Comparing the regression plot of "peak-rpm" and "highway-mpg", we see that the points for "highway-mpg" are much closer to the generated line and, on average decrease.
The points for "peak-rpm" have more spread around the predicted line and it is much harder to determine if the points are decreasing or increasing as the "highway-mpg" increases.

Using the method ".corr()" to verify the finding.

```
In [104…: # The variable "highway-mpg" has a stronger correlation with "price",
# it is approximate -0.704692 compared to "peak-rpm" which is approximate -0.101616.

df[["peak-rpm","highway-mpg","price"]].corr()
```

Out[104…:

	peak-rpm	highway-mpg	price
peak-rpm	1.000000	-0.058598	-0.101616
highway-mpg	-0.058598	1.000000	-0.704692
price	-0.101616	-0.704692	1.000000

Residual Plot

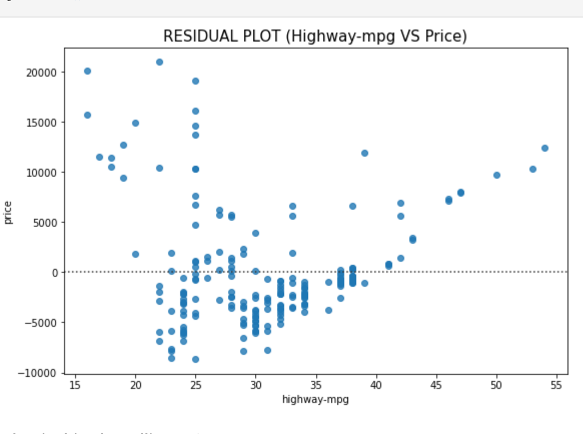
A good way to visualize the variance of the data is to use a residual plot.

What is a **residual**?
The difference between the observed value (y) and the predicted value (Yhat) is called the residual (e).
When we look at a regression plot, the residual is the distance from the data point to the fitted regression line.

So what is a **residual plot**?
A residual plot is a graph that shows the residuals on the vertical y-axis and the independent variable on the horizontal x-axis.

What do we pay attention to when looking at a residual plot?
We look at the spread of the residuals:
- If the points in a residual plot are **randomly spread out around the x-axis**, then a **linear model is appropriate** for the data. *Why is that?* Randomly spread out residuals means that the variance is constant, and thus the linear model is a good fit for this data.

```
In [105…: plt.figure(figsize=(9,6))
sns.residplot(x='highway-mpg',y='price',data = df)
plt.title("RESIDUAL PLOT (Highway-mpg VS Price)",size=15)
plt.show()
```



What is this plot telling us?

We can see from this residual plot that the residuals are not randomly spread around the x-axis, leading us to believe that maybe a non-linear model is more appropriate for this data.

Multiple Linear Regression

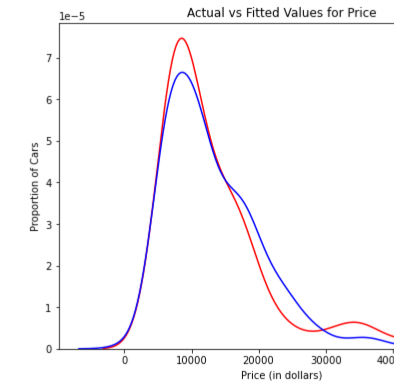
How do we visualize a model for Multiple Linear Regression?
This gets a bit more complicated because you can't visualize it with regression or residual plot.

One way to look at the fit of the model is by looking at the **kde plot** (Kernel Density Estimate).
We can look at the distribution of the fitted values that result from the model and compare it to the distribution of the actual values.

First, let's make a prediction:

```
In [106…: Y_hat = lm.predict(Z)
```

```
In [107…: plt.figure(figsize=(8, 6))

ax1 = sns.kdeplot(x=df['price'], data=df, color="r", label="Actual Value")
sns.kdeplot(x=Y_hat, data=df, color="b", label="Fitted Values" , ax=ax1)

# Adding Legends
ax1.legend()

plt.title('Actual vs Fitted Values for Price')
plt.xlabel('Price (in dollars)')
plt.ylabel('Proportion of Cars')

plt.show()
```



We can see that the fitted values are reasonably close to the actual values since the two distributions overlap a bit. However, there is definitely some room for improvement.

Polynomial Regression

Polynomial regression is a particular case of the general linear regression model or multiple linear regression models.
We get non-linear relationships by squaring or setting higher-order terms of the predictor variables.

There are different orders of polynomial regression:

Quadratic - 2nd Order

$$Yhat = a + b_1 X + b_2 X^2$$

Cubic - 3rd Order

$$Yhat = a + b_1 X + b_2 X^2 + b_3 X^3$$

Higher-Order:

$$Y = a + b_1 X + b_2 X^2 + b_3 X^3 \ldots.$$

We saw earlier that a linear model did not provide the best fit while using "highway-mpg" as the predictor variable. Let's see if we can try fitting a polynomial model to the data instead.

We will use the following function to plot the data:

```
def PlotPolly(model, independent_variable, dependent_variable, Name):
    x_new = np.linspace(15, 55, 100)
    y_new = model(x_new)

    plt.figure(figsize=(8,6))
    plt.plot(independent_variable, dependent_variable, '.', x_new, y_new, '-')
    plt.title('Polynomial Fit for Price ~ '+Name,size=15)
    ax = plt.gca()
    ax.set_facecolor((0.898, 0.898, 0.898))
    fig = plt.gcf()
    plt.ylim(0,)
    plt.xlabel(Name)
    plt.ylabel('Price of Cars')

    plt.show()
    plt.close()
```

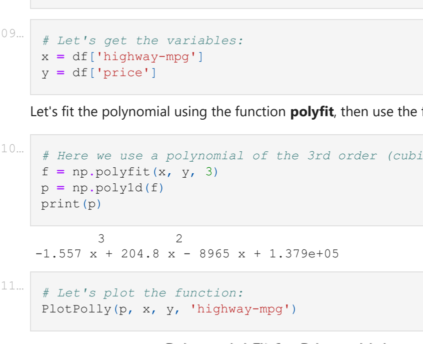
```
# Let's get the variables:
x = df['highway-mpg']
y = df['price']
```

Let's fit the polynomial using the function **polyfit**, then use the function **poly1d** to display the polynomial function.

```
# Here we use a polynomial of the 3rd order (cubic).
f = np.polyfit(x, y, 3)
p = np.poly1d(f)
print(p)
```

```
        3         2
-1.557 x + 204.8 x - 8965 x + 1.379e+05
```

```
# Let's plot the function:
PlotPolly(p, x, y, 'highway-mpg')
```

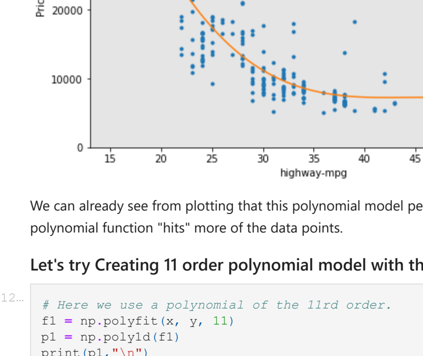


We can already see from plotting that this polynomial model performs better than the linear model. This is because the generated polynomial function "hits" more of the data points.

Let's try Creating 11 order polynomial model with the variables x and y from above.

```
# Here we use a polynomial of the 11rd order.
f1 = np.polyfit(x, y, 11)
p1 = np.poly1d(f1)
print(p1,"\n")
PlotPolly(p1,x,y,'Highway-MPG')
```

```
            11             10             9           8         7
-1.243e-08 x  + 4.722e-06 x  - 0.0008028 x + 0.08056 x - 5.297 x
          6        5             4             3             2
 + 239.5 x - 7588 x + 1.684e+05 x - 2.565e+06 x + 2.551e+07 x - 1.491e+08 x + 3.879e+08
```



The analytical expression for Multivariate Polynomial function gets complicated. For example, the expression for a second-order (degree=2) polynomial with two variables is given by:

$$Yhat = a + b\_1 X\_1 + b\_2 X\_2 + b\_3 X\_1 X\_2 + b\_4 X\_1^2 + b\_5 X\_2^2$$

We can perform a polynomial transform on multiple features.
First, we import the module:

```
from sklearn.preprocessing import PolynomialFeatures
```

We create a **PolynomialFeatures** object of degree 2:

```
pr=PolynomialFeatures(degree=2)
pr
```

```
PolynomialFeatures()
```

```
Z_pr=pr.fit_transform(Z)
```

In the original data, there are 201 samples and 4 features.

```
Z.shape
```

```
(201, 4)
```

After the transformation, there are 201 samples and 15 features.

```
Z_pr.shape
```

```
(201, 15)
```

Pipeline

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.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

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

```
Input=[('scale',StandardScaler()), ('polynomial', PolynomialFeatures(degree=2,include_bias=False)), ('model',Li
```

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

```
pipe=Pipeline(Input)
pipe
```

```
    plt.show()
    plt.close()
```

Training and Testing

An important step in testing the model is to split your data into training and testing data. We will place the target data **price** in a separate dataframe **y_data**:

```
y_data = df['price']
```

Drop price data in **x_data**

```
x_data=df.drop('price',axis=1)
```

Now we randomly split our data into training and testing data using the function **train_test_split**.

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.15, random_state=1)

print("Number of test samples    :", x_test.shape[0])
print("Number of training samples:",x_train.shape[0])
```

```
Number of test samples    : 31
Number of training samples: 170
```

The **test_size** parameter sets the proportion of data that is split into the testing set.
In the above, the testing set is set to 15% of the total dataset.

Let's import **LinearRegression** from the module **linear_model**.

```
from sklearn.linear_model import LinearRegression
```

We create a Linear Regression object:

```
lre=LinearRegression()
```

We fit the model using the feature horsepower

```
lre.fit(x_train[['horsepower']], y_train)
```

```
LinearRegression()
```

Let's Calculate the R^2 on the test data:

```
lre.score(x_test[['horsepower']], y_test)
```

```
0.7076967079117261
```

We can see the R^2 is smaller using the train data.

```
lre.score(x_train[['horsepower']], y_train)
```

```
0.6450110239384648
```

Sometimes we don't have sufficient testing data; as a result, we may want to perform Cross-validation. Let's go over several methods that you can use for Cross-validation.

Cross-validation Score

Lets import **model_selection** from the module **cross_val_score**.

```
from sklearn.model_selection import cross_val_score
```

We input the object, the feature in this case **horsepower**, the target data (y_data). The parameter **cv** determines the number of folds; in this case 4.

```
Rcross = cross_val_score(lre, x_data[['horsepower']], y_data, cv=4)
```

The default scoring is R^2; each element in the array has the average R^2 value in the fold:

```
Rcross
```

```
array([0.77465419, 0.51718424, 0.74814454, 0.04825398])
```

We can calculate the average and standard deviation of our estimate:

```
print("The mean of the folds is  :", round(Rcross.mean(),3))
print("The standard deviation is :" , round(Rcross.std(),3))
```

```
The mean of the folds is  : 0.522
The standard deviation is : 0.291
```

We can use mean_squared_error as a score by setting the parameter **scoring** metric to **neg_mean_squared_error**.

```
-1 * cross_val_score(lre,x_data[['horsepower']], y_data,cv=4,scoring='neg_mean_squared_error')
```

```
array([20251357.7835463 , 43743920.05390439, 12525158.34507633,
       17564549.69976653])
```

We can also use the function **cross_val_predict** to predict the output.
The function splits up the data into the specified number of folds, using one fold to get a prediction while the rest of the folds are used as training data.

First import the function:

```
from sklearn.model_selection import cross_val_predict
```

We input the model object, the feature in this case **'horsepower'** , the target data **y_data**.
Then, We can produce an output:

```
yhat = cross_val_predict(lre,x_data[['horsepower']], y_data,cv=4)
yhat[0:5]
```

```
array([14142.23793549, 14142.23793549, 20815.3029844 , 12745.549902  ,
       14762.9881726 ])
```

Overfitting and Model Selection

It turns out that the test data sometimes referred to as the out of sample data is a much better measure of how well your model performs in the real world.
One reason for this is overfitting; let's go over some examples so we can explore overfitting.

Let's create Multiple linear regression objects and train the model using **'horsepower'**, **'curb-weight'**, **'engine-size'** and **'highway-mpg'** as features.

```
lr = LinearRegression()
lr.fit(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_train)
```

```
LinearRegression()
```

Prediction using training data:

```
yhat_train = lr.predict(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
yhat_train[0:5]
```

```
array([11927.25153792, 11236.70125955,  6436.82274615, 21891.09897761,
       16669.10119352])
```

Prediction using test data:

```
yhat_test = lr.predict(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
yhat_test[0:5]
```

```
array([11349.48964574,  5914.6130239 , 11243.35261505,  6661.95904136,
       15555.94734434])
```

Let's perform some **model evaluation** using our training and testing data separately.

First we import the seaborn and matplotlib library for plotting.

```
import matplotlib.pyplot as plt
import seaborn as sns
```

Let's examine the distribution of the predicted values of the training data.

```
Title = 'Distribution Plot of Predicted vs Actual Values (Train Data)'
DistributionPlot(y_train, yhat_train, "Actual Values (Train)", "Predicted Values (Train)", Title)
```

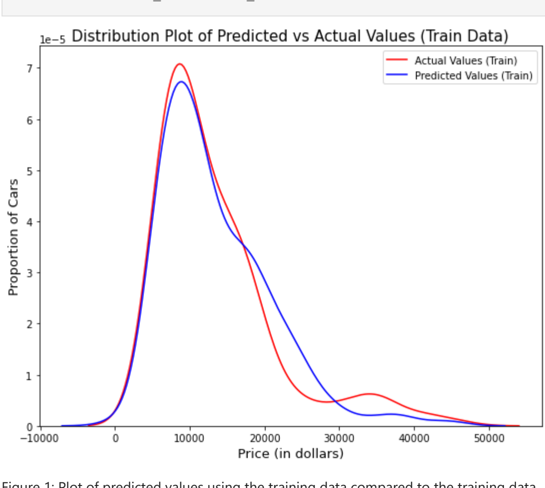


Figure 1: Plot of predicted values using the training data compared to the training data.

So far the model seems to be doing well in learning from the training dataset.
But what happens when the model encounters new data from the testing dataset?

When the model generates new values from the test data, we see the distribution of the predicted values is much different from the actual target values.

```
Title='Distribution Plot of Predicted vs Actual Values (Test Data)'
DistributionPlot(y_test,yhat_test,"Actual Values (Test)","Predicted Values (Test)",Title)
```

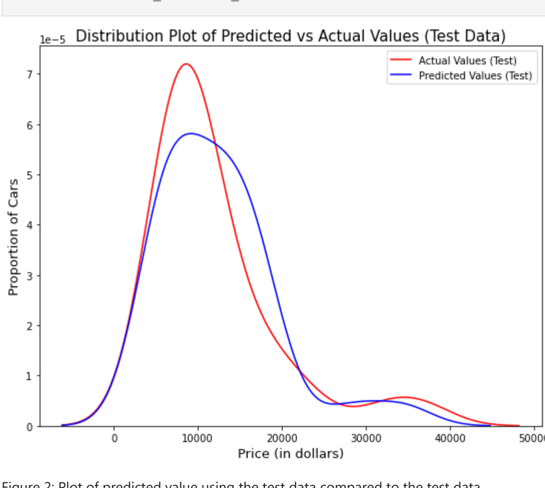


Figure 2: Plot of predicted value using the test data compared to the test data.

Comparing Figure 1 and Figure 2; it is evident that **distribution of the test data in Figure 1 is much better** at fitting the data.
This difference in Figure 2 is apparent where the ranges are from 5000 to 15,000. This is where the distribution shape is exceptionally different. Let's see if polynomial regression also exhibits a drop in the prediction accuracy when analysing the test dataset.

```
from sklearn.preprocessing import PolynomialFeatures
```

Overfitting

Overfitting occurs when the model fits the noise, not the underlying process. Therefore when testing our model using the test-set, the model does not perform as well as it is modelling noise, not the underlying process that generated the relationship.

Let's create a degree 5 polynomial model.

Let's use 45% of the data for testing and the rest for training:

```
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.45, random_state=0)
```

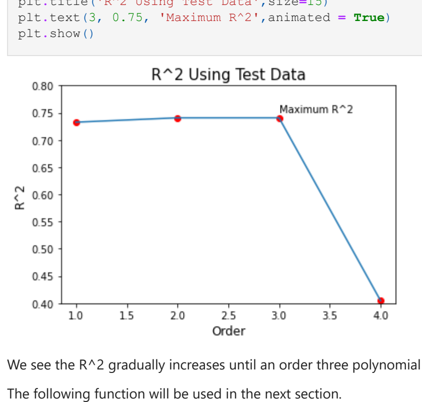
```
In [167…  poly.score(x_test_pr, y_test)
```

```
Out[167…  -29.815481910638443
```

We see the R^2 for the training data is 0.5567 while the R^2 on the test data was -29.87.
The lower the R^2, the worse the model; a Negative R^2 is a sign of Overfitting.

Let's see how the R^2 changes on test data for different Order Polynomials and Plot the results:

```
In [168…
Rsqu_test = []

order = [1, 2, 3, 4]
for n in order:
    pr = PolynomialFeatures(degree=n)

    x_train_pr = pr.fit_transform(x_train[['horsepower']])
    x_test_pr = pr.fit_transform(x_test[['horsepower']])

    lr.fit(x_train_pr, y_train)

    Rsqu_test.append(lr.score(x_test_pr, y_test))

plt.plot(order, Rsqu_test,"ro")
plt.plot(order, Rsqu_test)
plt.xlabel('Order',size=12)
plt.ylabel('R^2',size=12)
plt.ylim(0.40,0.80)
plt.title('R^2 Using Test Data',size=15)
plt.text(3, 0.75, 'Maximum R^2',animated = True)
plt.show()
```



We see the R^2 gradually increases until an order three polynomial is used. Then the R^2 dramatically decreases at four.

The following function will be used in the next section.

```
In [169…
def func(Order, Test_Data):
    x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=Test_Data, random_state=0)

    # pr   : polynomial transformation object
    # poly : linear regression object

    pr = PolynomialFeatures(degree = Order)
    x_train_pr = pr.fit_transform(x_train[['horsepower']])
    x_test_pr = pr.fit_transform(x_test[['horsepower']])

    poly = LinearRegression()
    poly.fit(x_train_pr,y_train)

    #User defined function for plotting
    PollyPlot(x_train[['horsepower']], x_test[['horsepower']], y_train,y_test, poly, pr, "Horsepower")

    print(f"R^2 Value of Training Data : {round(poly.score(x_train_pr, y_train),3)}")
    print(f"R^2 Value of Testing Data  : {round(poly.score(x_test_pr, y_test),3)}")
```

The following interface allows us to experiment with different polynomial orders and different amounts of data.

```
In [170…  interact(func, Order=(0, 6, 1), Test_Data=(0.10, 0.90, 0.05));
```

We can Perform polynomial transformations with more than one feature.

First, Create a "PolynomialFeatures" object "pr1" of degree two. And then, splitting the dataset into training and testng data.

```
In [171…  pr1=PolynomialFeatures(degree=2)
```

Transforming the training and testing samples for the features 'horsepower', 'curb-weight', 'engine-size' and 'highway-mpg'.

```
In [172…
x_train_pr1=pr.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])

x_test_pr1=pr.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
```

Now, How many dimensions does the new feature have?

```
In [173…  x_train_pr1.shape
```

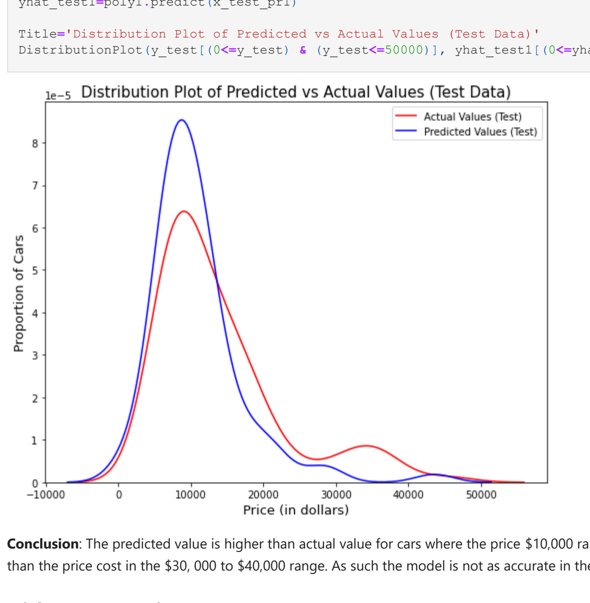
```
Out[173…  (110, 70)
```

Creating a linear regression model "poly1" and training the object.

```
In [174…  poly1 = LinearRegression().fit(x_train_pr1,y_train)
```

Predicting an output on the polynomial features, then using the function "DistributionPlot" to display the distribution of the predicted output vs the test data.

```
In [175…
yhat_test1=poly1.predict(x_test_pr1)

Title='Distribution Plot of Predicted vs Actual Values (Test Data)'
DistributionPlot(y_test[(0<=y_test) & (y_test<=50000)], yhat_test1[(0<=yhat_test1) & (yhat_test1 <= 50000)], "A
```



Conclusion: The predicted value is higher than actual value for cars where the price $10,000 range, conversely the predicted price is lower than the price cost in the $30, 000 to $40,000 range. As such the model is not as accurate in these ranges.

Ridge regression

Ridge Regression is usually used when there is a high correlation between independent variables. This is because, in the case of multi collinear data, the least square estimates give unbiased values. This is a powerful regression method where the model is less susceptible to overfitting.

Let's perform a degree two polynomial transformation on our data.

```
In [176…
pr=PolynomialFeatures(degree=2)
x_train_pr=pr.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg','normalized-loss
x_test_pr=pr.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg','normalized-losses
```

Let's import **Ridge** from the module **linear models**.

```
In [177…  from sklearn.linear_model import Ridge
```

Let's create a Ridge regression object, setting the Regularization Parameter "alpha" to 0.1 .

```
In [178…  RigeModel=Ridge(alpha=0.1)
```

Now, fit the model using the method **fit**.

```
In [193…  RigeModel.fit(x_train_pr, y_train);
```

Similarly, we can obtain a prediction:

```
In [180…  yhat = RigeModel.predict(x_test_pr)
```

Let's compare the first four predicted samples to our test set.

```
In [181…
print('Predicted :', yhat[0:4])
print('Test-set  :', y_test[0:4].values)
```

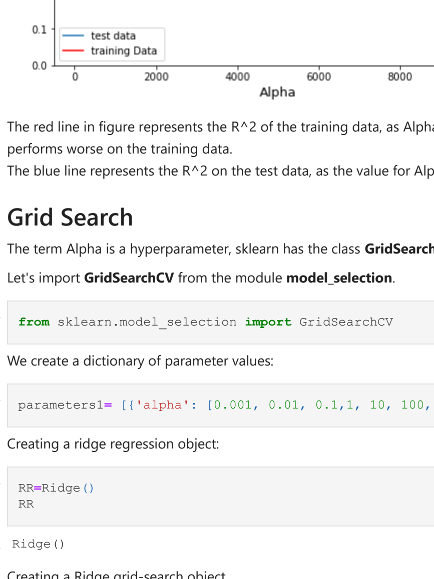
```
Predicted : [ 6569.10080596  9595.9695303  20834.19869602 19347.43557722]
Test-set  : [ 6295. 10698. 13860. 13499.]
```

Now, We will select the value of Alpha that minimizes the test error.

```
In [182…
Rsqu_test = []
Rsqu_train = []

ALPHA = 10 * np.array(range(0,1000))
for a in ALPHA:
    RigeModel = Ridge(alpha=a)
    RigeModel.fit(x_train_pr, y_train)
    Rsqu_test.append(RigeModel.score(x_test_pr, y_test))
    Rsqu_train.append(RigeModel.score(x_train_pr, y_train))
```

We can plot out the value of R^2 for different Alphas.

```
In [183…
plt.figure(figsize=(8, 6))

plt.plot(ALPHA,Rsqu_test, label='test data')
plt.plot(ALPHA,Rsqu_train, 'r', label='training Data ')

plt.title("R^2 for Different Alpha",size=15)
plt.xlabel('Alpha',size=13)
plt.ylabel('R^2',size=13)

plt.ylim(0,)
plt.legend()
plt.show()
```



The red line in figure represents the R^2 of the training data, as Alpha increases the R^2 decreases; therefore as Alpha increases the model performs worse on the training data.
The blue line represents the R^2 on the test data, as the value for Alpha increases the R^2 increases.

Grid Search

The term Alpha is a hyperparameter, sklearn has the class **GridSearchCV** to make the process of finding the best hyperparameter simpler.

Let's import **GridSearchCV** from the module **model_selection**.

```
In [184…  from sklearn.model_selection import GridSearchCV
```

We create a dictionary of parameter values:

```
In [185…  parameters1= [{'alpha': [0.001, 0.01, 0.1,1, 10, 100, 1000, 10000, 100000],'normalize':[True,False]}]
```

Creating a ridge regression object:

```
In [186…
RR=Ridge()
RR
```

```
Out[186…  Ridge()
```

Creating a Ridge grid-search object.

```
In [187…  Grid1 = GridSearchCV(RR, parameters1,cv=4)
```

Fit the model

```
In [188…  Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg','normalized-losses','symboling']],
```

```
Out[188…  GridSearchCV(cv=4, estimator=Ridge(),
                       param_grid=[{'alpha': [0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000,
                                              100000],
                                    'normalize': [True, False]}])
```

The object finds the best parameter values on the validation data. We can obtain the estimator with the best parameters and assign it to the variable BestRR as follows:

```
In [189…
BestRR=Grid1.best_estimator_
BestRR
```

```
Out[189…  Ridge(alpha=0.1, normalize=True)
```

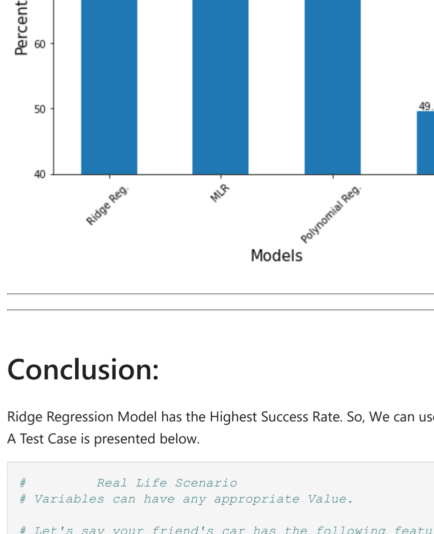
We now test our model on the test data

```
In [190…
# R^2 Value
r2 = BestRR.score(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg','normalized-losses','symbol
evalDict['Ridge Reg.'] = r2*100
r2
```

```
Out[190…  0.841673905432361
```

Plotting a Model Success Plot

```
In [191…
def plotSuccess():
    s = pd.Series(evalDict)
    s = s.sort_values(ascending=False)
    plt.figure(figsize=(8,6))

    ax = s.plot(kind='bar')
    for p in ax.patches:
        ax.annotate(str(round(p.get_height(),2)), (p.get_x() * 1.005, p.get_height() * 1.005))
    plt.ylim([40.0, 90.0])
    plt.xlabel('Models',size=15)
    plt.xticks(rotation=45)
    plt.ylabel('Percentage',size=15)
    plt.title('Success Rate of Models',size=20)
    plt.show()

plotSuccess()
```



Conclusion:

Ridge Regression Model has the Highest Success Rate. So, We can use it to predict the prices of Cars successfully.
A Test Case is presented below.

```
In [192…
#         Real Life Scenario
# Variables can have any appropriate Value.

# Let's say your friend's car has the following features.
horsepower = 110.0
curb_weight = 2550.0
engine_size = 120.0
highway_mpg = 25.0
normalized_losses = 110.0
symboling = 1.0

testCase = [[horsepower, curb_weight, engine_size, highway_mpg ,normalized_losses ,symboling]]
TestDf = pd.DataFrame(testCase, columns = ['horsepower', 'curb-weight', 'engine-size', 'highway-mpg','normalize
SellingPrice = BestRR.predict(TestDf)

# In Indian Currency  (1 Dollar = 74.19 Rupees).
SellingPrice *= 74.19

print("Selling Price of Your Friend's Car is {:,.2f} Rupees.".format(SellingPrice[0]))
```

```
Selling Price of Your Friend's Car is 979,056.43 Rupees.
```
