

Automobile Data Analysis

February 12, 2024

Lab Component By Dhesika

1 Importing Dataset

Estimated time needed: **15** minutes

1.1 Objectives

- Acquire data in various ways
- Obtain insights from data with Pandas library

Table of Contents

Data Acquisition

Basic Insights from the Data set

2 Data Acquisition

There are various formats for a data set: .csv, .json, .xlsx etc. The data set can be stored in different places, on your local machine or sometimes online.

In our case, the Automobile Data set is an online source, and it is in a CSV (comma separated value) format. Let's use this data set as an example to practice data reading.

Data source: <https://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.data>

Data type: csv

The Pandas Library is a useful tool that enables us to read various datasets into a data frame; our Jupyter notebook platforms have a built-in Pandas Library so that all we need to do is import Pandas without installing.

```
[1]: import pandas as pd
import numpy as np
```

Read Data

We utilize the `pandas.read_csv()` function for reading CSV files.

```
[2]: file_name="auto.csv"
```

Utilize the Pandas method `read_csv()` to load the data into a dataframe.

```
[3]: df = pd.read_csv(file_name)
```

After reading the data set, we can use the `data_frame.head(n)` method to check the top `n` rows of the data frame, where `n` is an integer. Contrary to `data_frame.head(n)`, `data_frame.tail(n)` will show you the bottom `n` rows of the data frame.

```
[4]: # show the first 5 rows using dataframe.head() method
print("The first 5 rows of the dataframe")
df.head(5)
```

The first 5 rows of the dataframe

```
[4]:      3      ?  alfa-romero  gas  std   two  convertible  rwd  front  88.6  ...  \
0  3      ?  alfa-romero  gas  std   two  convertible  rwd  front  88.6  ...
1  1      ?  alfa-romero  gas  std   two   hatchback  rwd  front  94.5  ...
2  2  164          audi  gas  std   four          sedan  fwd  front  99.8  ...
3  2  164          audi  gas  std   four          sedan  4wd  front  99.4  ...
4  2      ?          audi  gas  std   two          sedan  fwd  front  99.8  ...
```

```
      130  mpfi  3.47  2.68   9.0  111  5000  21  27  13495
0  130  mpfi  3.47  2.68   9.0  111  5000  21  27  16500
1  152  mpfi  2.68  3.47   9.0  154  5000  19  26  16500
2  109  mpfi  3.19  3.40  10.0  102  5500  24  30  13950
3  136  mpfi  3.19  3.40   8.0  115  5500  18  22  17450
4  136  mpfi  3.19  3.40   8.5  110  5500  19  25  15250
```

[5 rows x 26 columns]

```
[5]: print("The last 10 rows of the dataframe\n")
df.tail(10)
```

The last 10 rows of the dataframe

```
[5]:      3      ?  alfa-romero      gas      std   two  convertible  rwd  front  88.6  \
194 -1   74          volvo      gas      std   four          wagon  rwd  front  104.3
195 -2  103          volvo      gas      std   four          sedan  rwd  front  104.3
196 -1   74          volvo      gas      std   four          wagon  rwd  front  104.3
197 -2  103          volvo      gas  turbo   four          sedan  rwd  front  104.3
198 -1   74          volvo      gas  turbo   four          wagon  rwd  front  104.3
199 -1   95          volvo      gas      std   four          sedan  rwd  front  109.1
200 -1   95          volvo      gas  turbo   four          sedan  rwd  front  109.1
201 -1   95          volvo      gas      std   four          sedan  rwd  front  109.1
202 -1   95          volvo  diesel  turbo   four          sedan  rwd  front  109.1
203 -1   95          volvo      gas  turbo   four          sedan  rwd  front  109.1

      ...  130  mpfi  3.47  2.68   9.0  111  5000  21  27  13495
194  ...  141  mpfi  3.78  3.15   9.5  114  5400  23  28  13415
195  ...  141  mpfi  3.78  3.15   9.5  114  5400  24  28  15985
```

196	...	141	mpfi	3.78	3.15	9.5	114	5400	24	28	16515
197	...	130	mpfi	3.62	3.15	7.5	162	5100	17	22	18420
198	...	130	mpfi	3.62	3.15	7.5	162	5100	17	22	18950
199	...	141	mpfi	3.78	3.15	9.5	114	5400	23	28	16845
200	...	141	mpfi	3.78	3.15	8.7	160	5300	19	25	19045
201	...	173	mpfi	3.58	2.87	8.8	134	5500	18	23	21485
202	...	145	idi	3.01	3.40	23.0	106	4800	26	27	22470
203	...	141	mpfi	3.78	3.15	9.5	114	5400	19	25	22625

[10 rows x 26 columns]

Add Headers

Take a look at the data set. Pandas automatically set the header with an integer starting from 0.

To better describe the data, you can introduce a header. This information is available at: <https://archive.ics.uci.edu/ml/datasets/Automobile>.

Thus, you have to add headers manually.

First, create a list “headers” that include all column names in order. Then, use `dataframe.columns = headers` to replace the headers with the list you created.

```
[6]: # create headers list
headers = ["symboling", "normalized-losses", "make", "fuel-type", "aspiration",
↪ "num-of-doors", "body-style",
           "drive-wheels", "engine-location", "wheel-base",
↪ "length", "width", "height", "curb-weight", "engine-type",
           "num-of-cylinders",
↪ "engine-size", "fuel-system", "bore", "stroke", "compression-ratio", "horsepower",
           "peak-rpm", "city-mpg", "highway-mpg", "price"]
print("headers\n", headers)
```

headers

```
['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration', 'num-of-
doors', 'body-style', 'drive-wheels', 'engine-location', 'wheel-base', 'length',
'width', 'height', 'curb-weight', 'engine-type', 'num-of-cylinders', 'engine-
size', 'fuel-system', 'bore', 'stroke', 'compression-ratio', 'horsepower',
'peak-rpm', 'city-mpg', 'highway-mpg', 'price']
```

Replace headers and recheck our data frame:

```
[7]: df.columns = headers
df.columns
```

```
[7]: Index(['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration',
           'num-of-doors', 'body-style', 'drive-wheels', 'engine-location',
           'wheel-base', 'length', 'width', 'height', 'curb-weight', 'engine-type',
           'num-of-cylinders', 'engine-size', 'fuel-system', 'bore', 'stroke',
           'compression-ratio', 'horsepower', 'peak-rpm', 'city-mpg',
```

```
'highway-mpg', 'price'],
dtype='object')
```

You can also see the first 10 entries of the updated data frame and note that the headers are updated.

```
[8]: df.head(10)
```

```
[8]:   symboling normalized-losses      make fuel-type aspiration num-of-doors \
0         3             ?  alfa-romero    gas      std         two
1         1             ?  alfa-romero    gas      std         two
2         2          164      audi    gas      std         four
3         2          164      audi    gas      std         four
4         2             ?      audi    gas      std         two
5         1          158      audi    gas      std         four
6         1             ?      audi    gas      std         four
7         1          158      audi    gas    turbo         four
8         0             ?      audi    gas    turbo         two
9         2          192      bmw    gas      std         two
```

```
   body-style drive-wheels engine-location  wheel-base  ...  engine-size  \
0  convertible         rwd         front      88.6  ...      130
1   hatchback         rwd         front      94.5  ...      152
2         sedan         fwd         front      99.8  ...      109
3         sedan         4wd         front      99.4  ...      136
4         sedan         fwd         front      99.8  ...      136
5         sedan         fwd         front     105.8  ...      136
6         wagon         fwd         front     105.8  ...      136
7         sedan         fwd         front     105.8  ...      131
8   hatchback         4wd         front      99.5  ...      131
9         sedan         rwd         front     101.2  ...      108
```

```
   fuel-system  bore  stroke  compression-ratio  horsepower  peak-rpm  city-mpg  \
0         mpfi  3.47   2.68             9.0         111      5000      21
1         mpfi  2.68   3.47             9.0         154      5000      19
2         mpfi  3.19   3.40            10.0         102      5500      24
3         mpfi  3.19   3.40             8.0         115      5500      18
4         mpfi  3.19   3.40             8.5         110      5500      19
5         mpfi  3.19   3.40             8.5         110      5500      19
6         mpfi  3.19   3.40             8.5         110      5500      19
7         mpfi  3.13   3.40             8.3         140      5500      17
8         mpfi  3.13   3.40             7.0         160      5500      16
9         mpfi  3.50   2.80             8.8         101      5800      23
```

```
   highway-mpg  price
0           27  16500
1           26  16500
2           30  13950
```

```

3      22  17450
4      25  15250
5      25  17710
6      25  18920
7      20  23875
8      22    ?
9      29  16430

```

[10 rows x 26 columns]

Now, we need to replace the “?” symbol with NaN so the dropna() can remove the missing values:

```
[9]: df1=df.replace('?',np.NaN)
```

You can drop missing values along the column “price” as follows:

```
[10]: df=df1.dropna(subset=["price"], axis=0)
df.head(20)
```

```
[10]:      symboling normalized-losses      make fuel-type aspiration \
0          3             NaN  alfa-romero    gas      std
1          1             NaN  alfa-romero    gas      std
2          2            164      audi      gas      std
3          2            164      audi      gas      std
4          2             NaN      audi      gas      std
5          1            158      audi      gas      std
6          1             NaN      audi      gas      std
7          1            158      audi      gas    turbo
9          2            192      bmw      gas      std
10         0            192      bmw      gas      std
11         0            188      bmw      gas      std
12         0            188      bmw      gas      std
13         1             NaN      bmw      gas      std
14         0             NaN      bmw      gas      std
15         0             NaN      bmw      gas      std
16         0             NaN      bmw      gas      std
17         2            121  chevrolet      gas      std
18         1            98   chevrolet      gas      std
19         0            81   chevrolet      gas      std
20        1            118     dodge      gas      std

      num-of-doors  body-style drive-wheels engine-location  wheel-base  ... \
0          two  convertible      rwd      front      88.6  ...
1          two   hatchback      rwd      front      94.5  ...
2          four    sedan      fwd      front      99.8  ...
3          four    sedan      4wd      front      99.4  ...
4          two    sedan      fwd      front      99.8  ...
5          four    sedan      fwd      front     105.8  ...

```

6	four	wagon	fwd	front	105.8	...
7	four	sedan	fwd	front	105.8	...
9	two	sedan	rwd	front	101.2	...
10	four	sedan	rwd	front	101.2	...
11	two	sedan	rwd	front	101.2	...
12	four	sedan	rwd	front	101.2	...
13	four	sedan	rwd	front	103.5	...
14	four	sedan	rwd	front	103.5	...
15	two	sedan	rwd	front	103.5	...
16	four	sedan	rwd	front	110.0	...
17	two	hatchback	fwd	front	88.4	...
18	two	hatchback	fwd	front	94.5	...
19	four	sedan	fwd	front	94.5	...
20	two	hatchback	fwd	front	93.7	...

	engine-size	fuel-system	bore	stroke	compression-ratio	horsepower	\
0	130	mpfi	3.47	2.68	9.00	111	
1	152	mpfi	2.68	3.47	9.00	154	
2	109	mpfi	3.19	3.40	10.00	102	
3	136	mpfi	3.19	3.40	8.00	115	
4	136	mpfi	3.19	3.40	8.50	110	
5	136	mpfi	3.19	3.40	8.50	110	
6	136	mpfi	3.19	3.40	8.50	110	
7	131	mpfi	3.13	3.40	8.30	140	
9	108	mpfi	3.50	2.80	8.80	101	
10	108	mpfi	3.50	2.80	8.80	101	
11	164	mpfi	3.31	3.19	9.00	121	
12	164	mpfi	3.31	3.19	9.00	121	
13	164	mpfi	3.31	3.19	9.00	121	
14	209	mpfi	3.62	3.39	8.00	182	
15	209	mpfi	3.62	3.39	8.00	182	
16	209	mpfi	3.62	3.39	8.00	182	
17	61	2bbl	2.91	3.03	9.50	48	
18	90	2bbl	3.03	3.11	9.60	70	
19	90	2bbl	3.03	3.11	9.60	70	
20	90	2bbl	2.97	3.23	9.41	68	

	peak-rpm	city-mpg	highway-mpg	price
0	5000	21	27	16500
1	5000	19	26	16500
2	5500	24	30	13950
3	5500	18	22	17450
4	5500	19	25	15250
5	5500	19	25	17710
6	5500	19	25	18920
7	5500	17	20	23875
9	5800	23	29	16430

10	5800	23	29	16925
11	4250	21	28	20970
12	4250	21	28	21105
13	4250	20	25	24565
14	5400	16	22	30760
15	5400	16	22	41315
16	5400	15	20	36880
17	5100	47	53	5151
18	5400	38	43	6295
19	5400	38	43	6575
20	5500	37	41	5572

[20 rows x 26 columns]

Here, axis=0 means that the contents along the entire row will be dropped wherever the entity 'price' is found to be NaN

Now, you have successfully read the raw data set and added the correct headers into the data frame.

Find the name of the columns of the dataframe.

```
[11]: print(df.columns)
```

```
Index(['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration',
      'num-of-doors', 'body-style', 'drive-wheels', 'engine-location',
      'wheel-base', 'length', 'width', 'height', 'curb-weight', 'engine-type',
      'num-of-cylinders', 'engine-size', 'fuel-system', 'bore', 'stroke',
      'compression-ratio', 'horsepower', 'peak-rpm', 'city-mpg',
      'highway-mpg', 'price'],
      dtype='object')
```

Save Dataset

Correspondingly, Pandas enables you to save the data set to CSV. By using the dataframe.to_csv() method, you can add the file path and name along with quotation marks in the brackets.

For example, if you save the data frame df as automobile.csv to your local machine, you may use the syntax below, where index = False means the row names will not be written.

```
[12]: df.to_csv("usedCars.csv", index=False)
```

You can also read and save other file formats. You can use similar functions like **pd.read_csv()** and **df.to_csv()** for other data formats. The functions are listed in the following table:

Read/Save Other Data Formats

Data Formate	Read	Save
csv	pd.read_csv()	df.to_csv()
json	pd.read_json()	df.to_json()
excel	pd.read_excel()	df.to_excel()
hdf	pd.read_hdf()	df.to_hdf()

Data Formate	Read	Save
sql	<code>pd.read_sql()</code>	<code>df.to_sql()</code>
...

3 Basic Insights from the Data set

After reading data into Pandas dataframe, it is time for you to explore the data set.

There are several ways to obtain essential insights of the data to help you better understand it.

Data Types

Data has a variety of types.

The main types stored in Pandas data frames are object, float, int, bool and datetime64. In order to better learn about each attribute, you should always know the data type of each column. In Pandas:

```
[13]: df.dtypes
```

```
[13]: 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
```

Returns a series with the data type of each column.


```
[14]: # check the data type of data frame "df" by .dtypes
print(df.dtypes)
```

```
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 shown above, you can clearly to see that the data type of “symboling” and “curb-weight” are int64, “normalized-losses” is object, and “wheel-base” is float64, etc.

These data types can be changed; you will learn how to accomplish this in a later module.

Describe

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

```
[15]: df.describe()
```

```
[15]:
```

	symboling	wheel-base	length	width	height	\
count	200.000000	200.000000	200.000000	200.000000	200.000000	
mean	0.830000	98.848000	174.228000	65.898000	53.791500	
std	1.248557	6.038261	12.347132	2.102904	2.428449	
min	-2.000000	86.600000	141.100000	60.300000	47.800000	
25%	0.000000	94.500000	166.675000	64.175000	52.000000	
50%	1.000000	97.000000	173.200000	65.500000	54.100000	

75%	2.000000	102.400000	183.500000	66.675000	55.525000
max	3.000000	120.900000	208.100000	72.000000	59.800000

	curb-weight	engine-size	compression-ratio	city-mpg	highway-mpg
count	200.000000	200.000000	200.000000	200.000000	200.000000
mean	2555.705000	126.860000	10.170100	25.200000	30.705000
std	518.594552	41.650501	4.014163	6.432487	6.827227
min	1488.000000	61.000000	7.000000	13.000000	16.000000
25%	2163.000000	97.750000	8.575000	19.000000	25.000000
50%	2414.000000	119.500000	9.000000	24.000000	30.000000
75%	2928.250000	142.000000	9.400000	30.000000	34.000000
max	4066.000000	326.000000	23.000000	49.000000	54.000000

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

This shows the statistical summary of all numeric-typed (int, float) columns. 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. However, what if you would also like to check all the columns including those that are of type object? You can add an argument include = “all” inside the bracket. Try it again.

```
[16]: # describe all the columns in "df"
df.describe(include = "all")
```

```
[16]:
```

	symboling	normalized-losses	make	fuel-type	aspiration	\
count	200.000000	164	200	200	200	
unique	NaN	51	22	2	2	
top	NaN	161	toyota	gas	std	
freq	NaN	11	32	180	164	
mean	0.830000	NaN	NaN	NaN	NaN	
std	1.248557	NaN	NaN	NaN	NaN	
min	-2.000000	NaN	NaN	NaN	NaN	
25%	0.000000	NaN	NaN	NaN	NaN	
50%	1.000000	NaN	NaN	NaN	NaN	
75%	2.000000	NaN	NaN	NaN	NaN	
max	3.000000	NaN	NaN	NaN	NaN	

	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	\
count	198	200	200	200	200.000000	...	
unique	2	5	3	2	NaN	...	
top	four	sedan	fwd	front	NaN	...	
freq	113	94	118	197	NaN	...	
mean	NaN	NaN	NaN	NaN	98.848000	...	
std	NaN	NaN	NaN	NaN	6.038261	...	
min	NaN	NaN	NaN	NaN	86.600000	...	
25%	NaN	NaN	NaN	NaN	94.500000	...	

50%	NaN	NaN	NaN	NaN	97.000000	...
75%	NaN	NaN	NaN	NaN	102.400000	...
max	NaN	NaN	NaN	NaN	120.900000	...

	engine-size	fuel-system	bore	stroke	compression-ratio	horsepower \
count	200.000000	200	196	196	200.000000	198
unique	NaN	8	38	36	NaN	58
top	NaN	mpfi	3.62	3.40	NaN	68
freq	NaN	91	23	19	NaN	19
mean	126.860000	NaN	NaN	NaN	10.170100	NaN
std	41.650501	NaN	NaN	NaN	4.014163	NaN
min	61.000000	NaN	NaN	NaN	7.000000	NaN
25%	97.750000	NaN	NaN	NaN	8.575000	NaN
50%	119.500000	NaN	NaN	NaN	9.000000	NaN
75%	142.000000	NaN	NaN	NaN	9.400000	NaN
max	326.000000	NaN	NaN	NaN	23.000000	NaN

	peak-rpm	city-mpg	highway-mpg	price
count	198	200.000000	200.000000	200
unique	22	NaN	NaN	185
top	5500	NaN	NaN	16500
freq	36	NaN	NaN	2
mean	NaN	25.200000	30.705000	NaN
std	NaN	6.432487	6.827227	NaN
min	NaN	13.000000	16.000000	NaN
25%	NaN	19.000000	25.000000	NaN
50%	NaN	24.000000	30.000000	NaN
75%	NaN	30.000000	34.000000	NaN
max	NaN	49.000000	54.000000	NaN

[11 rows x 26 columns]

Now it provides the statistical summary of all the columns, including object-typed attributes.

YOu can now see how many unique values there, which one is the top value, and the frequency of the top value in the object-typed columns.

Some values in the table above show “NaN”. Those numbers are not available regarding a particular column type.

You can select the columns of a dataframe by indicating the name of each column. For example, you can select the three columns as follows:

```
dataframe[['column 1','column 2', 'column 3']]
```

Where “column” is the name of the column, you can apply the method “.describe()” to get the statistics of those columns as follows:

```
dataframe[['column 1','column 2', 'column 3'] ].describe()
```

Apply the method to “.describe()” to the columns ‘length’ and ‘compression-ratio’.

```
[17]: df[['length', 'compression-ratio']].describe()
```

```
[17]:
```

	length	compression-ratio
count	200.000000	200.000000
mean	174.228000	10.170100
std	12.347132	4.014163
min	141.100000	7.000000
25%	166.675000	8.575000
50%	173.200000	9.000000
75%	183.500000	9.400000
max	208.100000	23.000000

Info

You can also use another method to check your data set:

`dataframe.info()` It provides a concise summary of your data frame.

This method prints information about a data frame including the index dtype and columns, non-null values and memory usage.

```
[18]: # look at the info of "df"
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 200 entries, 0 to 203
Data columns (total 26 columns):
#   Column                Non-Null Count  Dtype
---  -
0   symboling              200 non-null    int64
1   normalized-losses      164 non-null    object
2   make                   200 non-null    object
3   fuel-type              200 non-null    object
4   aspiration              200 non-null    object
5   num-of-doors            198 non-null    object
6   body-style              200 non-null    object
7   drive-wheels            200 non-null    object
8   engine-location         200 non-null    object
9   wheel-base              200 non-null    float64
10  length                  200 non-null    float64
11  width                   200 non-null    float64
12  height                  200 non-null    float64
13  curb-weight             200 non-null    int64
14  engine-type             200 non-null    object
15  num-of-cylinders        200 non-null    object
16  engine-size             200 non-null    int64
17  fuel-system             200 non-null    object
18  bore                    196 non-null    object
19  stroke                  196 non-null    object
20  compression-ratio       200 non-null    float64
```

```

21 horsepower      198 non-null    object
22 peak-rpm        198 non-null    object
23 city-mpg        200 non-null    int64
24 highway-mpg     200 non-null    int64
25 price           200 non-null    object
dtypes: float64(5), int64(5), object(16)
memory usage: 42.2+ KB

```

4 Data Wrangling

Estimated time needed: **30** minutes

4.1 Objectives

- Handle missing values
- Correct data formatting
- Standardize and normalize data

Table of Contents

Identify and handle missing values

Identify missing values

Deal with missing values

Correct data format

Data standardization

Data normalization (centering/scaling)

Binning

Indicator variable

What is the purpose of data wrangling?

You use data wrangling to convert data from an initial format to a format that may be better for analysis.

What is the fuel consumption (L/100k) rate for the diesel car?

Import data

You can find the “Automobile Dataset” from the following link:
<https://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.data>. You will be using this data set throughout this course.

Import pandas

```

[19]: #install specific version of libraries used in lab
      #! mamba install pandas==1.3.3
      #! mamba install numpy=1.21.2

```

```
[20]: import pandas as pd
import matplotlib.pyplot as plt
```

Reading the dataset from the URL and adding the related headers

This dataset was hosted on IBM Cloud object. Click [HERE](#) for free storage.

The functions below will download the dataset into your browser:

```
[21]: '''from pyodide.http import pyfetch

async def download(url, filename):
    response = await pyfetch(url)
    if response.status == 200:
        with open(filename, "wb") as f:
            f.write(await response.bytes())'''
```

```
[21]: 'from pyodide.http import pyfetch\n\nasync def download(url, filename):\n    response = await pyfetch(url)\n    if response.status == 200:\n        with\n    open(filename, "wb") as f:\n        f.write(await response.bytes())'
```

First, assign the URL of the data set to “filepath”.

```
[22]: #file_path="https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/\n↪IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/auto.csv"
```

To obtain the dataset, utilize the download() function as defined above:

```
[23]: #await download(file_path, "usedcars.csv")
file_name="usedCars.csv"
```

Then, create a Python list headers containing name of headers.

```
[24]: '''headers = ["symboling", "normalized-losses", "make", "fuel-type", "aspiration", \
↪ "num-of-doors", "body-style",
    "drive-wheels", "engine-location", "wheel-base", \
↪ "length", "width", "height", "curb-weight", "engine-type",
    "num-of-cylinders", \
↪ "engine-size", "fuel-system", "bore", "stroke", "compression-ratio", "horsepower",
    "peak-rpm", "city-mpg", "highway-mpg", "price"]'''
```

```
[24]: 'headers = ["symboling", "normalized-losses", "make", "fuel-type", "aspiration",
"num-of-doors", "body-style", \n        "drive-wheels", "engine-location", "wheel-
base", "length", "width", "height", "curb-weight", "engine-type", \n        "num-of-
cylinders", "engine-size", "fuel-system", "bore", "stroke", "compression-
ratio", "horsepower", \n        "peak-rpm", "city-mpg", "highway-mpg", "price"]'
```

Use the Pandas method read_csv() to load the data from the web address. Set the parameter “names” equal to the Python list “headers”.

```
[25]: df = pd.read_csv('usedCars.csv')
```

```
[26]: #filepath = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/
      ↪IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/auto.csv"
      #df = pd.read_csv(filepath, header=headers)    # Utilize the same header list
      ↪defined above
```

Use the method head() to display the first five rows of the dataframe.

```
[27]: # To see what the data set looks like, we'll use the head() method.
      df.head()
```

```
[27]:
```

	symboling	normalized-losses	make	fuel-type	aspiration	\
0	3	NaN	alfa-romero	gas	std	
1	1	NaN	alfa-romero	gas	std	
2	2	164.0	audi	gas	std	
3	2	164.0	audi	gas	std	
4	2	NaN	audi	gas	std	

	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	\
0	two	convertible	rwd	front	88.6	...	
1	two	hatchback	rwd	front	94.5	...	
2	four	sedan	fwd	front	99.8	...	
3	four	sedan	4wd	front	99.4	...	
4	two	sedan	fwd	front	99.8	...	

	engine-size	fuel-system	bore	stroke	compression-ratio	horsepower	\
0	130	mpfi	3.47	2.68	9.0	111.0	
1	152	mpfi	2.68	3.47	9.0	154.0	
2	109	mpfi	3.19	3.40	10.0	102.0	
3	136	mpfi	3.19	3.40	8.0	115.0	
4	136	mpfi	3.19	3.40	8.5	110.0	

	peak-rpm	city-mpg	highway-mpg	price
0	5000.0	21	27	16500
1	5000.0	19	26	16500
2	5500.0	24	30	13950
3	5500.0	18	22	17450
4	5500.0	19	25	15250

[5 rows x 26 columns]

As you can see, several question marks appeared in the data frame; those missing values may hinder further analysis.

So, how do we identify all those missing values and deal with them?

How to work with missing data?

Steps for working with missing data:

Identify missing data

Deal with missing data

Correct data format

5 Identify and handle missing values

5.0.1 Identify missing values

Convert “?” to NaN

In the car data set, missing data comes with the question mark “?”. We replace “?” with NaN (Not a Number), Python’s default missing value marker for reasons of computational speed and convenience. Use the function:

to replace A by B.

```
[28]: import numpy as np

# replace "?" to NaN
df.replace("?", np.nan, inplace = True)
df.head(5)
```

```
[28]:      symboling  normalized-losses      make fuel-type aspiration \
0           3             NaN  alfa-romero      gas      std
1           1             NaN  alfa-romero      gas      std
2           2          164.0     audi      gas      std
3           2          164.0     audi      gas      std
4           2             NaN     audi      gas      std

      num-of-doors  body-style drive-wheels engine-location  wheel-base  ... \
0           two  convertible      rwd      front      88.6  ...
1           two   hatchback      rwd      front      94.5  ...
2           four     sedan      fwd      front      99.8  ...
3           four     sedan      4wd      front      99.4  ...
4           two     sedan      fwd      front      99.8  ...

      engine-size  fuel-system  bore  stroke  compression-ratio  horsepower  \
0           130      mpfi  3.47   2.68           9.0      111.0
1           152      mpfi  2.68   3.47           9.0      154.0
2           109      mpfi  3.19   3.40          10.0      102.0
3           136      mpfi  3.19   3.40           8.0      115.0
4           136      mpfi  3.19   3.40           8.5      110.0

      peak-rpm  city-mpg  highway-mpg  price
0      5000.0      21      27  16500
1      5000.0      19      26  16500
2      5500.0      24      30  13950
3      5500.0      18      22  17450
4      5500.0      19      25  15250
```



```
[5 rows x 26 columns]
```

Evaluating for Missing Data

The missing values are converted by default. Use the following functions to identify these missing values. You can use two methods to detect missing data:

```
.isnull()
```

```
.notnull()
```

The output is a boolean value indicating whether the value that is passed into the argument is in fact missing data.

```
[29]: missing_data = df.isnull()
missing_data.head(5)
```

```
[29]:
```

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	\
0	False	True	False	False	False	False	
1	False	True	False	False	False	False	
2	False	False	False	False	False	False	
3	False	False	False	False	False	False	
4	False	True	False	False	False	False	

	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	\
0	False	False	False	False	...	False	
1	False	False	False	False	...	False	
2	False	False	False	False	...	False	
3	False	False	False	False	...	False	
4	False	False	False	False	...	False	

	fuel-system	bore	stroke	compression-ratio	horsepower	peak-rpm	\
0	False	False	False	False	False	False	
1	False	False	False	False	False	False	
2	False	False	False	False	False	False	
3	False	False	False	False	False	False	
4	False	False	False	False	False	False	

	city-mpg	highway-mpg	price
0	False	False	False
1	False	False	False
2	False	False	False
3	False	False	False
4	False	False	False

```
[5 rows x 26 columns]
```

“True” means the value is a missing value while “False” means the value is not a missing value.

Count missing values in each column

Using a for loop in Python, you can quickly figure out the number of missing values in each column. As mentioned above, “True” represents a missing value and “False” means the value is present in the data set. In the body of the for loop the method “.value_counts()” counts the number of “True” values.

```
[30]: for column in missing_data.columns.values.tolist():  
      print(column)  
      print (missing_data[column].value_counts())  
      print("")
```

```
symboling  
symboling  
False      200  
Name: count, dtype: int64
```

```
normalized-losses  
normalized-losses  
False      164  
True        36  
Name: count, dtype: int64
```

```
make  
make  
False      200  
Name: count, dtype: int64
```

```
fuel-type  
fuel-type  
False      200  
Name: count, dtype: int64
```

```
aspiration  
aspiration  
False      200  
Name: count, dtype: int64
```

```
num-of-doors  
num-of-doors  
False      198  
True         2  
Name: count, dtype: int64
```

```
body-style  
body-style  
False      200  
Name: count, dtype: int64
```

```
drive-wheels
```

drive-wheels
False 200
Name: count, dtype: int64

engine-location
engine-location
False 200
Name: count, dtype: int64

wheel-base
wheel-base
False 200
Name: count, dtype: int64

length
length
False 200
Name: count, dtype: int64

width
width
False 200
Name: count, dtype: int64

height
height
False 200
Name: count, dtype: int64

curb-weight
curb-weight
False 200
Name: count, dtype: int64

engine-type
engine-type
False 200
Name: count, dtype: int64

num-of-cylinders
num-of-cylinders
False 200
Name: count, dtype: int64

engine-size
engine-size
False 200
Name: count, dtype: int64

fuel-system
fuel-system
False 200
Name: count, dtype: int64

bore
bore
False 196
True 4
Name: count, dtype: int64

stroke
stroke
False 196
True 4
Name: count, dtype: int64

compression-ratio
compression-ratio
False 200
Name: count, dtype: int64

horsepower
horsepower
False 198
True 2
Name: count, dtype: int64

peak-rpm
peak-rpm
False 198
True 2
Name: count, dtype: int64

city-mpg
city-mpg
False 200
Name: count, dtype: int64

highway-mpg
highway-mpg
False 200
Name: count, dtype: int64

price
price
False 200

Name: count, dtype: int64

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

“normalized-losses”: 41 missing data

“num-of-doors”: 2 missing data

“bore”: 4 missing data

“stroke” : 4 missing data

“horsepower”: 2 missing data

“peak-rpm”: 2 missing data

“price”: 4 missing data

5.0.2 Deal with missing data

How should you deal with missing data?

Drop data a. Drop the whole row b. Drop the whole column

Replace data a. Replace it by mean b. Replace it by frequency c. Replace it based on other functions

You should only drop whole columns if most entries in the column are empty. In the data set, none of the columns are empty enough to drop entirely. You have some freedom in choosing which method to replace data; however, some methods may seem more reasonable than others. Apply each method to 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 are 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: You want to predict price. You cannot use any data entry without price data for prediction; therefore any row now without price data is not useful to you.

Calculate the mean value for the “normalized-losses” column

```
[31]: 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

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

Calculate the mean value for the “bore” column

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

Average of bore: 3.3300000000000005

Replace “NaN” with the mean value in the “bore” column

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

Replace NaN in “stroke” column with the mean value.

```
[35]: #Calculate the mean vaule for "stroke" column
      avg_stroke = df["stroke"].astype("float").mean(axis = 0)
      print("Average of stroke:", avg_stroke)

      # replace NaN by mean value in "stroke" column
      df["stroke"].replace(np.nan, avg_stroke, inplace = True)
```

Average of stroke: 3.2598469387755107

Calculate the mean value for the “horsepower” column

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

Average horsepower: 103.35858585858585

Replace “NaN” with the mean value in the “horsepower” column

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

Calculate the mean value for “peak-rpm” column

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

Average peak rpm: 5118.181818181818

Replace “NaN” with the mean value in the “peak-rpm” column

```
[39]: 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:

```
[40]: df['num-of-doors'].value_counts()
```

```
[40]: num-of-doors
four      113
two       85
Name: count, dtype: int64
```

You can see that four doors is the most common type. We can also use the “.idxmax()” method to calculate the most common type automatically:

```
[41]: df['num-of-doors'].value_counts().idxmax()
```

```
[41]: 'four'
```

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

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

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

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

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

```
[44]: df.head()
```

```
[44]:   symboling  normalized-losses      make fuel-type aspiration \
0         3         122.0  alfa-romero    gas      std
1         1         122.0  alfa-romero    gas      std
2         2         164.0    audi      gas      std
3         2         164.0    audi      gas      std
4         2         122.0    audi      gas      std

   num-of-doors  body-style drive-wheels engine-location  wheel-base  ...  \
0         two  convertible      rwd      front      88.6  ...
1         two   hatchback      rwd      front      94.5  ...
2         four    sedan      fwd      front      99.8  ...
3         four    sedan      4wd      front      99.4  ...
4         two    sedan      fwd      front      99.8  ...

   engine-size  fuel-system  bore  stroke  compression-ratio  horsepower  \
0         130      mpfi  3.47   2.68           9.0      111.0
1         152      mpfi  2.68   3.47           9.0      154.0
```

2	109	mpfi	3.19	3.40	10.0	102.0
3	136	mpfi	3.19	3.40	8.0	115.0
4	136	mpfi	3.19	3.40	8.5	110.0

	peak-rpm	city-mpg	highway-mpg	price
0	5000.0	21	27	16500
1	5000.0	19	26	16500
2	5500.0	24	30	13950
3	5500.0	18	22	17450
4	5500.0	19	25	15250

[5 rows x 26 columns]

Now, you have a data set with no missing values.

5.0.3 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, you use:

.dtype() to check the data type

.astype() to change the data type

Let's list the data types for each column

```
[45]: df.dtypes
```

```
[45]: symboling          int64
normalized-losses      float64
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            float64
peak-rpm              float64
city-mpg              int64
highway-mpg           int64
price                 int64
dtype: object

```

As you 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, the numerical values 'bore' and 'stroke' describe the engines, so you should expect them to be of the type 'float' or 'int'; however, they are shown as type 'object'. You have to convert data types into a proper format for each column using the "astype()" method.

Convert data types to proper format

```

[46]: 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")

```

Let us list the columns after the conversion

```

[47]: df.dtypes

```

```

[47]: 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            float64
peak-rpm              float64

```

```

city-mpg          int64
highway-mpg       int64
price            float64
dtype: object

```

Now you finally obtained the cleansed data set with no missing values and with all data in its proper format.

5.1 Data Standardization

You usually collect data from different agencies in different formats. (Data standardization is also a term for a particular type of data normalization where you subtract the mean and divide by the standard deviation.)

What is standardization?

Standardization is the process of transforming data into a common format, allowing the researcher to make the meaningful comparison.

Example

Transform mpg to L/100km:

In your data set, the fuel consumption columns “city-mpg” and “highway-mpg” are represented by mpg (miles per gallon) unit. Assume you are developing an application in a country that accepts the fuel consumption with L/100km standard.

You will need to apply data transformation to transform mpg into L/100km.

Use this formula for unit conversion:

$L/100km = 235 / mpg$

You can do many mathematical operations directly using Pandas.

```
[48]: df.head()
```

```

[48]:   symboling  normalized-losses      make fuel-type aspiration \
0         3         122  alfa-romero    gas      std
1         1         122  alfa-romero    gas      std
2         2         164      audi    gas      std
3         2         164      audi    gas      std
4         2         122      audi    gas      std

   num-of-doors  body-style drive-wheels engine-location  wheel-base  ... \
0         two  convertible      rwd      front      88.6  ...
1         two   hatchback      rwd      front      94.5  ...
2         four     sedan      fwd      front      99.8  ...
3         four     sedan      4wd      front      99.4  ...
4         two     sedan      fwd      front      99.8  ...

   engine-size  fuel-system  bore  stroke  compression-ratio horsepower \

```

0	130	mpfi	3.47	2.68	9.0	111.0
1	152	mpfi	2.68	3.47	9.0	154.0
2	109	mpfi	3.19	3.40	10.0	102.0
3	136	mpfi	3.19	3.40	8.0	115.0
4	136	mpfi	3.19	3.40	8.5	110.0

	peak-rpm	city-mpg	highway-mpg	price
0	5000.0	21	27	16500.0
1	5000.0	19	26	16500.0
2	5500.0	24	30	13950.0
3	5500.0	18	22	17450.0
4	5500.0	19	25	15250.0

[5 rows x 26 columns]

```
[49]: # Convert mpg to L/100km by mathematical operation (235 divided by mpg)
df['city-L/100km'] = 235/df["city-mpg"]

# check your transformed data
df.head()
```

```
[49]:      symboling  normalized-losses      make fuel-type aspiration \
0           3           122  alfa-romero      gas      std
1           1           122  alfa-romero      gas      std
2           2           164      audi      gas      std
3           2           164      audi      gas      std
4           2           122      audi      gas      std

      num-of-doors  body-style drive-wheels engine-location  wheel-base  ... \
0           two  convertible      rwd      front      88.6  ...
1           two   hatchback      rwd      front      94.5  ...
2          four      sedan      fwd      front      99.8  ...
3          four      sedan      4wd      front      99.4  ...
4           two      sedan      fwd      front      99.8  ...

      fuel-system  bore  stroke  compression-ratio  horsepower  peak-rpm  city-mpg  \
0          mpfi  3.47   2.68              9.0      111.0  5000.0      21
1          mpfi  2.68   3.47              9.0      154.0  5000.0      19
2          mpfi  3.19   3.40             10.0      102.0  5500.0      24
3          mpfi  3.19   3.40              8.0      115.0  5500.0      18
4          mpfi  3.19   3.40              8.5      110.0  5500.0      19

      highway-mpg  price  city-L/100km
0           27  16500.0    11.190476
1           26  16500.0    12.368421
2           30  13950.0     9.791667
3           22  17450.0    13.055556
```

```
4          25  15250.0      12.368421
```

```
[5 rows x 27 columns]
```

Transform mpg to L/100km in the column of “highway-mpg” and change the name of column to “highway-L/100km”.

```
[50]: # transform mpg to L/100km by mathematical operation (235 divided by mpg)
df['highway-L/100km']=235/df["highway-mpg"]

# check your transformed data
df.head()
```

```
[50]:   symboling  normalized-losses      make fuel-type aspiration \
0          3                122  alfa-romero      gas      std
1          1                122  alfa-romero      gas      std
2          2                164      audi      gas      std
3          2                164      audi      gas      std
4          2                122      audi      gas      std

   num-of-doors  body-style drive-wheels engine-location  wheel-base  ... \
0          two  convertible      rwd      front      88.6  ...
1          two   hatchback      rwd      front      94.5  ...
2          four      sedan      fwd      front      99.8  ...
3          four      sedan      4wd      front      99.4  ...
4          two      sedan      fwd      front      99.8  ...

   bore  stroke  compression-ratio  horsepower  peak-rpm  city-mpg  highway-mpg  \
0  3.47   2.68                9.0      111.0   5000.0      21      27
1  2.68   3.47                9.0      154.0   5000.0      19      26
2  3.19   3.40               10.0      102.0   5500.0      24      30
3  3.19   3.40                8.0      115.0   5500.0      18      22
4  3.19   3.40                8.5      110.0   5500.0      19      25

   price  city-L/100km  highway-L/100km
0  16500.0      11.190476      8.703704
1  16500.0      12.368421      9.038462
2  13950.0       9.791667      7.833333
3  17450.0      13.055556     10.681818
4  15250.0      12.368421      9.400000
```

```
[5 rows x 28 columns]
```

5.2 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

scaling the variable so the variable values range from 0 to 1

Example

To demonstrate normalization, say you want to scale the columns “length”, “width” and “height”.

Target: normalize those variables so their value ranges from 0 to 1

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

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

Normalize the column “height”.

```
[52]: df['height'] = df['height']/df['height'].max()

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

```
[52]:
```

	length	width	height
0	0.811148	0.890278	0.816054
1	0.822681	0.909722	0.876254
2	0.848630	0.919444	0.908027
3	0.848630	0.922222	0.908027
4	0.851994	0.920833	0.887960

Here you’ve normalized “length”, “width” and “height” to fall in the range of [0,1].

5.3 Binning

Why binning?

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

Example:

In your data set, “horsepower” is a real valued variable ranging from 48 to 288 and it has 59 unique values. What if you only care about the price difference between cars with high horsepower, medium horsepower, and little horsepower (3 types)? You can rearrange them into three ‘bins’ to simplify analysis.

Use the Pandas method ‘cut’ to segment the ‘horsepower’ column into 3 bins.

Example of Binning Data In Pandas

Convert data to correct format:

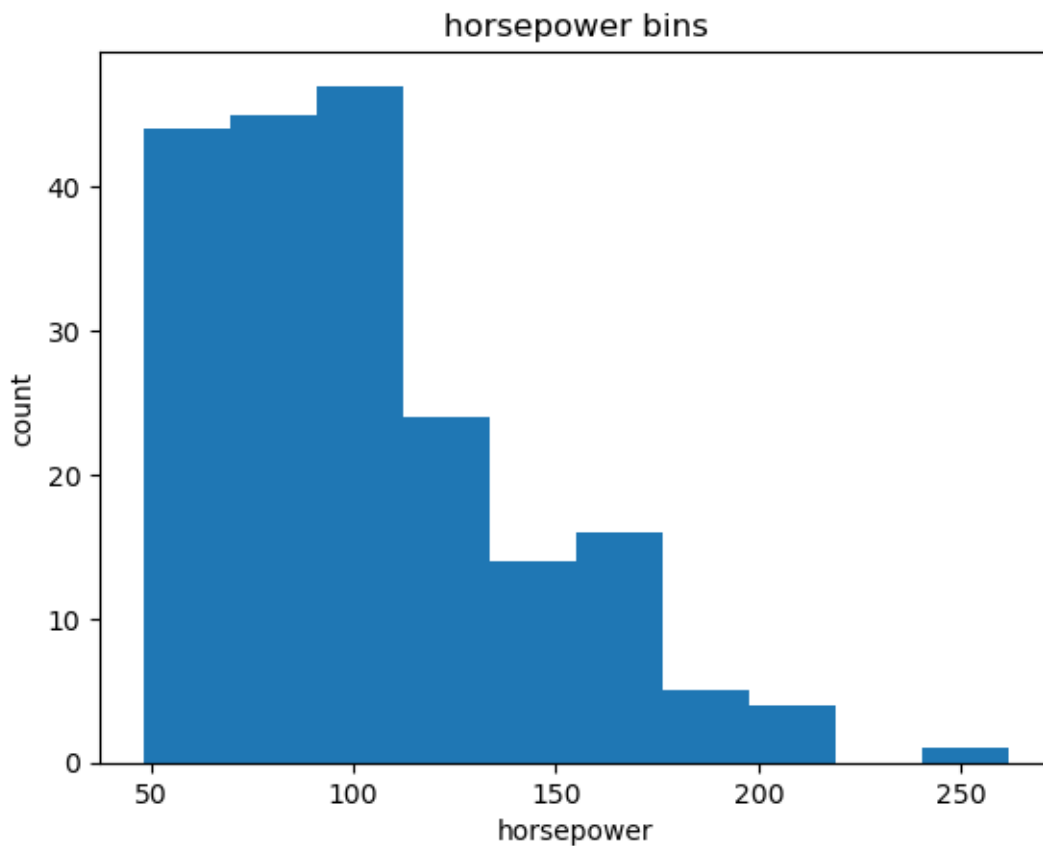
```
[53]: df["horsepower"]=df["horsepower"].astype(int, copy=True)
```

Plot the histogram of horsepower to see the distribution of horsepower.

```
[54]: %matplotlib inline
import matplotlib as plt
from matplotlib import pyplot
plt.pyplot.hist(df["horsepower"])

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

```
[54]: Text(0.5, 1.0, 'horsepower bins')
```



Find 3 bins of equal size bandwidth by using Numpy's `linspace(start_value, end_value, numbers_generated)` function.

Since you want to include the minimum value of horsepower, set `start_value = min(df["horsepower"])`.

Since you want to include the maximum value of horsepower, set `end_value = max(df["horsepower"])`.

Since you are building 3 bins of equal length, you need 4 dividers, so `numbers_generated = 4`.

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.

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

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

Set group names:

```
[56]: group_names = ['Low', 'Medium', 'High']
```

Apply the function “cut” to determine what each value of `df['horsepower']` belongs to.

```
[57]: df['horsepower-binned'] = pd.cut(df['horsepower'], bins, labels=group_names,
    ↪ include_lowest=True )
      df[['horsepower', 'horsepower-binned']].head(20)
```

```
[57]:
```

	horsepower	horsepower-binned
0	111	Low
1	154	Medium
2	102	Low
3	115	Low
4	110	Low
5	110	Low
6	110	Low
7	140	Medium
8	101	Low
9	101	Low
10	121	Medium
11	121	Medium
12	121	Medium
13	182	Medium
14	182	Medium
15	182	Medium
16	48	Low
17	70	Low
18	70	Low
19	68	Low

See the number of vehicles in each bin:

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

```
[58]: horsepower-binned
      Low      152
      Medium   43
      High     5
```

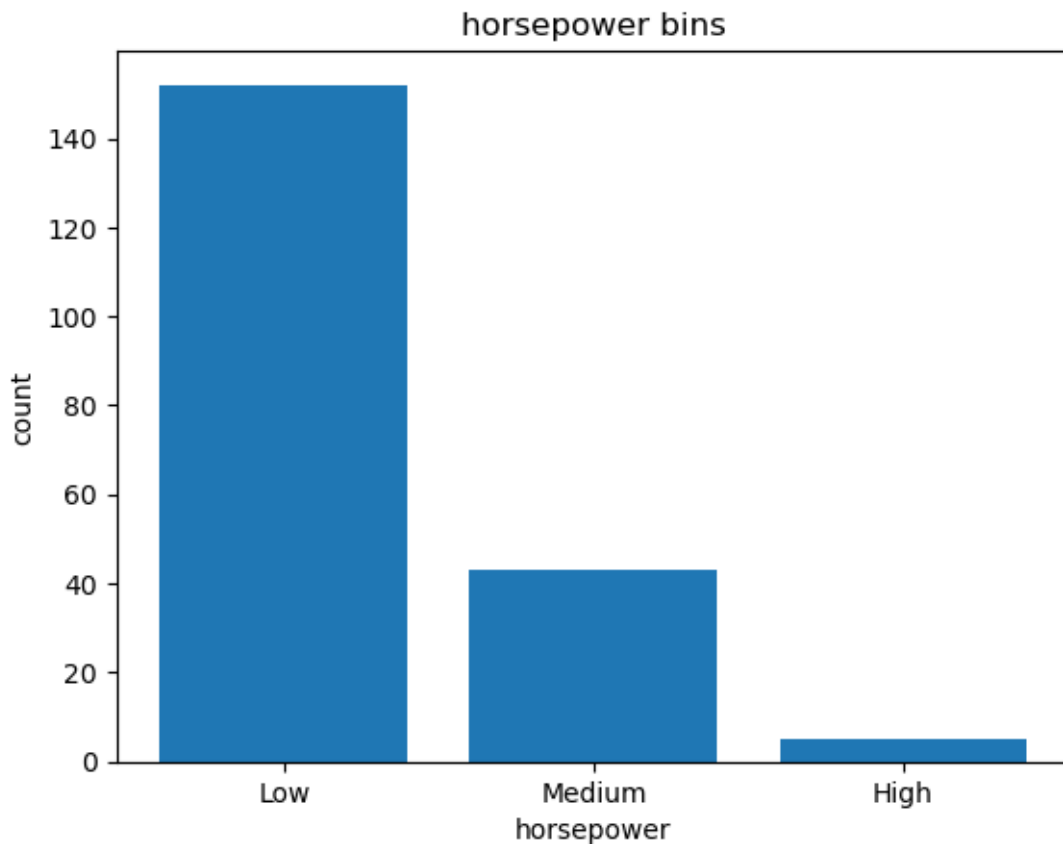
Name: count, dtype: int64

Plot the distribution of each bin:

```
[59]: %matplotlib inline
import matplotlib as plt
from matplotlib import pyplot
pyplot.bar(group_names, df["horsepower-binned"].value_counts())

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

[59]: Text(0.5, 1.0, 'horsepower bins')



Look at the data frame above carefully. You will find that the last column provides the bins for “horsepower” based on 3 categories (“Low”, “Medium” and “High”).

You successfully narrowed down the intervals from 59 to 3!

Bins Visualization

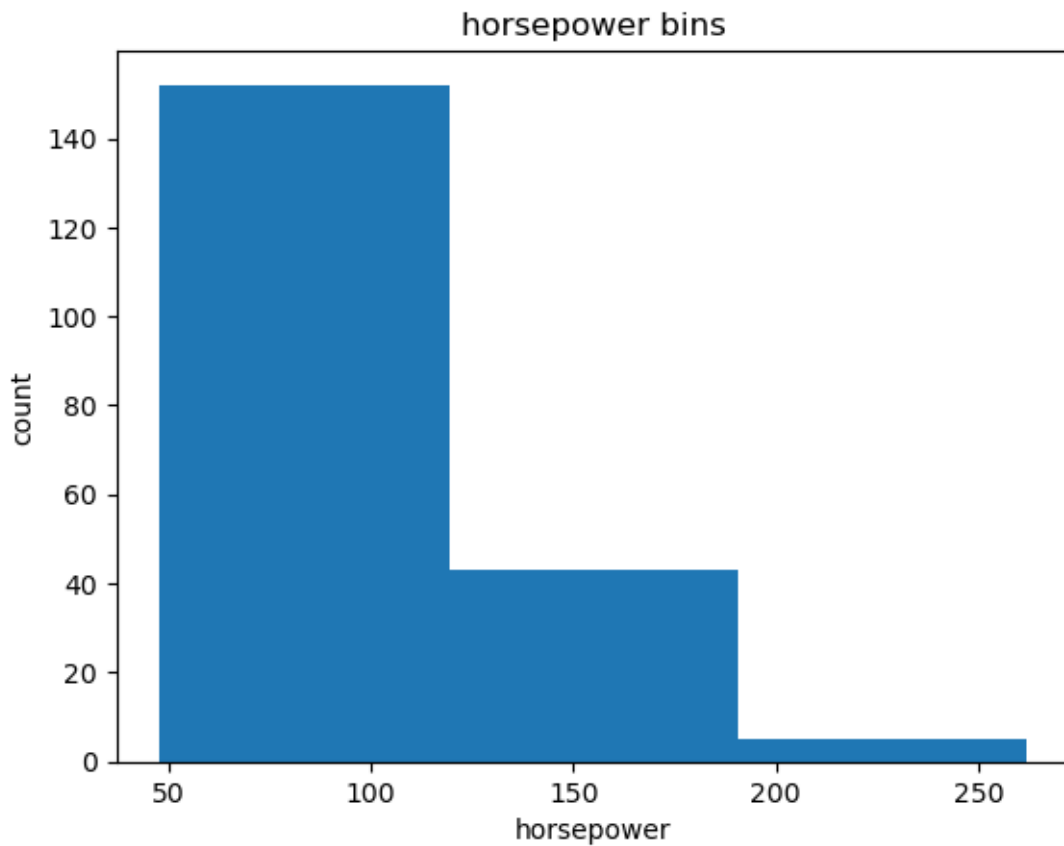
Normally, you use a histogram to visualize the distribution of bins we created above.

```
[60]: %matplotlib inline
import matplotlib as plt
from matplotlib import pyplot

# draw histogram of attribute "horsepower" with bins = 3
plt.pyplot.hist(df["horsepower"], bins = 3)

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

```
[60]: Text(0.5, 1.0, 'horsepower bins')
```



The plot above shows the binning result for the attribute “horsepower”.

5.4 Indicator Variable

What is an indicator 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.

Why use indicator variables?

You use indicator variables so you can use categorical variables for regression analysis in the later modules.

Example

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, you can convert “fuel-type” to indicator variables.

Use the Panda method ‘get_dummies’ to assign numerical values to different categories of fuel type.

```
[61]: df.columns
```

```
[61]: Index(['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration',  
          'num-of-doors', 'body-style', 'drive-wheels', 'engine-location',  
          'wheel-base', 'length', 'width', 'height', 'curb-weight', 'engine-type',  
          'num-of-cylinders', 'engine-size', 'fuel-system', 'bore', 'stroke',  
          'compression-ratio', 'horsepower', 'peak-rpm', 'city-mpg',  
          'highway-mpg', 'price', 'city-L/100km', 'highway-L/100km',  
          'horsepower-binned'],  
          dtype='object')
```

Get the indicator variables and assign it to data frame “dummy_variable_1”:

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

```
[62]:   diesel   gas  
0   False   True  
1   False   True  
2   False   True  
3   False   True  
4   False   True
```

Change the column names for clarity:

```
[63]: dummy_variable_1.rename(columns={'gas': 'fuel-type-gas', 'diesel':  
    ↪ 'fuel-type-diesel'}, inplace=True)  
      dummy_variable_1.head()
```

```
[63]:   fuel-type-diesel  fuel-type-gas  
0                False              True
```

1	False	True
2	False	True
3	False	True
4	False	True

In the data frame, column ‘fuel-type’ now has values for ‘gas’ and ‘diesel’ as 0s and 1s.

```
[64]: # 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)
```

```
[65]: df.head()
```

```
[65]:  symboling  normalized-losses      make aspiration num-of-doors  \
0         3           122  alfa-romero      std         two
1         1           122  alfa-romero      std         two
2         2           164      audi      std         four
3         2           164      audi      std         four
4         2           122      audi      std         two

      body-style drive-wheels engine-location  wheel-base  length  ...  \
0  convertible      rwd      front      88.6  0.811148  ...
1   hatchback      rwd      front      94.5  0.822681  ...
2      sedan      fwd      front      99.8  0.848630  ...
3      sedan      4wd      front      99.4  0.848630  ...
4      sedan      fwd      front      99.8  0.851994  ...

      horsepower  peak-rpm  city-mpg  highway-mpg  price  city-L/100km  \
0         111    5000.0      21      27  16500.0    11.190476
1         154    5000.0      19      26  16500.0    12.368421
2         102    5500.0      24      30  13950.0     9.791667
3         115    5500.0      18      22  17450.0    13.055556
4         110    5500.0      19      25  15250.0    12.368421

      highway-L/100km  horsepower-binned  fuel-type-diesel  fuel-type-gas
0         8.703704             Low      False      True
1         9.038462           Medium      False      True
2         7.833333             Low      False      True
3        10.681818             Low      False      True
4         9.400000             Low      False      True
```

```
[5 rows x 30 columns]
```

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

Create an indicator variable for the column “aspiration”

```
[66]: # get indicator variables of aspiration and assign it to data frame
      ↪ "dummy_variable_2"
      dummy_variable_2 = pd.get_dummies(df['aspiration'])

      # change column names for clarity
      dummy_variable_2.rename(columns={'std': 'aspiration-std', 'turbo':
      ↪ 'aspiration-turbo'}, inplace=True)

      # show first 5 instances of data frame "dummy_variable_1"
      dummy_variable_2.head()
```

```
[66]: aspiration-std aspiration-turbo
0          True          False
1          True          False
2          True          False
3          True          False
4          True          False
```

Merge the new dataframe to the original dataframe, then drop the column 'aspiration'.

```
[67]: # merge the new dataframe to the original dataframe
      #df = pd.concat([df, dummy_variable_2], axis=1)
```

Save the new csv:

```
[68]: df.to_csv('usedCars.csv', index=None)
```

6 Exploratory Data Analysis

Estimated time needed: **30** minutes

6.1 Objectives

- Explore features or characteristics to predict price of car
- Analyze patterns and run descriptive statistical analysis
- Group data based on identified parameters and create pivot tables
- Identify the effect of independent attributes on price of cars

Table of Contents

Import Data from Module

Analyzing Individual Feature Patterns using Visualization

Descriptive Statistical Analysis

Basics of Grouping

Correlation and Causation

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

6.2 Import Data from Module 2

Setup

Import libraries:

```
[69]: #install specific version of libraries used in lab
      #! mamba install pandas==1.3.3
      #! mamba install numpy=1.21.2
      #! mamba install scipy=1.7.1-y
      #! mamba install seaborn=0.9.0-y
```

```
[70]: import pandas as pd
      import numpy as np
      import seaborn as sns
```

Download the updated dataset by running the cell below.

The functions below will download the dataset into your browser and store it in dataframe df:

This dataset was hosted on IBM Cloud object. Click [HERE](#) for free storage.

```
[71]: '''from pyodide.http import pyfetch

      async def download(url, filename):
          response = await pyfetch(url)
          if response.status == 200:
              with open(filename, "wb") as f:
                  f.write(await response.bytes())'''
```

```
[71]: 'from pyodide.http import pyfetch\n\nasync def download(url, filename):\n    response = await pyfetch(url)\n    if response.status == 200:\n        with\n        open(filename, "wb") as f:\n            f.write(await response.bytes())'
```

```
[72]: #file_path= "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/\n      ↪IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/\n      ↪automobileEDA.csv"
```

```
[73]: file_name="usedCars.csv"
```

```
[74]: df = pd.read_csv(file_name)
```

```
[75]: #filepath='https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/\n      ↪IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/\n      ↪automobileEDA.csv'\n      #df = pd.read_csv(filepath, header=None)
```

View the first 5 values of the updated dataframe using `dataframe.head()`

```
[76]: df.head()
```

```
[76]:      symboling  normalized-losses      make aspiration num-of-doors \
0         3             122  alfa-romero      std         two
1         1             122  alfa-romero      std         two
2         2             164      audi      std         four
3         2             164      audi      std         four
4         2             122      audi      std         two

      body-style drive-wheels engine-location  wheel-base  length  ... \
0  convertible      rwd      front      88.6  0.811148  ...
1   hatchback      rwd      front      94.5  0.822681  ...
2      sedan      fwd      front      99.8  0.848630  ...
3      sedan      4wd      front      99.4  0.848630  ...
4      sedan      fwd      front      99.8  0.851994  ...

      horsepower  peak-rpm  city-mpg  highway-mpg  price  city-L/100km \
0         111    5000.0      21      27  16500.0    11.190476
1         154    5000.0      19      26  16500.0    12.368421
2         102    5500.0      24      30  13950.0     9.791667
3         115    5500.0      18      22  17450.0    13.055556
4         110    5500.0      19      25  15250.0    12.368421

      highway-L/100km  horsepower-binned  fuel-type-diesel  fuel-type-gas
0         8.703704                Low      False      True
1         9.038462             Medium      False      True
2         7.833333                Low      False      True
3        10.681818                Low      False      True
4         9.400000                Low      False      True

[5 rows x 30 columns]
```

6.3 Analyzing Individual Feature Patterns Using Visualization

Import visualization packages “Matplotlib” and “Seaborn”. Don’t forget about “%matplotlib inline” to plot in a Jupyter notebook.

```
[77]: import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

How to choose the right visualization method?

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.

```
[78]: # list the data types for each column
print(df.dtypes)
```

```
symboling      int64
normalized-losses  int64
```

```

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           int64
peak-rpm            float64
city-mpg             int64
highway-mpg          int64
price               float64
city-L/100km         float64
highway-L/100km      float64
horsepower-binned    object
fuel-type-diesel     bool
fuel-type-gas        bool
dtype: object

```

```
[79]: df['peak-rpm'].dtypes
```

```
[79]: dtype('float64')
```

For example, we can calculate the correlation between variables of type “int64” or “float64” using the method “corr”:

```
[80]: numeric_df = df.select_dtypes(include=['float64', 'int64'])
numeric_df.corr()
```

```
[80]:
```

	symboling	normalized-losses	wheel-base	length	\
symboling	1.000000	0.469772	-0.529145	-0.364511	
normalized-losses	0.469772	1.000000	-0.057068	0.019433	
wheel-base	-0.529145	-0.057068	1.000000	0.879005	
length	-0.364511	0.019433	0.879005	1.000000	
width	-0.237262	0.086961	0.814593	0.857271	
height	-0.542261	-0.377664	0.583789	0.492955	
curb-weight	-0.234743	0.099404	0.787584	0.881058	

engine-size	-0.112069	0.112362	0.576779	0.685531
bore	-0.145667	-0.029867	0.501534	0.610817
stroke	0.008244	0.055759	0.144675	0.120888
compression-ratio	-0.181073	-0.114738	0.249689	0.159203
horsepower	0.074581	0.217323	0.375732	0.580477
peak-rpm	0.284011	0.239580	-0.364971	-0.286754
city-mpg	-0.030158	-0.225255	-0.480029	-0.667658
highway-mpg	0.041248	-0.182011	-0.552211	-0.700186
price	-0.083327	0.133999	0.589147	0.691044
city-L/100km	0.062423	0.238712	0.484047	0.659174
highway-L/100km	-0.033159	0.181247	0.584953	0.708466

	width	height	curb-weight	engine-size	bore \
symboling	-0.237262	-0.542261	-0.234743	-0.112069	-0.145667
normalized-losses	0.086961	-0.377664	0.099404	0.112362	-0.029867
wheel-base	0.814593	0.583789	0.787584	0.576779	0.501534
length	0.857271	0.492955	0.881058	0.685531	0.610817
width	1.000000	0.300995	0.867720	0.731100	0.548478
height	0.300995	1.000000	0.310660	0.076255	0.187794
curb-weight	0.867720	0.310660	1.000000	0.849090	0.644532
engine-size	0.731100	0.076255	0.849090	1.000000	0.572786
bore	0.548478	0.187794	0.644532	0.572786	1.000000
stroke	0.182855	-0.081273	0.168642	0.208004	-0.051087
compression-ratio	0.189008	0.259526	0.156444	0.029005	0.002021
horsepower	0.617032	-0.085725	0.758095	0.822656	0.566690
peak-rpm	-0.247388	-0.315756	-0.279411	-0.256702	-0.267010
city-mpg	-0.638155	-0.057087	-0.750390	-0.651002	-0.581365
highway-mpg	-0.684700	-0.111568	-0.795515	-0.679877	-0.590753
price	0.752795	0.137284	0.834420	0.872337	0.543431
city-L/100km	0.677111	0.008923	0.785868	0.745337	0.554069
highway-L/100km	0.739845	0.088903	0.837217	0.783593	0.558759

	stroke	compression-ratio	horsepower	peak-rpm \
symboling	0.008244	-0.181073	0.074581	0.284011
normalized-losses	0.055759	-0.114738	0.217323	0.239580
wheel-base	0.144675	0.249689	0.375732	-0.364971
length	0.120888	0.159203	0.580477	-0.286754
width	0.182855	0.189008	0.617032	-0.247388
height	-0.081273	0.259526	-0.085725	-0.315756
curb-weight	0.168642	0.156444	0.758095	-0.279411
engine-size	0.208004	0.029005	0.822656	-0.256702
bore	-0.051087	0.002021	0.566690	-0.267010
stroke	1.000000	0.186761	0.100351	-0.066173
compression-ratio	0.186761	1.000000	-0.214162	-0.436244
horsepower	0.100351	-0.214162	1.000000	0.108161
peak-rpm	-0.066173	-0.436244	0.108161	1.000000
city-mpg	-0.040677	0.330897	-0.822397	-0.116308

highway-mpg	-0.040282	0.267929	-0.804714	-0.059326
price	0.083296	0.071176	0.809779	-0.101519
city-L/100km	0.041470	-0.298898	0.889584	0.116510
highway-L/100km	0.051148	-0.222957	0.840687	0.018183

	city-mpg	highway-mpg	price	city-L/100km	\
symboling	-0.030158	0.041248	-0.083327	0.062423	
normalized-losses	-0.225255	-0.182011	0.133999	0.238712	
wheel-base	-0.480029	-0.552211	0.589147	0.484047	
length	-0.667658	-0.700186	0.691044	0.659174	
width	-0.638155	-0.684700	0.752795	0.677111	
height	-0.057087	-0.111568	0.137284	0.008923	
curb-weight	-0.750390	-0.795515	0.834420	0.785868	
engine-size	-0.651002	-0.679877	0.872337	0.745337	
bore	-0.581365	-0.590753	0.543431	0.554069	
stroke	-0.040677	-0.040282	0.083296	0.041470	
compression-ratio	0.330897	0.267929	0.071176	-0.298898	
horsepower	-0.822397	-0.804714	0.809779	0.889584	
peak-rpm	-0.116308	-0.059326	-0.101519	0.116510	
city-mpg	1.000000	0.972024	-0.687186	-0.949692	
highway-mpg	0.972024	1.000000	-0.705115	-0.929940	
price	-0.687186	-0.705115	1.000000	0.790291	
city-L/100km	-0.949692	-0.929940	0.790291	1.000000	
highway-L/100km	-0.909113	-0.951133	0.801313	0.958312	

	highway-L/100km
symboling	-0.033159
normalized-losses	0.181247
wheel-base	0.584953
length	0.708466
width	0.739845
height	0.088903
curb-weight	0.837217
engine-size	0.783593
bore	0.558759
stroke	0.051148
compression-ratio	-0.222957
horsepower	0.840687
peak-rpm	0.018183
city-mpg	-0.909113
highway-mpg	-0.951133
price	0.801313
city-L/100km	0.958312
highway-L/100km	1.000000

The diagonal elements are always one; we will study correlation more precisely Pearson correlation in-depth at the end of the notebook.

```
[81]: # Write your code below and press Shift+Enter to execute
df[['bore', 'stroke', 'compression-ratio', 'horsepower']].corr()
```

```
[81]:
```

	bore	stroke	compression-ratio	horsepower
bore	1.000000	-0.051087	0.002021	0.566690
stroke	-0.051087	1.000000	0.186761	0.100351
compression-ratio	0.002021	0.186761	1.000000	-0.214162
horsepower	0.566690	0.100351	-0.214162	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. This will be useful later on for visualizing the fit of the simple linear regression model as well.

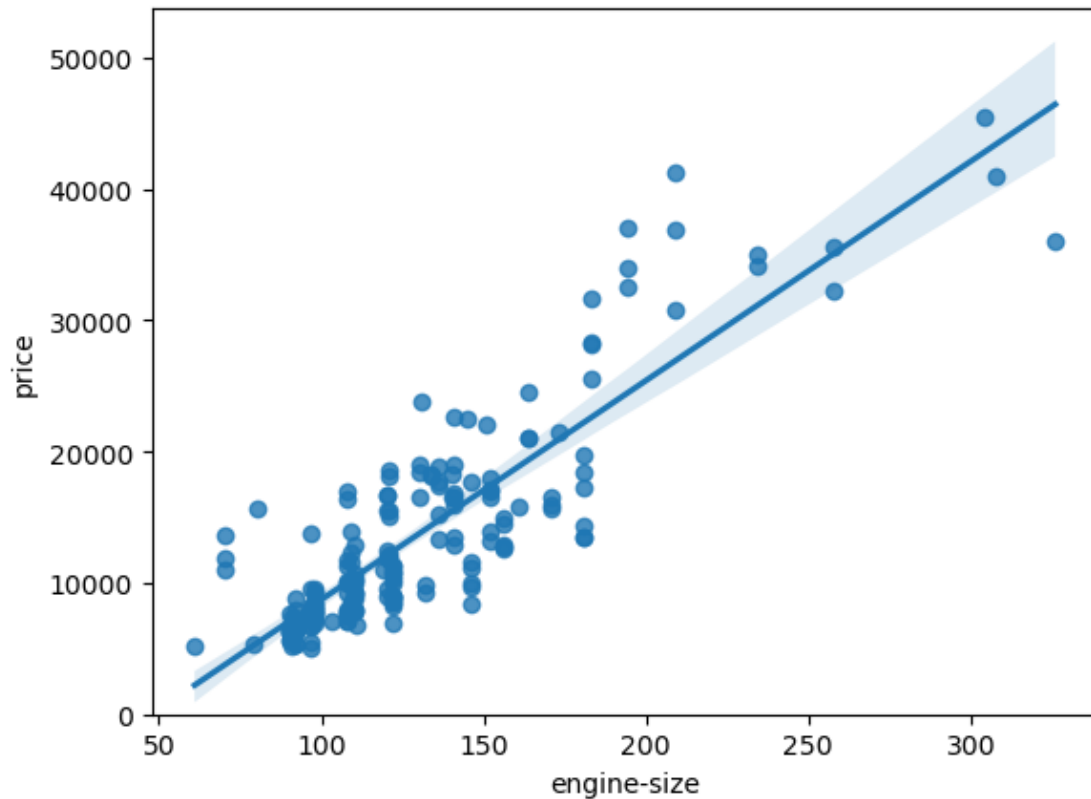
Let’s see several examples of different linear relationships:

Positive Linear Relationship

Let’s find the scatterplot of “engine-size” and “price”.

```
[82]: # Engine size as potential predictor variable of price
sns.regplot(x="engine-size", y="price", data=df)
plt.ylim(0,)
```

```
[82]: (0.0, 53738.88790772015)
```



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.

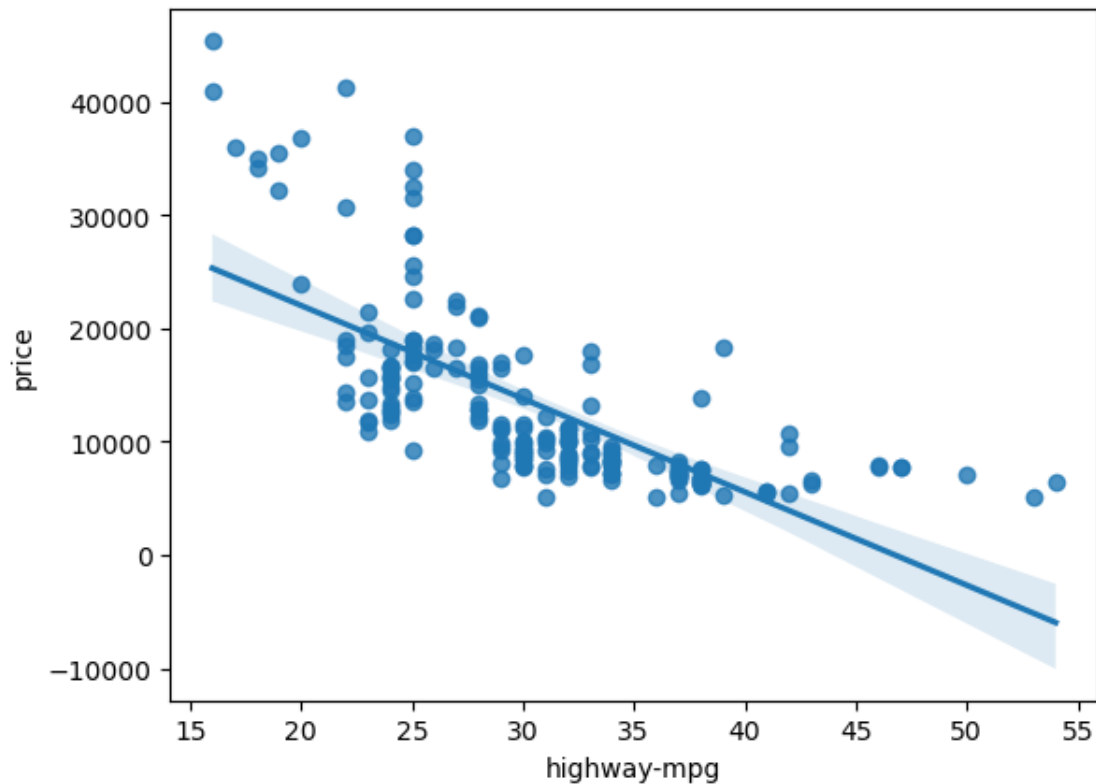
```
[83]: df[["engine-size", "price"]].corr()
```

```
[83]:      engine-size    price
engine-size    1.000000  0.872337
price          0.872337  1.000000
```

Highway mpg is a potential predictor variable of price. Let’s find the scatterplot of “highway-mpg” and “price”.

```
[84]: sns.regplot(x="highway-mpg", y="price", data=df)
```

```
[84]: <Axes: xlabel='highway-mpg', ylabel='price'>
```



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.

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

```
[85]:
```

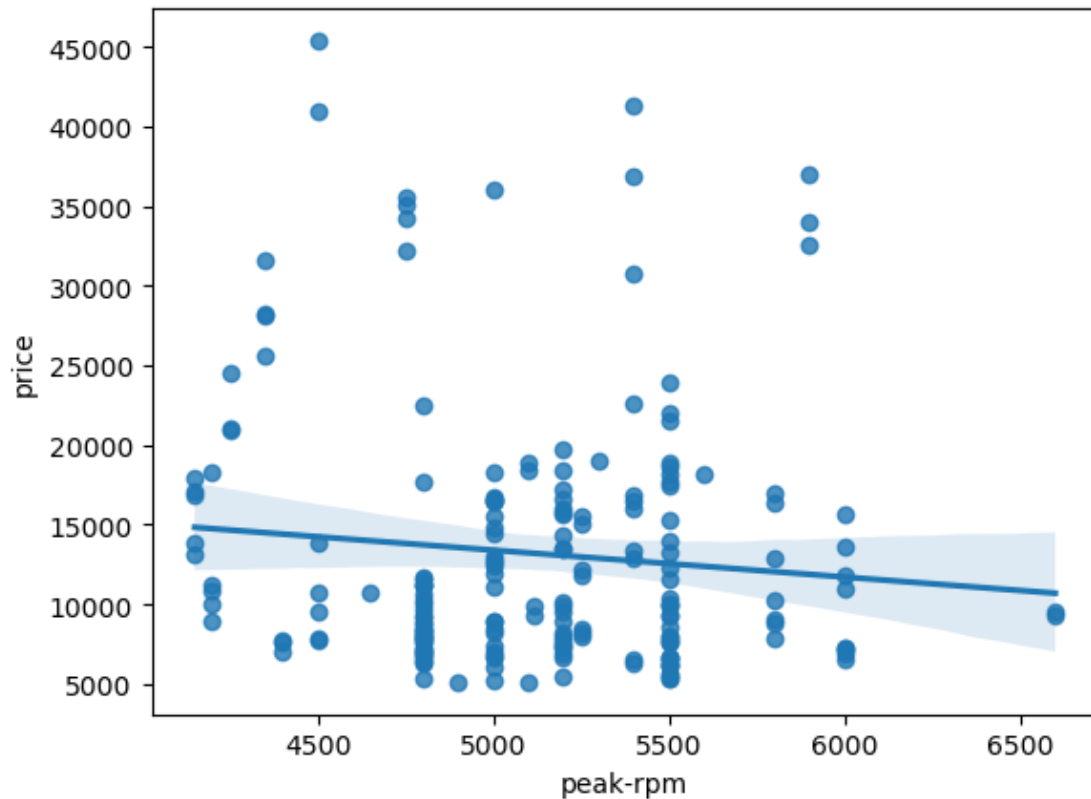
	highway-mpg	price
highway-mpg	1.000000	-0.705115
price	-0.705115	1.000000

Weak Linear Relationship

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

```
[86]: sns.regplot(x="peak-rpm", y="price", data=df)
```

```
[86]: <Axes: xlabel='peak-rpm', ylabel='price'>
```



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.

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

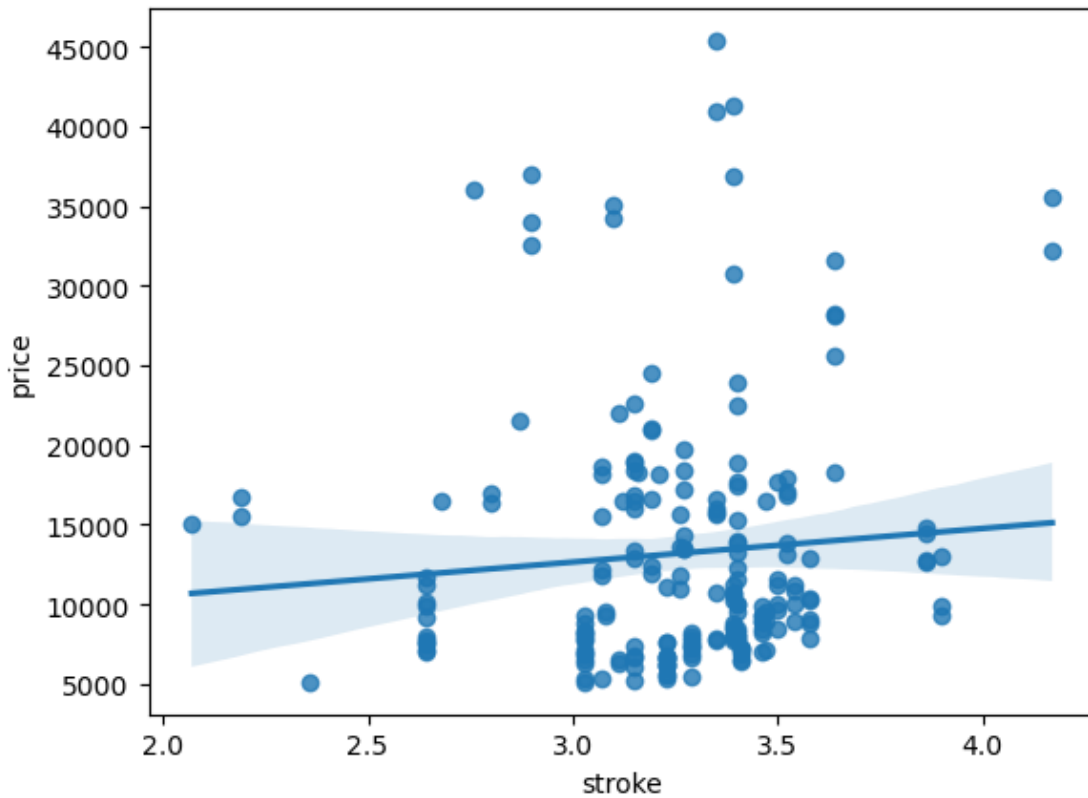
```
[87]:      peak-rpm    price
peak-rpm  1.000000 -0.101519
price    -0.101519  1.000000
```

```
[88]: # Write your code below and press Shift+Enter to execute
df[["stroke", "price"]].corr()
```

```
[88]:      stroke    price
stroke  1.000000  0.083296
price   0.083296  1.000000
```

```
[89]: # Write your code below and press Shift+Enter to execute
sns.regplot(x="stroke", y="price", data=df)
```

```
[89]: <Axes: xlabel='stroke', ylabel='price'>
```



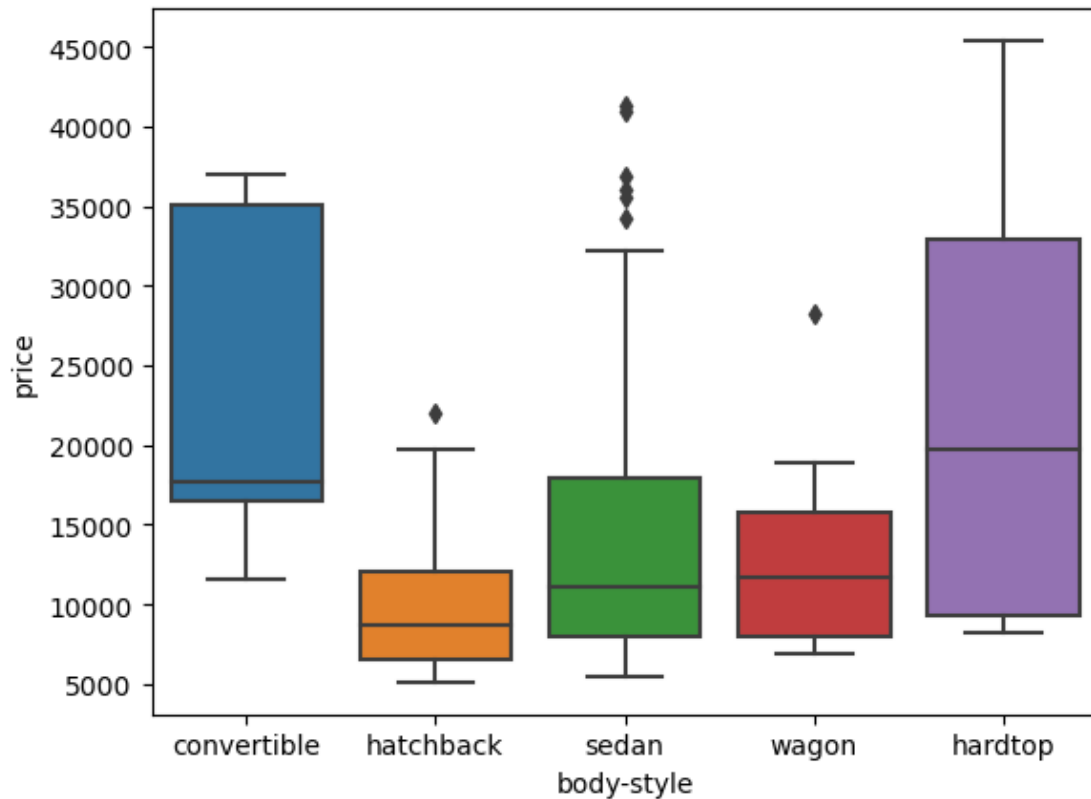
Categorical Variables

These are variables that describe a ‘characteristic’ of a data unit, and are selected from a small group of categories. The categorical variables can have the type “object” or “int64”. A good way to visualize categorical variables is by using boxplots.

Let’s look at the relationship between “body-style” and “price”.

```
[90]: sns.boxplot(x="body-style", y="price", data=df)
```

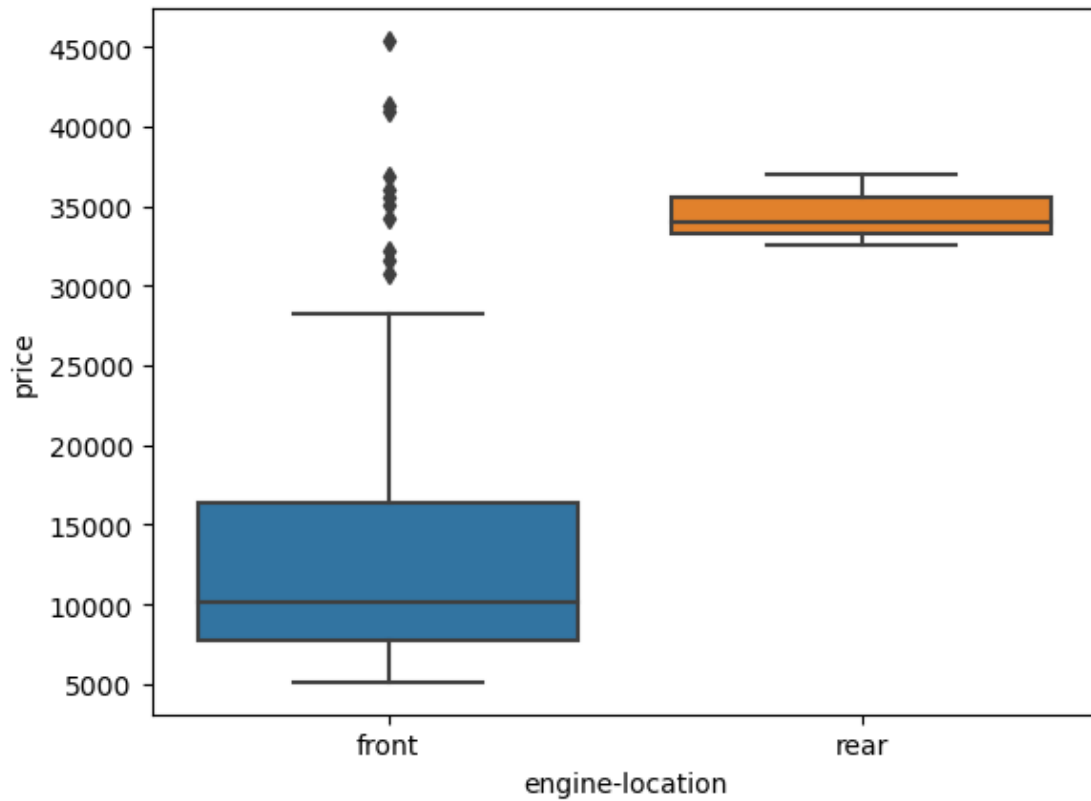
```
[90]: <Axes: xlabel='body-style', ylabel='price'>
```



We see that the distributions of price between the different body-style categories have a significant overlap, so body-style would not be a good predictor of price. Let's examine engine "engine-location" and "price":

```
[91]: sns.boxplot(x="engine-location", y="price", data=df)
```

```
[91]: <Axes: xlabel='engine-location', ylabel='price'>
```

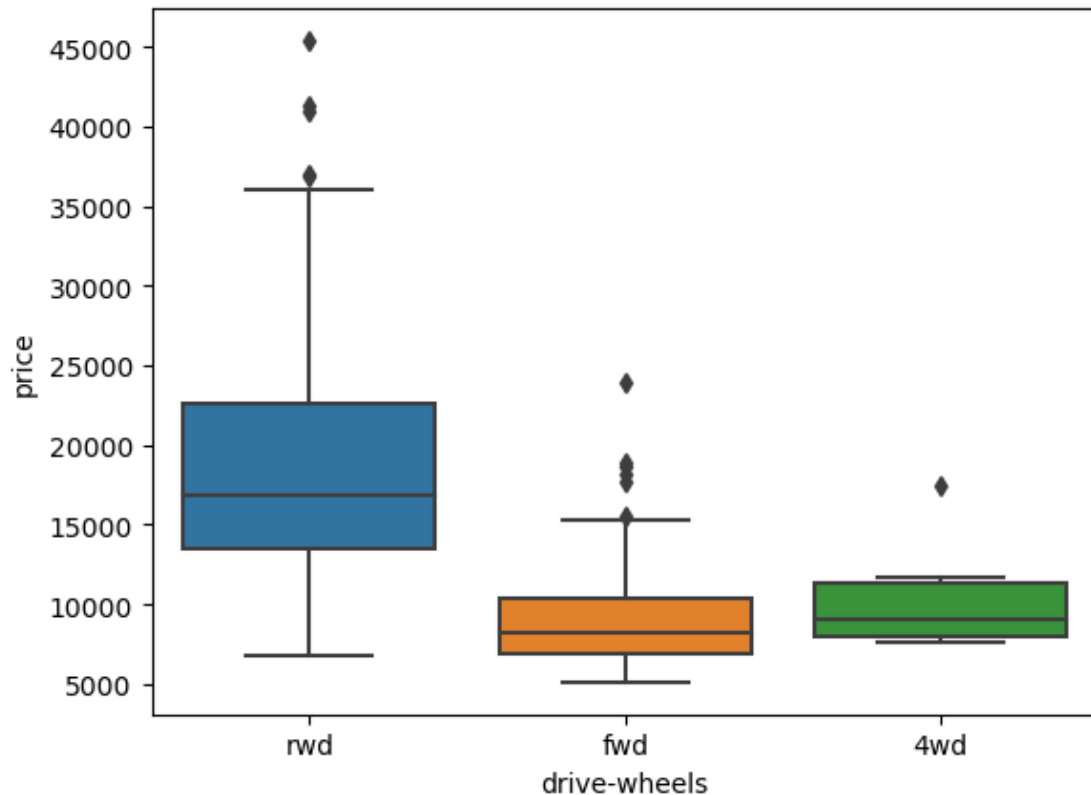


Here we see that the distribution of price between these two engine-location categories, front and rear, are distinct enough to take engine-location as a potential good predictor of price.

Let's examine "drive-wheels" and "price".

```
[92]: # drive-wheels
sns.boxplot(x="drive-wheels", y="price", data=df)
```

```
[92]: <Axes: xlabel='drive-wheels', ylabel='price'>
```

Here we see that the distribution of price between the different drive-wheels categories differs. As such, drive-wheels could potentially be a predictor of price.

6.4 Descriptive Statistical Analysis

Let's first take a look at the variables by utilizing a description method.

The describe function automatically computes basic statistics for all continuous variables. Any NaN values are automatically skipped in these statistics.

This will show:

the count of that variable

the mean

the standard deviation (std)

the minimum value

the IQR (Interquartile Range: 25%, 50% and 75%)

the maximum value

We can apply the method "describe" as follows:

```
[93]: df.describe()
```

```
[93]:      symboling  normalized-losses  wheel-base    length    width \
count  200.000000      200.000000  200.000000  200.000000  200.000000
mean    0.830000      122.000000   98.848000    0.837232    0.915250
std     1.248557      32.076542    6.038261    0.059333    0.029207
min    -2.000000      65.000000   86.600000    0.678039    0.837500
25%     0.000000     100.250000   94.500000    0.800937    0.891319
50%     1.000000     122.000000   97.000000    0.832292    0.909722
75%     2.000000     138.250000  102.400000    0.881788    0.926042
max      3.000000     256.000000  120.900000    1.000000    1.000000

      height  curb-weight  engine-size    bore    stroke \
count  200.000000  200.000000  200.000000  200.000000  200.000000
mean    0.899523  2555.705000  126.860000    3.330000    3.259847
std     0.040610   518.594552   41.650501    0.268562    0.314177
min     0.799331  1488.000000   61.000000    2.540000    2.070000
25%     0.869565  2163.000000   97.750000    3.150000    3.117500
50%     0.904682  2414.000000  119.500000    3.310000    3.290000
75%     0.928512  2928.250000  142.000000    3.582500    3.410000
max      1.000000  4066.000000  326.000000    3.940000    4.170000

      compression-ratio  horsepower    peak-rpm    city-mpg  highway-mpg \
count      200.000000  200.000000  200.000000  200.000000  200.000000
mean       10.170100  103.355000  5118.181818   25.200000   30.705000
std         4.014163   37.455487   479.240110    6.432487    6.827227
min         7.000000   48.000000  4150.000000   13.000000   16.000000
25%         8.575000   70.000000  4800.000000   19.000000   25.000000
50%         9.000000   95.000000  5159.090909   24.000000   30.000000
75%         9.400000  116.000000  5500.000000   30.000000   34.000000
max        23.000000  262.000000  6600.000000   49.000000   54.000000

      price  city-L/100km  highway-L/100km
count  200.000000      200.000000      200.000000
mean   13205.690000      9.937914      8.041663
std     7966.982558      2.539415      1.844764
min     5118.000000      4.795918      4.351852
25%     7775.000000      7.833333      6.911765
50%    10270.000000      9.791667      7.833333
75%    16500.750000     12.368421      9.400000
max    45400.000000     18.076923     14.687500
```

The default setting of “describe” skips variables of type object. We can apply the method “describe” on the variables of type ‘object’ as follows:

```
[94]: df.describe(include=['object'])
```

```
[94]:      make aspiration num-of-doors body-style drive-wheels \
count      200      200      200      200      200
unique      22       2       2       5       3
top    toyota      std      four      sedan      fwd
freq       32     164     115      94     118

      engine-location engine-type num-of-cylinders fuel-system \
count      200      200      200      200
unique       2       6       7       8
top        front      ohc      four      mpfi
freq       197     145     156     91

      horsepower-binned
count      200
unique       3
top        Low
freq       152
```

Value Counts

Value counts is a good way of understanding how many units of each characteristic/variable we have. We can apply the “value_counts” method on the column “drive-wheels”. Don’t forget the method “value_counts” only works on pandas series, not pandas dataframes. As a result, we only include one bracket df[‘drive-wheels’], not two brackets df[[‘drive-wheels’]].

```
[95]: df['drive-wheels'].value_counts()
```

```
[95]: drive-wheels
fwd    118
rwd     74
4wd     8
Name: count, dtype: int64
```

We can convert the series to a dataframe as follows:

```
[96]: df['drive-wheels'].value_counts().to_frame()
```

```
[96]:      count
drive-wheels
fwd      118
rwd       74
4wd        8
```

Let’s repeat the above steps but save the results to the dataframe “drive_wheels_counts” and rename the column ‘drive-wheels’ to ‘value_counts’.

```
[97]: drive_wheels_counts = df['drive-wheels'].value_counts().to_frame()
drive_wheels_counts.rename(columns={'drive-wheels': 'value_counts'},
↪ inplace=True)
```

```
drive_wheels_counts
```

```
[97]:          count
drive-wheels
fwd          118
rwd           74
4wd           8
```

Now let's rename the index to 'drive-wheels':

```
[98]: drive_wheels_counts.index.name = 'drive-wheels'
drive_wheels_counts
```

```
[98]:          count
drive-wheels
fwd          118
rwd           74
4wd           8
```

We can repeat the above process for the variable 'engine-location'.

```
[99]: # engine-location as variable
engine_loc_counts = df['engine-location'].value_counts().to_frame()
engine_loc_counts.rename(columns={'engine-location': 'value_counts'},
    inplace=True)
engine_loc_counts.index.name = 'engine-location'
engine_loc_counts.head(10)
```

```
[99]:          count
engine-location
front          197
rear             3
```

After examining the value counts of the engine location, we see that engine location would not be a good predictor variable for the price. This is because we only have three cars with a rear engine and 198 with an engine in the front, so this result is skewed. Thus, we are not able to draw any conclusions about the engine location.

6.5 Basics of Grouping

The “groupby” method groups data by different categories. The data is grouped based on one or several variables, and analysis is performed on the individual groups.

For example, let's group by the variable “drive-wheels”. We see that there are 3 different categories of drive wheels.

```
[100]: df['drive-wheels'].unique()
```

```
[100]: array(['rwd', 'fwd', '4wd'], dtype=object)
```

If we want to know, on average, which type of drive wheel is most valuable, we can group “drive-wheels” and then average them.

We can select the columns ‘drive-wheels’, ‘body-style’ and ‘price’, then assign it to the variable “df_group_one”.

```
[101]: df_group_one = df[['drive-wheels', 'body-style', 'price']]
```

We can then calculate the average price for each of the different categories of data.

```
[102]: # Assuming you want to analyze 'price' column based on 'drive-wheels'
df_group_one = df.groupby(['drive-wheels'], as_index=False)['price'].mean()
df_group_one
```

```
[102]:  drive-wheels      price
0         4wd  10241.000000
1         fwd   9244.779661
2         rwd  19842.243243
```

From our data, it seems rear-wheel drive vehicles are, on average, the most expensive, while 4-wheel and front-wheel are approximately the same in price.

You can also group by multiple variables. For example, let’s group by both ‘drive-wheels’ and ‘body-style’. This groups the dataframe by the unique combination of ‘drive-wheels’ and ‘body-style’. We can store the results in the variable ‘grouped_test1’.

```
[103]: # grouping results
df_gptest = df[['drive-wheels', 'body-style', 'price']]
grouped_test1 = df_gptest.groupby(['drive-wheels', 'body-style'], as_index=False).
    ↪mean()
grouped_test1
```

```
[103]:  drive-wheels  body-style      price
0         4wd    hatchback  7603.000000
1         4wd      sedan    12647.333333
2         4wd      wagon    9095.750000
3         fwd  convertible  11595.000000
4         fwd    hardtop    8249.000000
5         fwd    hatchback   8396.387755
6         fwd      sedan    9811.800000
7         fwd      wagon    9997.333333
8         rwd  convertible  26563.250000
9         rwd    hardtop   24202.714286
10        rwd    hatchback  14337.777778
11        rwd      sedan   21711.833333
12        rwd      wagon   16994.222222
```

This grouped data is much easier to visualize when it is made into a pivot table. A pivot table is like an Excel spreadsheet, with one variable along the column and another along the row. We can convert the dataframe to a pivot table using the method “pivot” to create a pivot table from the

groups.

In this case, we will leave the drive-wheels variable as the rows of the table, and pivot body-style to become the columns of the table:

```
[104]: grouped_pivot = grouped_test1.pivot(index='drive-wheels', columns='body-style')
grouped_pivot
```

```
[104]:
```

	price			
body-style	convertible	hardtop	hatchback	sedan
drive-wheels				
4wd	NaN	NaN	7603.000000	12647.333333
fwd	11595.00	8249.000000	8396.387755	9811.800000
rwd	26563.25	24202.714286	14337.777778	21711.833333

body-style	wagon
drive-wheels	
4wd	9095.750000
fwd	9997.333333
rwd	16994.222222

Often, we won't have data for some of the pivot cells. We can fill these missing cells with the value 0, but any other value could potentially be used as well. It should be mentioned that missing data is quite a complex subject and is an entire course on its own.

```
[105]: grouped_pivot = grouped_pivot.fillna(0) #fill missing values with 0
grouped_pivot
```

```
[105]:
```

	price			
body-style	convertible	hardtop	hatchback	sedan
drive-wheels				
4wd	0.00	0.000000	7603.000000	12647.333333
fwd	11595.00	8249.000000	8396.387755	9811.800000
rwd	26563.25	24202.714286	14337.777778	21711.833333

body-style	wagon
drive-wheels	
4wd	9095.750000
fwd	9997.333333
rwd	16994.222222

```
[106]: df_gptest2 = df[['body-style', 'price']]
grouped_test_bodystyle = df_gptest2.groupby(['body-style'], as_index=False).
    .mean()
grouped_test_bodystyle
```

```
[106]:      body-style      price
0  convertible  23569.600000
1    hardtop   22208.500000
2   hatchback   9957.441176
3      sedan   14459.755319
4      wagon   12371.960000
```

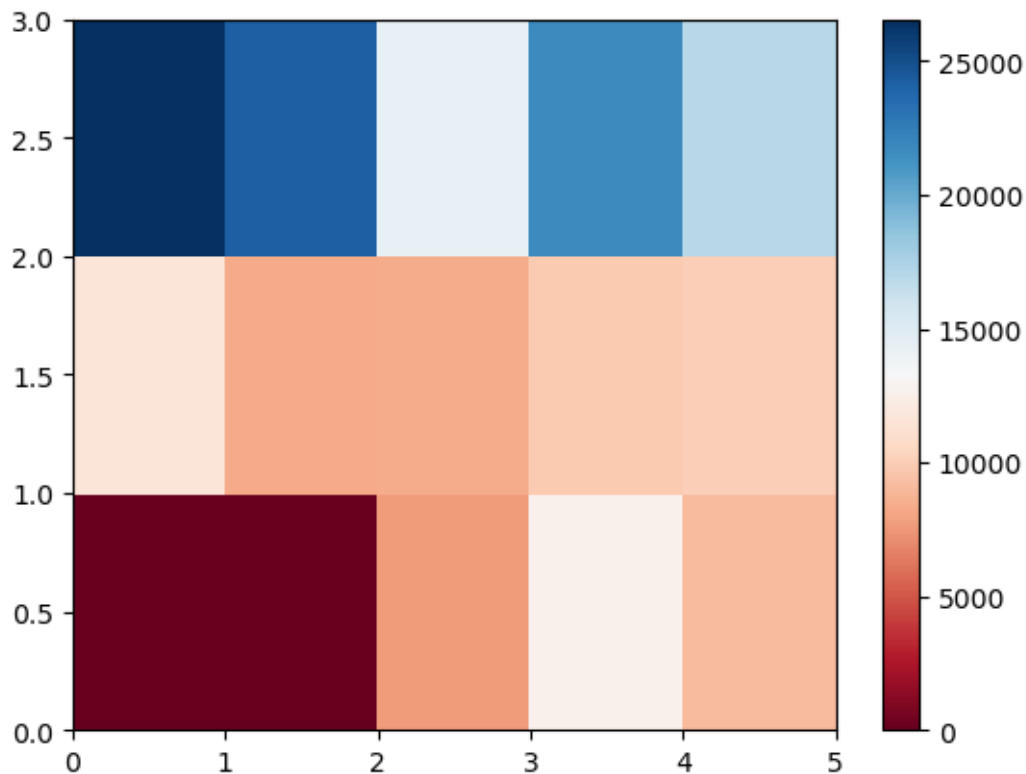
If you did not import “pyplot”, let’s do it again.

```
[107]: import matplotlib.pyplot as plt
      %matplotlib inline
```

Variables: Drive Wheels and Body Style vs. Price

Let’s use a heat map to visualize the relationship between Body Style vs Price.

```
[108]: #use the grouped results
plt.pcolor(grouped_pivot, cmap='RdBu')
plt.colorbar()
plt.show()
```



The heatmap plots the target variable (price) proportional to colour with respect to the variables ‘drive-wheel’ and ‘body-style’ on the vertical and horizontal axis, respectively. This allows us to visualize how the price is related to ‘drive-wheel’ and ‘body-style’.

The default labels convey no useful information to us. Let's change that:

```
[109]: fig, ax = plt.subplots()
im = ax.pcolor(grouped_pivot, cmap='RdBu')

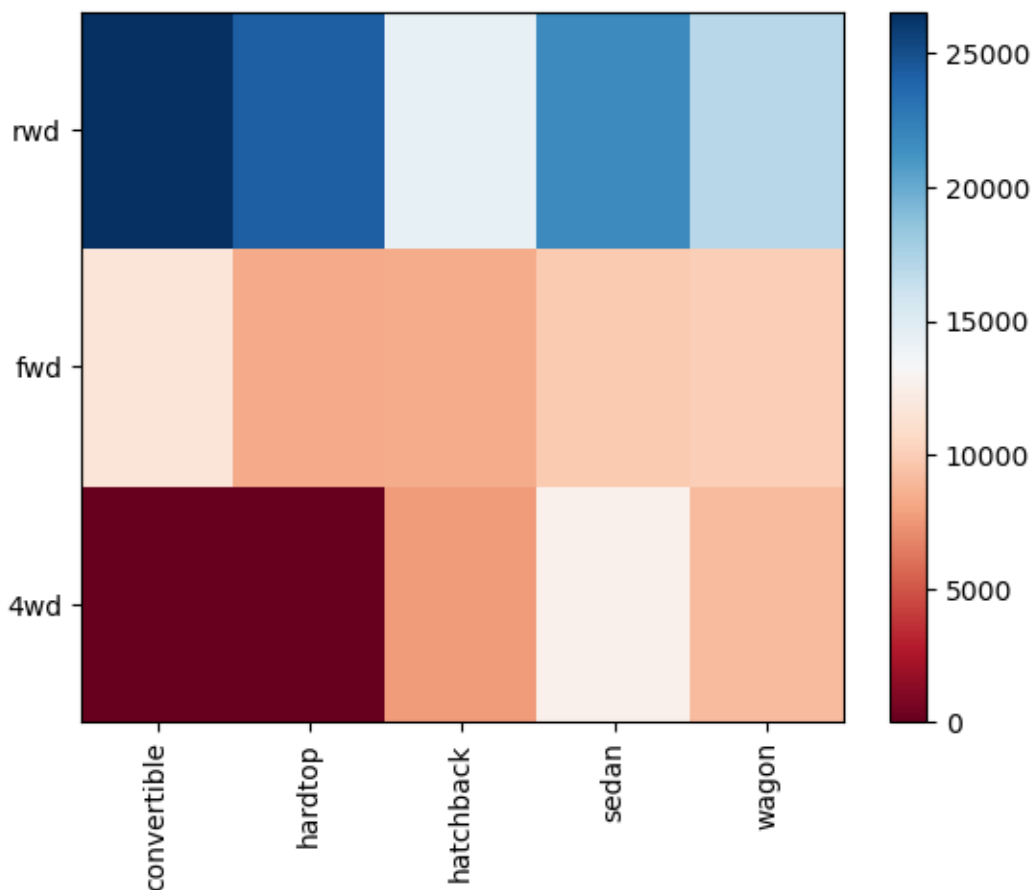
#label names
row_labels = grouped_pivot.columns.levels[1]
col_labels = grouped_pivot.index

#move ticks and labels to the center
ax.set_xticks(np.arange(grouped_pivot.shape[1]) + 0.5, minor=False)
ax.set_yticks(np.arange(grouped_pivot.shape[0]) + 0.5, minor=False)

#insert labels
ax.set_xticklabels(row_labels, minor=False)
ax.set_yticklabels(col_labels, minor=False)

#rotate label if too long
plt.xticks(rotation=90)

fig.colorbar(im)
plt.show()
```

Visualization is very important in data science, and Python visualization packages provide great freedom. We will go more in-depth in a separate Python visualizations course.

The main question we want to answer in this module is, “What are the main characteristics which have the most impact on the car price?”.

To get a better measure of the important characteristics, we look at the correlation of these variables with the car price. In other words: how is the car price dependent on this variable?

6.6 Correlation and Causation

Correlation: a measure of the extent of interdependence between variables.

Causation: the relationship between cause and effect between two variables.

It is important to know the difference between these two. Correlation does not imply causation. Determining correlation is much simpler the determining causation as causation may require independent experimentation.

Pearson Correlation

The Pearson Correlation measures the linear dependence between two variables X and Y.

The resulting coefficient is a value between -1 and 1 inclusive, where:

1: Perfect positive linear correlation.

0: No linear correlation, the two variables most likely do not affect each other.

-1: Perfect negative linear correlation.

Pearson Correlation is the default method of the function “corr”. Like before, we can calculate the Pearson Correlation of the of the ‘int64’ or ‘float64’ variables.

```
[110]: numeric_df = df.select_dtypes(include=['float64', 'int64'])
numeric_df.corr()
```

```
[110]:
```

	symboling	normalized-losses	wheel-base	length	\
symboling	1.000000	0.469772	-0.529145	-0.364511	
normalized-losses	0.469772	1.000000	-0.057068	0.019433	
wheel-base	-0.529145	-0.057068	1.000000	0.879005	
length	-0.364511	0.019433	0.879005	1.000000	
width	-0.237262	0.086961	0.814593	0.857271	
height	-0.542261	-0.377664	0.583789	0.492955	
curb-weight	-0.234743	0.099404	0.787584	0.881058	
engine-size	-0.112069	0.112362	0.576779	0.685531	
bore	-0.145667	-0.029867	0.501534	0.610817	
stroke	0.008244	0.055759	0.144675	0.120888	
compression-ratio	-0.181073	-0.114738	0.249689	0.159203	
horsepower	0.074581	0.217323	0.375732	0.580477	
peak-rpm	0.284011	0.239580	-0.364971	-0.286754	
city-mpg	-0.030158	-0.225255	-0.480029	-0.667658	
highway-mpg	0.041248	-0.182011	-0.552211	-0.700186	
price	-0.083327	0.133999	0.589147	0.691044	
city-L/100km	0.062423	0.238712	0.484047	0.659174	
highway-L/100km	-0.033159	0.181247	0.584953	0.708466	

	width	height	curb-weight	engine-size	bore	\
symboling	-0.237262	-0.542261	-0.234743	-0.112069	-0.145667	
normalized-losses	0.086961	-0.377664	0.099404	0.112362	-0.029867	
wheel-base	0.814593	0.583789	0.787584	0.576779	0.501534	
length	0.857271	0.492955	0.881058	0.685531	0.610817	
width	1.000000	0.300995	0.867720	0.731100	0.548478	
height	0.300995	1.000000	0.310660	0.076255	0.187794	
curb-weight	0.867720	0.310660	1.000000	0.849090	0.644532	
engine-size	0.731100	0.076255	0.849090	1.000000	0.572786	
bore	0.548478	0.187794	0.644532	0.572786	1.000000	
stroke	0.182855	-0.081273	0.168642	0.208004	-0.051087	
compression-ratio	0.189008	0.259526	0.156444	0.029005	0.002021	
horsepower	0.617032	-0.085725	0.758095	0.822656	0.566690	
peak-rpm	-0.247388	-0.315756	-0.279411	-0.256702	-0.267010	
city-mpg	-0.638155	-0.057087	-0.750390	-0.651002	-0.581365	

highway-mpg	-0.684700	-0.111568	-0.795515	-0.679877	-0.590753
price	0.752795	0.137284	0.834420	0.872337	0.543431
city-L/100km	0.677111	0.008923	0.785868	0.745337	0.554069
highway-L/100km	0.739845	0.088903	0.837217	0.783593	0.558759

	stroke	compression-ratio	horsepower	peak-rpm	\
symboling	0.008244	-0.181073	0.074581	0.284011	
normalized-losses	0.055759	-0.114738	0.217323	0.239580	
wheel-base	0.144675	0.249689	0.375732	-0.364971	
length	0.120888	0.159203	0.580477	-0.286754	
width	0.182855	0.189008	0.617032	-0.247388	
height	-0.081273	0.259526	-0.085725	-0.315756	
curb-weight	0.168642	0.156444	0.758095	-0.279411	
engine-size	0.208004	0.029005	0.822656	-0.256702	
bore	-0.051087	0.002021	0.566690	-0.267010	
stroke	1.000000	0.186761	0.100351	-0.066173	
compression-ratio	0.186761	1.000000	-0.214162	-0.436244	
horsepower	0.100351	-0.214162	1.000000	0.108161	
peak-rpm	-0.066173	-0.436244	0.108161	1.000000	
city-mpg	-0.040677	0.330897	-0.822397	-0.116308	
highway-mpg	-0.040282	0.267929	-0.804714	-0.059326	
price	0.083296	0.071176	0.809779	-0.101519	
city-L/100km	0.041470	-0.298898	0.889584	0.116510	
highway-L/100km	0.051148	-0.222957	0.840687	0.018183	

	city-mpg	highway-mpg	price	city-L/100km	\
symboling	-0.030158	0.041248	-0.083327	0.062423	
normalized-losses	-0.225255	-0.182011	0.133999	0.238712	
wheel-base	-0.480029	-0.552211	0.589147	0.484047	
length	-0.667658	-0.700186	0.691044	0.659174	
width	-0.638155	-0.684700	0.752795	0.677111	
height	-0.057087	-0.111568	0.137284	0.008923	
curb-weight	-0.750390	-0.795515	0.834420	0.785868	
engine-size	-0.651002	-0.679877	0.872337	0.745337	
bore	-0.581365	-0.590753	0.543431	0.554069	
stroke	-0.040677	-0.040282	0.083296	0.041470	
compression-ratio	0.330897	0.267929	0.071176	-0.298898	
horsepower	-0.822397	-0.804714	0.809779	0.889584	
peak-rpm	-0.116308	-0.059326	-0.101519	0.116510	
city-mpg	1.000000	0.972024	-0.687186	-0.949692	
highway-mpg	0.972024	1.000000	-0.705115	-0.929940	
price	-0.687186	-0.705115	1.000000	0.790291	
city-L/100km	-0.949692	-0.929940	0.790291	1.000000	
highway-L/100km	-0.909113	-0.951133	0.801313	0.958312	

	highway-L/100km
symboling	-0.033159

normalized-losses	0.181247
wheel-base	0.584953
length	0.708466
width	0.739845
height	0.088903
curb-weight	0.837217
engine-size	0.783593
bore	0.558759
stroke	0.051148
compression-ratio	-0.222957
horsepower	0.840687
peak-rpm	0.018183
city-mpg	-0.909113
highway-mpg	-0.951133
price	0.801313
city-L/100km	0.958312
highway-L/100km	1.000000

Sometimes we would like to know the significant of the correlation estimate.

P-value

What is this P-value? The P-value is the probability value that the correlation between these two variables is statistically significant. Normally, we choose a significance level of 0.05, which means that we are 95% confident that the correlation between the variables is significant.

By convention, when the

p-value is < 0.001 : we say there is strong evidence that the correlation is significant.

the p-value is < 0.05 : there is moderate evidence that the correlation is significant.

the p-value is < 0.1 : there is weak evidence that the correlation is significant.

the p-value is > 0.1 : there is no evidence that the correlation is significant.

We can obtain this information using “stats” module in the “scipy” library.

```
[111]: from scipy import stats
```

Wheel-Base vs. Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'wheel-base' and 'price'.

```
[112]: pearson_coef, p_value = stats.pearsonr(df['wheel-base'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value_
↳ of P =", p_value)
```

The Pearson Correlation Coefficient is 0.5891470005448705 with a P-value of P = 4.457019502050087e-20

Conclusion:

Since the p-value is < 0.001 , the correlation between wheel-base and price is statistically significant, although the linear relationship isn't extremely strong (~ 0.585).

Horsepower vs. Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'horsepower' and 'price'.

```
[113]: pearson_coef, p_value = stats.pearsonr(df['horsepower'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value)
```

The Pearson Correlation Coefficient is 0.8097789763551083 with a P-value of P = 9.887379251280244e-48

Conclusion:

Since the p-value is < 0.001 , the correlation between horsepower and price is statistically significant, and the linear relationship is quite strong (~ 0.809 , close to 1).

Length vs. Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'length' and 'price'.

```
[114]: pearson_coef, p_value = stats.pearsonr(df['length'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value)
```

The Pearson Correlation Coefficient is 0.6910440897821907 with a P-value of P = 9.960963222347273e-30

Conclusion:

Since the p-value is < 0.001 , the correlation between length and price is statistically significant, and the linear relationship is moderately strong (~ 0.691).

Width vs. Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'width' and 'price':

```
[115]: pearson_coef, p_value = stats.pearsonr(df['width'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value )
```

The Pearson Correlation Coefficient is 0.7527948631832613 with a P-value of P = 8.256714148307422e-38

Conclusion: Since the p-value is < 0.001 , the correlation between width and price is statistically significant, and the linear relationship is quite strong (~ 0.751).

6.6.1 Curb-Weight vs. Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'curb-weight' and 'price':

```
[116]: pearson_coef, p_value = stats.pearsonr(df['curb-weight'], df['price'])
print( "The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value)
```

The Pearson Correlation Coefficient is 0.8344204348498463 with a P-value of P = 3.9699775360213907e-53

Conclusion:

Since the p-value is < 0.001 , the correlation between curb-weight and price is statistically significant, and the linear relationship is quite strong (~ 0.834).

Engine-Size vs. Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'engine-size' and 'price':

```
[117]: pearson_coef, p_value = stats.pearsonr(df['engine-size'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value)
```

The Pearson Correlation Coefficient is 0.8723367498521142 with a P-value of P = 1.8977171466561833e-63

Conclusion:

Since the p-value is < 0.001 , the correlation between engine-size and price is statistically significant, and the linear relationship is very strong (~ 0.872).

Bore vs. Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'bore' and 'price':

```
[118]: pearson_coef, p_value = stats.pearsonr(df['bore'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value)
```

The Pearson Correlation Coefficient is 0.5434310033088079 with a P-value of P = 9.209749630850307e-17

Conclusion:

Since the p-value is < 0.001 , the correlation between bore and price is statistically significant, but the linear relationship is only moderate (~ 0.521).

We can relate the process for each 'city-mpg' and 'highway-mpg':

City-mpg vs. Price

```
[119]: pearson_coef, p_value = stats.pearsonr(df['city-mpg'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value)
```

The Pearson Correlation Coefficient is -0.6871861020862693 with a P-value of P = 2.729256568478666e-29

Conclusion:

Since the p-value is < 0.001 , the correlation between city-mpg and price is statistically significant, and the coefficient of about -0.687 shows that the relationship is negative and moderately strong.

Highway-mpg vs. Price

```
[120]: pearson_coef, p_value = stats.pearsonr(df['highway-mpg'], df['price'])
print( "The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value )
```

```
The Pearson Correlation Coefficient is -0.7051147088046401 with a P-value of P
= 2.1973260531584746e-31
```

Conclusion: Since the p-value is < 0.001 , the correlation between highway-mpg and price is statistically significant, and the coefficient of about -0.705 shows that the relationship is negative and moderately strong.

Conclusion: Important Variables

We now have a better idea of what our data looks like 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.

7 Model Development

Estimated time needed: **30** minutes

7.1 Objectives

- Develop prediction models

Some questions we want to ask in this module

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 often 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.

Setup

Import libraries:

```
[121]: #install specific version of libraries used in lab
      #! mamba install pandas==1.3.3-y
      #! mamba install numpy=1.21.2-y
      #! mamba install sklearn=0.20.1-y
```

```
[122]: '''import piplite
      await piplite.install('seaborn')'''
```

```
[122]: "import piplite\nawait piplite.install('seaborn')"
```

```
[123]: import pandas as pd
      import numpy as np
      import matplotlib.pyplot as plt
      import seaborn as sns
```

Load the data and store it in dataframe df:

This dataset was hosted on IBM Cloud object. Click [HERE](#) for free storage. Download it by running the cell below.

```
[124]: '''from pyodide.http import pyfetch

      async def download(url, filename):
          response = await pyfetch(url)
          if response.status == 200:
              with open(filename, "wb") as f:
                  f.write(await response.bytes())'''
```

```
[124]: 'from pyodide.http import pyfetch\n\nasync def download(url, filename):\n    response = await pyfetch(url)\n    if response.status == 200:\n        with\n        open(filename, "wb") as f:\n            f.write(await response.bytes())'
```



```
[125]: file_path= "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/
↳IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/
↳automobileEDA.csv"
```

```
#await download(file_path, "usedcars.csv")
file_name="usedCars.csv"
```

```
[126]: df = pd.read_csv(file_name)
df.head()
```

```
[126]:
```

	symboling	normalized-losses	make	aspiration	num-of-doors	\
0	3	122	alfa-romero	std	two	
1	1	122	alfa-romero	std	two	
2	2	164	audi	std	four	
3	2	164	audi	std	four	
4	2	122	audi	std	two	

	body-style	drive-wheels	engine-location	wheel-base	length	...	\
0	convertible	rwd	front	88.6	0.811148	...	
1	hatchback	rwd	front	94.5	0.822681	...	
2	sedan	fwd	front	99.8	0.848630	...	
3	sedan	4wd	front	99.4	0.848630	...	
4	sedan	fwd	front	99.8	0.851994	...	

	horsepower	peak-rpm	city-mpg	highway-mpg	price	city-L/100km	\
0	111	5000.0	21	27	16500.0	11.190476	
1	154	5000.0	19	26	16500.0	12.368421	
2	102	5500.0	24	30	13950.0	9.791667	
3	115	5500.0	18	22	17450.0	13.055556	
4	110	5500.0	19	25	15250.0	12.368421	

	highway-L/100km	horsepower-binned	fuel-type-diesel	fuel-type-gas
0	8.703704	Low	False	True
1	9.038462	Medium	False	True
2	7.833333	Low	False	True
3	10.681818	Low	False	True
4	9.400000	Low	False	True

[5 rows x 30 columns]

```
[127]: #filepath = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/
↳IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/
↳automobileEDA.csv"
#df = pd.read_csv(filepath, header=None)
```

1. 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.

$$Y : \text{Response Variable} \quad X : \text{Predictor Variables}$$

Linear Function

$$\hat{Y} = a + bX$$

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:

```
[128]: from sklearn.linear_model import LinearRegression
```

Create the linear regression object:

```
[129]: lm = LinearRegression()  
lm
```

```
[129]: LinearRegression()
```

How could “highway-mpg” help us predict car price?

For this example, we want to 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.

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

Fit the linear model using highway-mpg:

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

```
[131]: LinearRegression()
```

We can output a prediction:

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

```
[132]: array([16254.26934067, 17077.0977727 , 13785.78404458, 20368.41150083,
          17899.92620473])
```

What is the value of the intercept (a)?

```
[133]: lm.intercept_
```

```
[133]: 38470.63700549668
```

What is the value of the slope (b)?

```
[134]: lm.coef_
```

```
[134]: array([-822.82843203])
```

What is the final estimated linear model we get?

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

$$\hat{Y} = a + bX$$

Plugging in the actual values we get:

Price = 38423.31 - 821.73 x highway-mpg

Create a linear regression object called “lm1”.

```
[135]: lm1=LinearRegression()
lm1
```

```
[135]: LinearRegression()
```

Train the model using “engine-size” as the independent variable and “price” as the dependent variable?

```
[136]: # Write your code below and press Shift+Enter to execute
x=df[['engine-size']]
y=df[['price']]
lm1.fit(x,y)
yhat=lm1.predict(x)
yhat[0:5]
```

```
[136]: array([13729.63711709, 17400.60417954, 10225.53219385, 14730.80995231,
          14730.80995231])
```

Find the slope and intercept of the model.

Slope

```
[137]: lm1.coef_
```

```
[137]: array([166.8621392])
```

Intercept

```
[138]: # Write your code below and press Shift+Enter to execute
lm1.intercept_
```

```
[138]: -7962.4409791630915
```

Equation of the predicted line, can use x and yhat or “engine-size” or “price”.

```
[139]: # using X and Y
Yhat=-7963.34 + 166.86*X

Price=-7963.34 + 166.86*df['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, but these results can generalize to any integer:

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:

$$\hat{Y} = a + b_1X_1 + b_2X_2 + b_3X_3 + b_4X_4$$

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.

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

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

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

```
[141]: LinearRegression()
```

What is the value of the intercept(a)?

```
[142]: lm.intercept_
```

```
[142]: -15814.43913901131
```

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

```
[143]: lm.coef_
```

```
[143]: array([53.64350321,  4.70621169, 81.46397065, 36.26760488])
```

What is the final estimated linear model that we get?

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

$$\hat{Y} = a + b_1X_1 + b_2X_2 + b_3X_3 + b_4X_4$$

What is the linear function we get in this example?

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

Create and train a Multiple Linear Regression model “lm2” where the response variable is “price”, and the predictor variable is “normalized-losses” and “highway-mpg”.

```
[144]: lm2=LinearRegression()  
Z1 = df[['normalized-losses', 'highway-mpg']]  
lm2.fit(Z1,y)
```

```
[144]: LinearRegression()
```

Find the coefficient of the model.

```
[145]: # Write your code below and press Shift+Enter to execute  
lm2.coef_
```

```
[145]: array([ 1.45409594, -821.58496582])
```

2. 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:

```
[146]: # import the visualization package: seaborn
import seaborn as sns
%matplotlib inline
```

Regression Plot

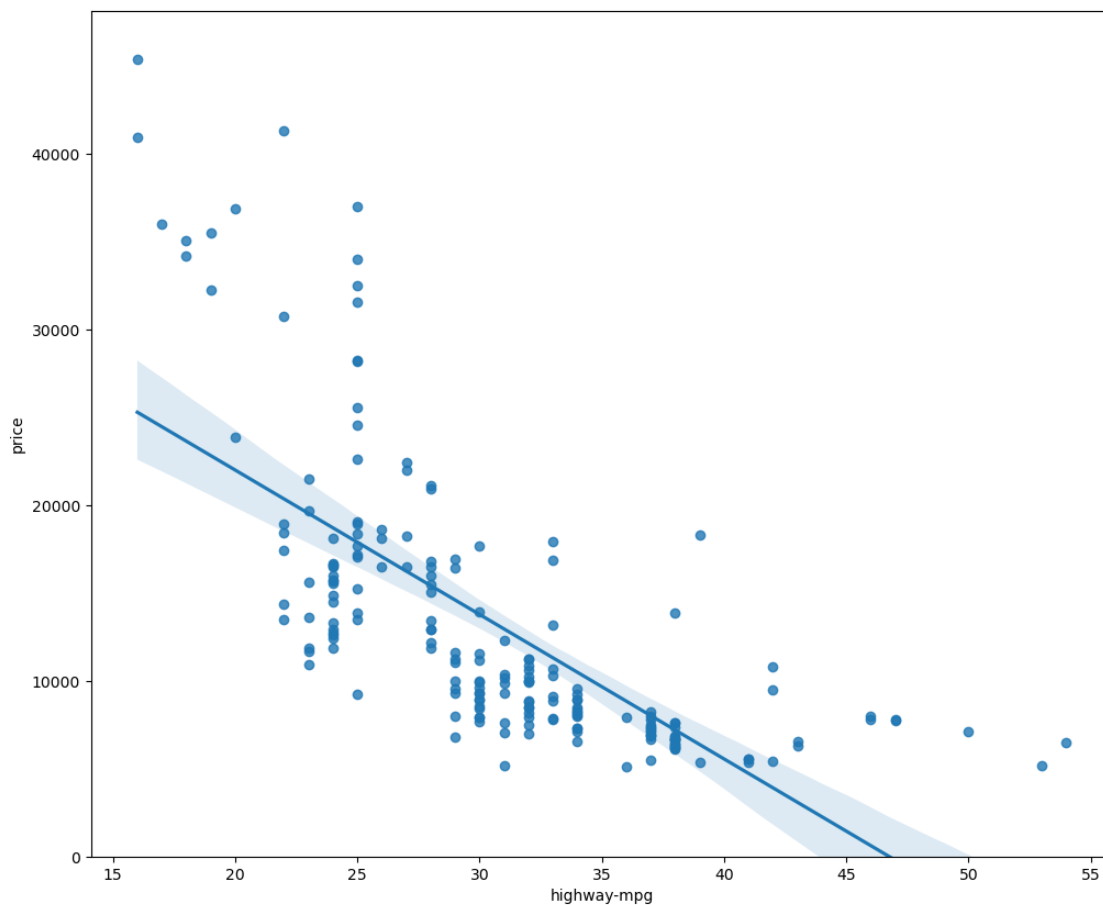
When it comes to simple linear regression, an excellent way to visualize the fit of our model is by using regression plots.

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:

```
[147]: width = 12
height = 10
plt.figure(figsize=(width, height))
sns.regplot(x="highway-mpg", y="price", data=df)
plt.ylim(0,)
```

```
[147]: (0.0, 48180.10936597729)
```



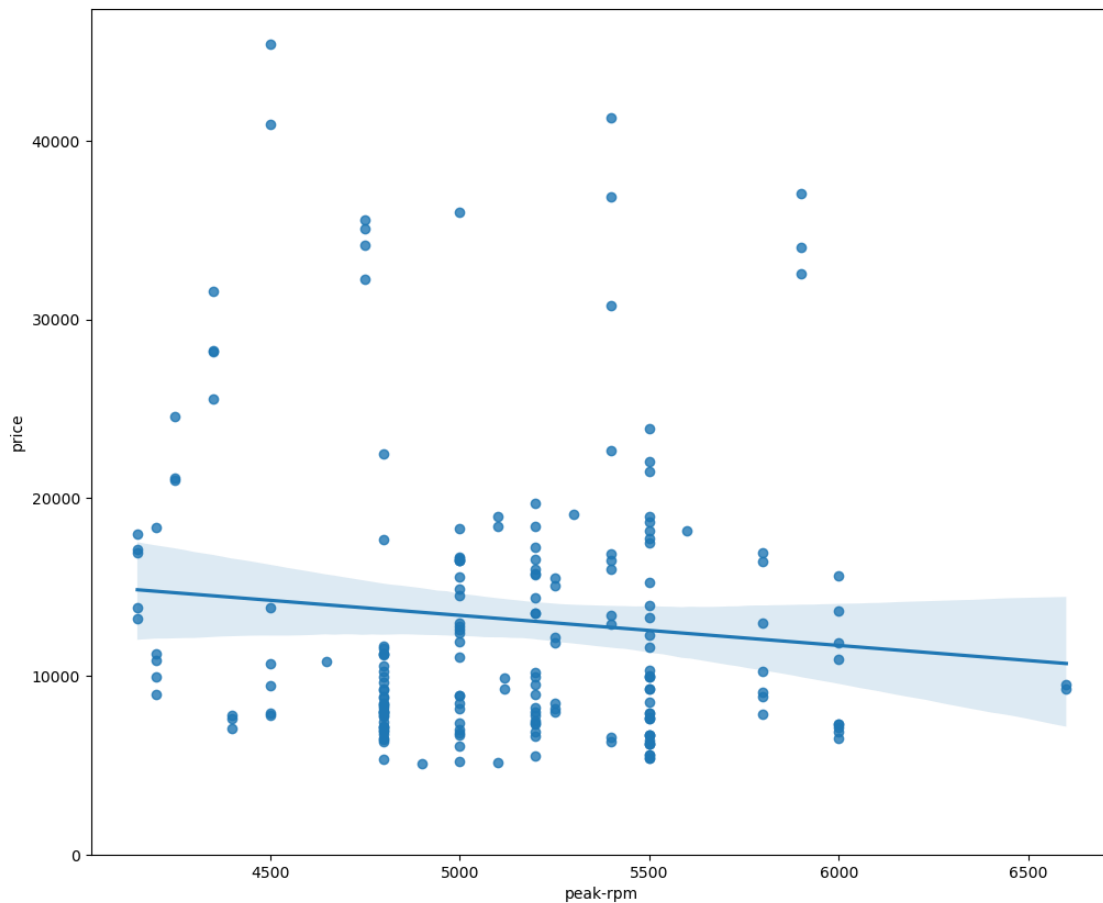
We can see from this plot that price is negatively correlated to highway-mpg since the regression slope is negative.

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".

```
[148]: plt.figure(figsize=(width, height))
sns.regplot(x="peak-rpm", y="price", data=df)
plt.ylim(0,)
```

```
[148]: (0.0, 47414.1)
```



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 “peak-rpm” increases.

Given the regression plots above, is “peak-rpm” or “highway-mpg” more strongly correlated with “price”? Can use the method “.corr()” to verify your answer.

```
[149]: # Write your code below and press Shift+Enter to execute
df[["peak-rpm", "highway-mpg", "price"]].corr()
```

```
[149]:
```

	peak-rpm	highway-mpg	price
peak-rpm	1.000000	-0.059326	-0.101519
highway-mpg	-0.059326	1.000000	-0.705115
price	-0.101519	-0.705115	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.

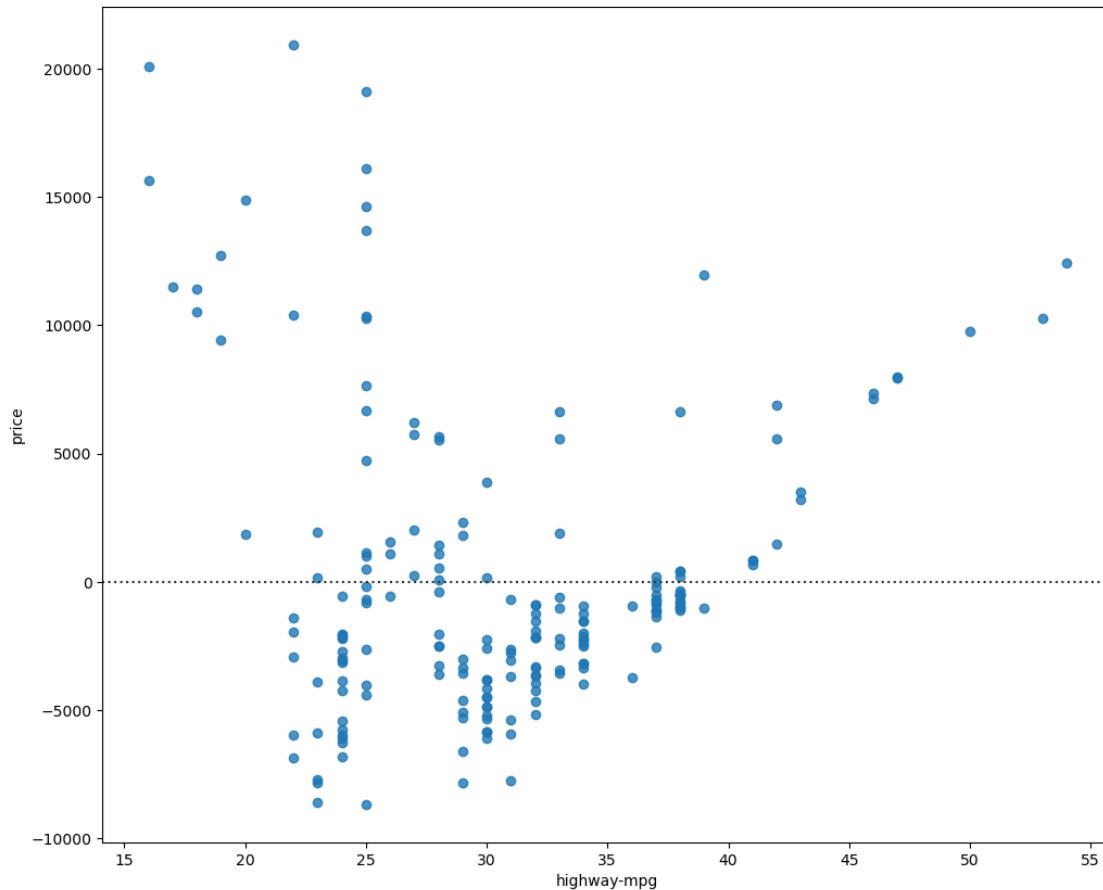
What do we pay attention to when looking at a residual plot?

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.

```
[150]: width = 12
height = 10
plt.figure(figsize=(width, height))
sns.residplot(x=df['highway-mpg'], y=df['price'])
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 distribution plot. 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:

```
[151]: Y_hat = lm.predict(Z)
```

```
[152]: plt.figure(figsize=(width, height))
```

```
ax1 = sns.distplot(df['price'], hist=False, color="r", label="Actual Value")
```

```
sns.distplot(Y_hat, hist=False, color="b", label="Fitted Values" , ax=ax1)

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

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

C:\Users\DHESIKA\AppData\Local\Temp\ipykernel_12304\4196657742.py:4:

UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).

For a guide to updating your code to use the new functions, please see

<https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751>

```
ax1 = sns.distplot(df['price'], hist=False, color="r", label="Actual Value")
```

C:\Users\DHESIKA\AppData\Local\Temp\ipykernel_12304\4196657742.py:5:

UserWarning:

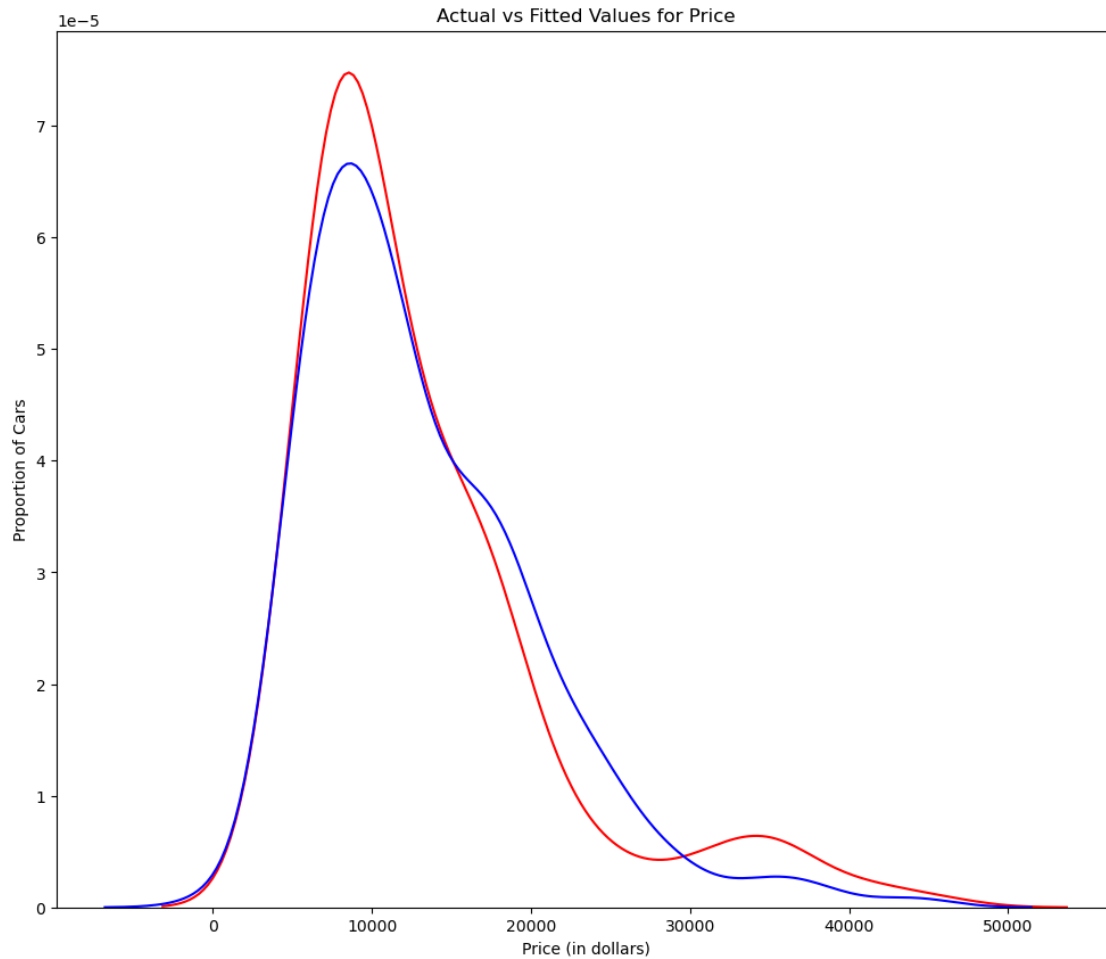
`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).

For a guide to updating your code to use the new functions, please see

<https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751>

```
sns.distplot(Y_hat, hist=False, color="b", label="Fitted Values" , ax=ax1)
```



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.

3. Polynomial Regression and Pipelines

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

$$\hat{Y} = a + b_1X + b_2X^2$$

Cubic - 3rd Order

$$\hat{Y} = a + b_1X + b_2X^2 + b_3X^3$$

Higher-Order:

$$Y = a + b_1X + b_2X^2 + b_3X^3 \dots$$

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:

```
[153]: def PlotPolly(model, independent_variable, dependent_variabble, Name):
        x_new = np.linspace(15, 55, 100)
        y_new = model(x_new)

        plt.plot(independent_variable, dependent_variabble, '.', x_new, y_new, '-')
        plt.title('Polynomial Fit with Matplotlib for Price ~ Length')
        ax = plt.gca()
        ax.set_facecolor((0.898, 0.898, 0.898))
        fig = plt.gcf()
        plt.xlabel(Name)
        plt.ylabel('Price of Cars')

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

Let’s get the variables:

```
[154]: x = df['highway-mpg']
        y = df['price']
```

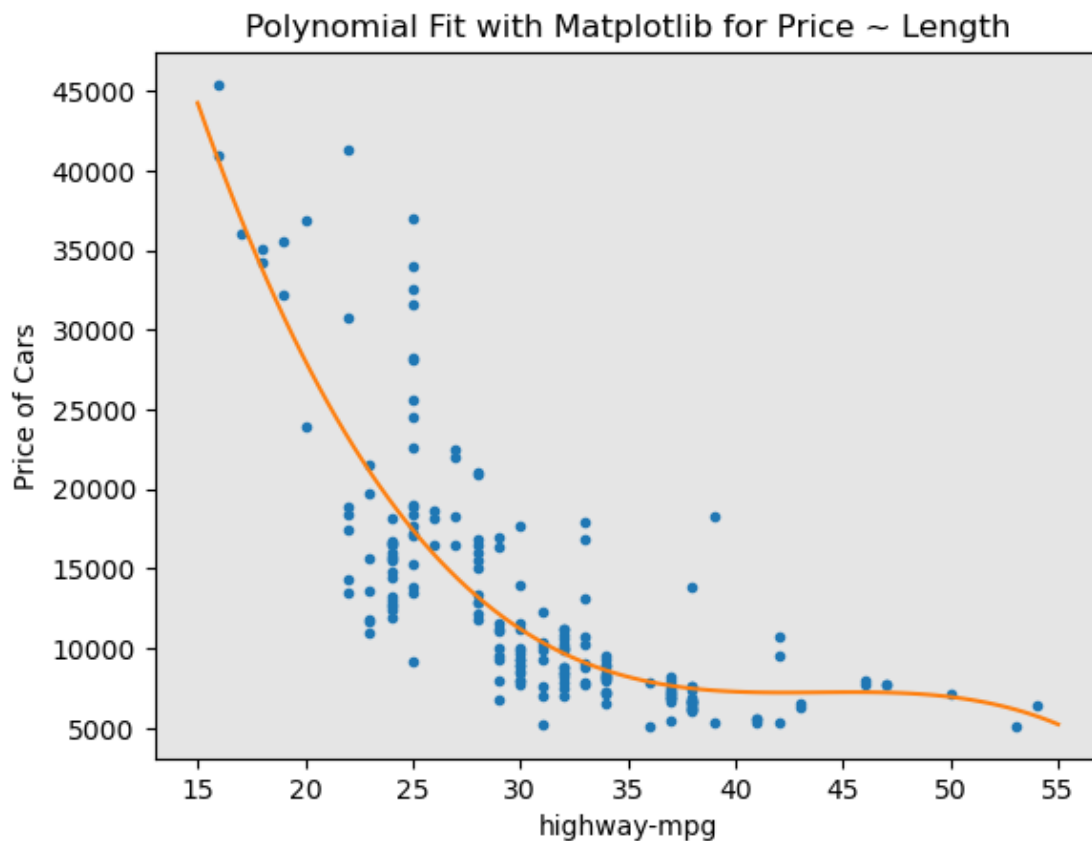
Let’s fit the polynomial using the function polyfit, then use the function poly1d to display the polynomial function.

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

```
          3          2
-1.552 x + 204.2 x - 8948 x + 1.378e+05
```

Let’s plot the function:

```
[156]: PlotPolly(p, x, y, 'highway-mpg')
```



```
[157]: np.polyfit(x, y, 3)
```

```
[157]: array([-1.55173297e+00,  2.04232144e+02, -8.94817574e+03,  1.37751367e+05])
```

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.

Create 11 order polynomial model with the variables x and y from above.

```
[158]: # Write your code below and press Shift+Enter to execute
x = df['highway-mpg']
y = df['price']
f = np.polyfit(x, y, 11)
p = np.poly1d(f)
print(p)
```

```

      11      10      9      8      7
-1.273e-08 x  + 4.839e-06 x  - 0.0008229 x + 0.08259 x - 5.432 x
      6      5      4      3      2
+ 245.6 x - 7786 x + 1.729e+05 x - 2.634e+06 x + 2.62e+07 x - 1.532e+08 x +
3.987e+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:

$$\hat{Y} = a + b_1X_1 + b_2X_2 + b_3X_1X_2 + b_4X_1^2 + b_5X_2^2$$

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

```
[159]: from sklearn.preprocessing import PolynomialFeatures
```

We create a PolynomialFeatures object of degree 2:

```
[160]: pr=PolynomialFeatures(degree=2)
pr
```

```
[160]: PolynomialFeatures()
```

```
[161]: Z_pr=pr.fit_transform(Z)
```

In the original data, there are 200 samples and 4 features.

```
[162]: Z.shape
```

```
[162]: (200, 4)
```

After the transformation, there are 200 samples and 15 features.

```
[163]: Z_pr.shape
```

```
[163]: (200, 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.

```
[164]: 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 corresponding constructor.

```
[165]: Input=[('scale',StandardScaler()), ('polynomial',PolynomialFeatures(include_bias=False)), ('model',LinearRegression())]
```

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

```
[166]: pipe=Pipeline(Input)
pipe
```

```
[166]: Pipeline(steps=[('scale', StandardScaler()),
                      ('polynomial', PolynomialFeatures(include_bias=False)),
```

```
('model', LinearRegression())])
```

First, we convert the data type Z to type float to avoid conversion warnings that may appear as a result of StandardScaler taking float inputs.

Then, we can normalize the data, perform a transform and fit the model simultaneously.

```
[167]: Z = Z.astype(float)
pipe.fit(Z,y)
```

```
[167]: Pipeline(steps=[('scale', StandardScaler()),
                      ('polynomial', PolynomialFeatures(include_bias=False)),
                      ('model', LinearRegression())])
```

Similarly, we can normalize the data, perform a transform and produce a prediction simultaneously.

```
[168]: ypipe=pipe.predict(Z)
ypipe[0:4]
```

```
[168]: array([13095.64294486, 18226.1683919 , 10389.2689322 , 16122.24836083])
```

Create a pipeline that standardizes the data, then produce a prediction using a linear regression model using the features Z and target y.

```
[169]: # Write your code below and press Shift+Enter to execute
Input=[("scale",StandardScaler()),("model",LinearRegression())]
pipe=Pipeline(Input)
Z = Z.astype(float)
pipe.fit(Z,y)
ypipe=pipe.predict(Z)
ypipe[0:4]
```

```
[169]: array([13700.95861278, 19057.77721438, 10623.21584883, 15521.89285072])
```

4. Measures for In-Sample Evaluation

When evaluating our models, not only do we want to visualize the results, but we also want a quantitative measure to determine how accurate the model is.

Two very important measures that are often used in Statistics to determine the accuracy of a model are:

R^2 / R-squared

Mean Squared Error (MSE)

R-squared

R squared, also known as the coefficient of determination, is a measure to indicate how close the data is to the fitted regression line.

The value of the R-squared is the percentage of variation of the response variable (y) that is explained by a linear model.

Mean Squared Error (MSE)

The Mean Squared Error measures the average of the squares of errors. That is, the difference between actual value (y) and the estimated value (\hat{y}).

Model 1: Simple Linear Regression

Let's calculate the R^2 :

```
[170]: #highway_mpg_fit
lm.fit(X, Y)
# Find the  $R^2$ 
print('The R-square is: ', lm.score(X, Y))
```

The R-square is: 0.49718675257265277

We can say that ~49% of the variation of the price is explained by this simple linear model “horse-power_fit”.

Let's calculate the MSE:

We can predict the output i.e., “yhat” using the predict method, where X is the input variable:

```
[171]: Yhat=lm.predict(X)
print('The output of the first four predicted value is: ', Yhat[0:4])
```

The output of the first four predicted value is: [16254.26934067 17077.0977727 13785.78404458 20368.41150083]

Let's import the function mean_squared_error from the module metrics:

```
[172]: from sklearn.metrics import mean_squared_error
```

We can compare the predicted results with the actual results:

```
[173]: mse = mean_squared_error(df['price'], Yhat)
print('The mean square error of price and predicted value is: ', mse)
```

The mean square error of price and predicted value is: 31755395.41081295

Model 2: Multiple Linear Regression

Let's calculate the R^2 :

```
[174]: # fit the model
lm.fit(Z, df['price'])
# Find the  $R^2$ 
print('The R-square is: ', lm.score(Z, df['price']))
```

The R-square is: 0.8094411114508352

We can say that ~80 % of the variation of price is explained by this multiple linear regression “multi_fit”.

Let's calculate the MSE.

We produce a prediction:

```
[175]: Y_predict_multifit = lm.predict(Z)
```

We compare the predicted results with the actual results:

```
[176]: print('The mean square error of price and predicted value using multifit is: ',  
↪\n      mean_squared_error(df['price'], Y_predict_multifit))
```

The mean square error of price and predicted value using multifit is:
12034831.790700043

Model 3: Polynomial Fit

Let's calculate the R^2 .

Let's import the function `r2_score` from the module `metrics` as we are using a different function.

```
[177]: from sklearn.metrics import r2_score
```

We apply the function to get the value of R^2 :

```
[178]: r_squared = r2_score(y, p(x))  
print('The R-square value is: ', r_squared)
```

The R-square value is: 0.7032923281262173

We can say that ~70 % of the variation of price is explained by this polynomial fit.

MSE

We can also calculate the MSE:

```
[179]: mean_squared_error(df['price'], p(x))
```

```
[179]: 18738705.652609386
```

5. Prediction and Decision Making

Prediction

In the previous section, we trained the model using the method `fit`. Now we will use the method `predict` to produce a prediction. Lets import `pyplot` for plotting; we will also be using some functions from `numpy`.

```
[180]: import matplotlib.pyplot as plt  
import numpy as np  
  
%matplotlib inline
```

Create a new input:

```
[181]: new_input=np.arange(1, 100, 1).reshape(-1, 1)
```

Fit the model:

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

```
[182]: LinearRegression()
```

Produce a prediction:

```
[183]: yhat=lm.predict(new_input)
yhat[0:5]
```

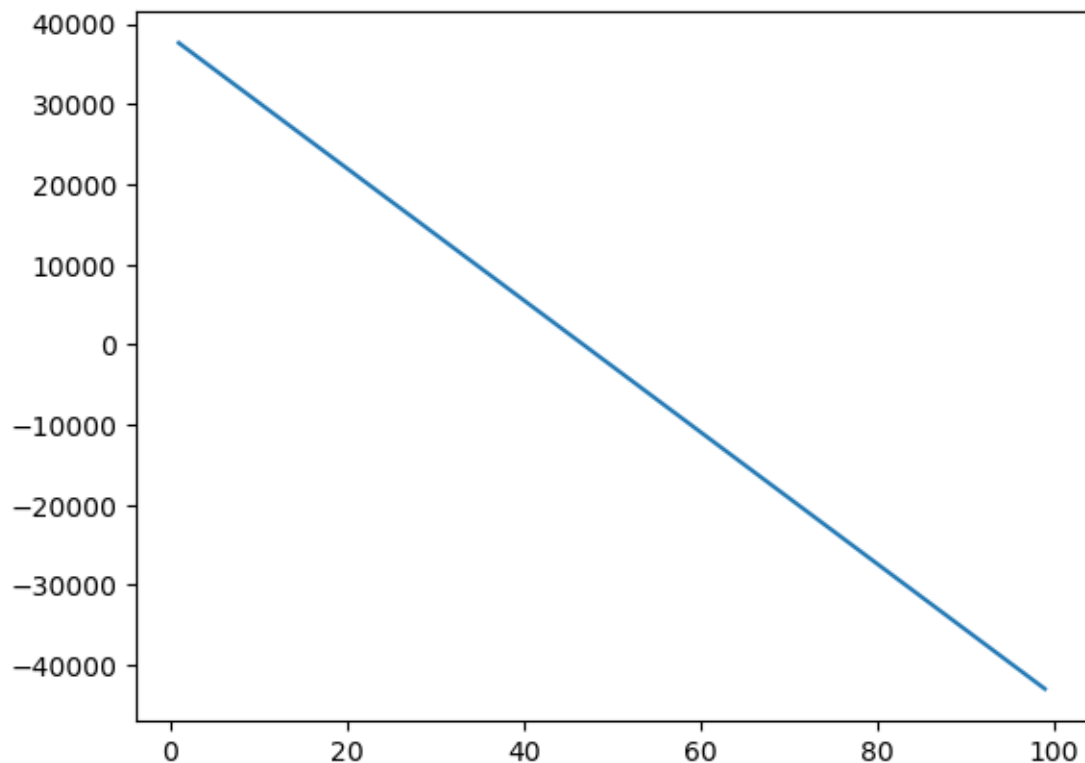
D:\Users\DHESIKA\anaconda3\Lib\site-packages\sklearn\base.py:464: UserWarning: X does not have valid feature names, but LinearRegression was fitted with feature names

```
warnings.warn(
```

```
[183]: array([37647.80857347, 36824.98014144, 36002.15170941, 35179.32327737,
34356.49484534])
```

We can plot the data:

```
[184]: plt.plot(new_input, yhat)
plt.show()
```



Decision Making: Determining a Good Model Fit

Now that we have visualized the different models, and generated the R-squared and MSE values for the fits, how do we determine a good model fit?

What is a good R-squared value?

When comparing models, the model with the higher R-squared value is a better fit for the data.

What is a good MSE?

When comparing models, the model with the smallest MSE value is a better fit for the data.

Let's take a look at the values for the different models.

Simple Linear Regression: Using Highway-mpg as a Predictor Variable of Price.

R-squared: 0.497

MSE: 3.17×10^7

Multiple Linear Regression: Using Horsepower, Curb-weight, Engine-size, and Highway-mpg as Predictor Variables of Price.

R-squared: 0.809

MSE: 1.2×10^7

Polynomial Fit: Using Highway-mpg as a Predictor Variable of Price.

R-squared: 0.703

MSE: 1.87×10^7

Simple Linear Regression Model (SLR) vs Multiple Linear Regression Model (MLR)

Usually, the more variables you have, the better your model is at predicting, but this is not always true. Sometimes you may not have enough data, you may run into numerical problems, or many of the variables may not be useful and even act as noise. As a result, you should always check the MSE and R^2 .

In order to compare the results of the MLR vs SLR models, we look at a combination of both the R-squared and MSE to make the best conclusion about the fit of the model.

MSE: The MSE of SLR is 3.17×10^7 while MLR has an MSE of 1.2×10^7 . The MSE of MLR is much smaller.

R-squared: In this case, we can also see that there is a big difference between the R-squared of the SLR and the R-squared of the MLR. The R-squared for the SLR (~ 0.49) is very small compared to the R-squared for the MLR (~ 0.80).

This R-squared in combination with the MSE show that MLR seems like the better model fit in this case compared to SLR.

Simple Linear Model (SLR) vs. Polynomial Fit

MSE: We can see that Polynomial Fit brought down the MSE, since this MSE is smaller than the one from the SLR.

R-squared: The R-squared for the Polynomial Fit is larger than the R-squared for the SLR, so the Polynomial Fit also brought up the R-squared quite a bit.

Since the Polynomial Fit resulted in a lower MSE and a higher R-squared, we can conclude that this was a better fit model than the simple linear regression for predicting “price” with “highway-mpg” as a predictor variable.

Multiple Linear Regression (MLR) vs. Polynomial Fit

MSE: The MSE for the MLR is smaller than the MSE for the Polynomial Fit.

R-squared: The R-squared for the MLR is also larger than for the Polynomial Fit.

Conclusion

Comparing these three models, we conclude that the MLR model is the best model to be able to predict price from our dataset. This result makes sense since we have 27 variables in total and we know that more than one of those variables are potential predictors of the final car price.

8 Model Evaluation and Refinement

Estimated time needed: **30** minutes

8.1 Objectives

After completing this lab you will be able to:

- Evaluate and refine prediction models

Table of Contents

Model Evaluation

Over-fitting, Under-fitting and Model Selection

Ridge Regression

Grid Search

If you are running the lab in your browser in Skills Network lab, so need to install the libraries using piplite.

```
[185]: #you are running the lab in your browser, so we will install the libraries
      ↪using ``piplite``
      '''import piplite
      await piplite.install(['pandas'])
      await piplite.install(['matplotlib'])
      await piplite.install(['scipy'])
      await piplite.install(['scikit-learn'])
      await piplite.install(['seaborn'])'''
```

```
[185]: "import piplite\nawait piplite.install(['pandas'])\nawait\npiplite.install(['matplotlib'])\nawait piplite.install(['scipy'])\nawait\npiplite.install(['scikit-learn'])\nawait piplite.install(['seaborn'])"
```

If you run the lab locally using Anaconda, you can load the correct library and versions by uncommenting the following:

```
[186]: #If you run the lab locally using Anaconda, you can load the correct library
        ↪and versions by uncommenting the following:
        #install specific version of libraries used in lab
        #! mamba install pandas==1.3.3-y
        #! mamba install numpy=1.21.2-y
        #! mamba install sklearn=0.20.1-y
```

Import libraries:

```
[187]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
```

This function will download the dataset into your browser

```
[188]: #This function will download the dataset into your browser

'''from pyodide.http import pyfetch

async def download(url, filename):
    response = await pyfetch(url)
    if response.status == 200:
        with open(filename, "wb") as f:
            f.write(await response.bytes())'''
```

```
[188]: 'from pyodide.http import pyfetch\n\nasync def download(url, filename):\n
response = await pyfetch(url)\n    if response.status == 200:\n        with
open(filename, "wb") as f:\n            f.write(await response.bytes())'
```

This dataset was hosted on IBM Cloud object. Click [HERE](#) for free storage.

you will need to download the dataset; using the 'download()' function.

```
[189]: #you will need to download the dataset;
#await download('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.
        ↪cloud/IBMDeveloperSkillsNetwork-DAO101EN-SkillsNetwork/labs/Data%20files/
        ↪module_5_auto.csv', 'module_5_auto.csv')
```

Load the data and store it in dataframe df:

```
[190]: df = pd.read_csv("usedcarsfinal.csv", header=0)
```

```
[191]: df.head()
```

```
[191]:
```

	symboling	normalized-losses	make	aspiration	num-of-doors	\
0	3	122	alfa-romero	std	two	
1	3	122	alfa-romero	std	two	
2	1	122	alfa-romero	std	two	

3	2	164	audi	std	four
4	2	164	audi	std	four

	body-style	drive-wheels	engine-location	wheel-base	length	...	\
0	convertible	rwd	front	88.6	0.811148	...	
1	convertible	rwd	front	88.6	0.811148	...	
2	hatchback	rwd	front	94.5	0.822681	...	
3	sedan	fwd	front	99.8	0.848630	...	
4	sedan	4wd	front	99.4	0.848630	...	

	compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg	price	\
0	9.0	111.0	5000.0	21	27	13495.0	
1	9.0	111.0	5000.0	21	27	16500.0	
2	9.0	154.0	5000.0	19	26	16500.0	
3	10.0	102.0	5500.0	24	30	13950.0	
4	8.0	115.0	5500.0	18	22	17450.0	

	city-L/100km	horsepower-binned	diesel	gas
0	11.190476	Medium	0	1
1	11.190476	Medium	0	1
2	12.368421	Medium	0	1
3	9.791667	Medium	0	1
4	13.055556	Medium	0	1

[5 rows x 29 columns]

First, let's only use numeric data:

```
[192]: df=df._get_numeric_data()
df.head()
```

```
[192]:
```

	symboling	normalized-losses	wheel-base	length	width	height	\
0	3	122	88.6	0.811148	0.890278	48.8	
1	3	122	88.6	0.811148	0.890278	48.8	
2	1	122	94.5	0.822681	0.909722	52.4	
3	2	164	99.8	0.848630	0.919444	54.3	
4	2	164	99.4	0.848630	0.922222	54.3	

	curb-weight	engine-size	bore	stroke	compression-ratio	horsepower	\
0	2548	130	3.47	2.68	9.0	111.0	
1	2548	130	3.47	2.68	9.0	111.0	
2	2823	152	2.68	3.47	9.0	154.0	
3	2337	109	3.19	3.40	10.0	102.0	
4	2824	136	3.19	3.40	8.0	115.0	

	peak-rpm	city-mpg	highway-mpg	price	city-L/100km	diesel	gas
0	5000.0	21	27	13495.0	11.190476	0	1
1	5000.0	21	27	16500.0	11.190476	0	1

2	5000.0	19	26	16500.0	12.368421	0	1
3	5500.0	24	30	13950.0	9.791667	0	1
4	5500.0	18	22	17450.0	13.055556	0	1

```
[193]: df.columns
```

```
[193]: Index(['symboling', 'normalized-losses', 'wheel-base', 'length', 'width',
          'height', 'curb-weight', 'engine-size', 'bore', 'stroke',
          'compression-ratio', 'horsepower', 'peak-rpm', 'city-mpg',
          'highway-mpg', 'price', 'city-L/100km', 'diesel', 'gas'],
          dtype='object')
```

Libraries for plotting:

```
[194]: #df.drop("highway-L/100km",axis=1,inplace=True)
```

```
[195]: from ipywidgets import interact, interactive, fixed, interact_manual
```

Functions for Plotting

```
[196]: def DistributionPlot(RedFunction, BlueFunction, RedName, BlueName, Title):
        width = 12
        height = 10
        plt.figure(figsize=(width, height))

        ax1 = sns.kdeplot(RedFunction, color="r", label=RedName)
        ax2 = sns.kdeplot(BlueFunction, color="b", label=BlueName, ax=ax1)

        plt.title(Title)
        plt.xlabel('Price (in dollars)')
        plt.ylabel('Proportion of Cars')
        plt.show()
        plt.close()
```

```
[197]: def PollyPlot(xtrain, xtest, y_train, y_test, lr, poly_transform):
        width = 12
        height = 10
        plt.figure(figsize=(width, height))

        #training data
        #testing data
        # lr: linear regression object
        #poly_transform: polynomial transformation object

        xmax=max([xtrain.values.max(), xtest.values.max()])

        xmin=min([xtrain.values.min(), xtest.values.min()])
```

```

x=np.arange(xmin, xmax, 0.1)

plt.plot(xtrain, y_train, 'ro', label='Training Data')
plt.plot(xtest, y_test, 'go', label='Test Data')
plt.plot(x, lr.predict(poly_transform.fit_transform(x.reshape(-1, 1))),
↪label='Predicted Function')
plt.ylim([-10000, 60000])
plt.ylabel('Price')
plt.legend()

```

Part 1: Training and Testing

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

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

Drop price data in dataframe `x_data`:

```
[199]: x_data=df.drop('price',axis=1)
```

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

```
[200]: 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.
↪10, 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 : 21
number of training samples: 180

```

The `test_size` parameter sets the proportion of data that is split into the testing set. In the above, the testing set is 10% of the total dataset.

Use the function “`train_test_split`” to split up the dataset such that 40% of the data samples will be utilized for testing. Set the parameter “`random_state`” equal to zero. The output of the function should be the following: “`x_train1`”, “`x_test1`”, “`y_train1`” and “`y_test1`”.

```
[201]: x_train1, x_test1, y_train1, y_test1 = train_test_split(x_data, y_data,
↪test_size=0.4, random_state=0)
print("number of test samples :", x_test1.shape[0])
print("number of training samples:",x_train1.shape[0])

```


number of test samples : 81
number of training samples: 120

Let's import LinearRegression from the module linear_model.

```
[202]: from sklearn.linear_model import LinearRegression
```

We create a Linear Regression object:

```
[203]: lre=LinearRegression()
```

We fit the model using the feature "horsepower":

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

```
[204]: LinearRegression()
```

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

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

```
[205]: 0.36358755750788263
```

We can see the R^2 is smaller using the test data compared to the training data.

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

```
[206]: 0.6619724197515104
```

Find the R^2 on the test data using 40% of the dataset for testing.

```
[207]: x_train1, x_test1, y_train1, y_test1 = train_test_split(x_data, y_data,
    ↪test_size=0.4, random_state=0)
lre.fit(x_train1[['horsepower']], y_train1)
lre.score(x_test1[['horsepower']], y_test1)
```

```
[207]: 0.7139364665406973
```

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

Cross-Validation Score

Let's import cross_val_score from the module model_selection.

```
[208]: from sklearn.model_selection import cross_val_score
```

We input the object, the feature ("horsepower"), and the target data (y_data). The parameter 'cv' determines the number of folds. In this case, it is 4.

```
[209]: 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 for the fold:

```
[210]: Rcross
```

```
[210]: array([0.7746232 , 0.51716687, 0.74785353, 0.04839605])
```

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

```
[211]: print("The mean of the folds are", Rcross.mean(), "and the standard deviation_↵↵↵is" , Rcross.std())
```

The mean of the folds are 0.522009915042119 and the standard deviation is 0.29118394447560286

We can use negative squared error as a score by setting the parameter ‘scoring’ metric to ‘neg_mean_squared_error’.

```
[212]: -1 * cross_val_score(lre,x_data[['horsepower']],↵↵↵y_data,cv=4,scoring='neg_mean_squared_error')
```

```
[212]: array([20254142.84026704, 43745493.26505169, 12539630.34014931,↵17561927.7224759 ])
```

Calculate the average R^2 using two folds, then find the average R^2 for the second fold utilizing the “horsepower” feature:

```
[213]: Rc=cross_val_score(lre,x_data[['horsepower']], y_data,cv=2)↵Rc.mean()
```

```
[213]: 0.5166761697127429
```

You can also use the function ‘cross_val_predict’ to predict the output. The function splits up the data into the specified number of folds, with one fold for testing and the other folds are used for training. First, import the function:

```
[214]: from sklearn.model_selection import cross_val_predict
```

We input the object, the feature “horsepower”, and the target data y_data. The parameter ‘cv’ determines the number of folds. In this case, it is 4. We can produce an output:

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

```
[215]: array([14141.63807508, 14141.63807508, 20814.29423473, 12745.03562306,↵14762.35027598])
```

Part 2: Overfitting, Underfitting 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. It turns out these differences are more apparent in Multiple Linear Regression and Polynomial Regression so we will explore overfitting in that context.

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

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

```
[216]: LinearRegression()
```

Prediction using training data:

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

```
[217]: array([ 7426.6731551 , 28323.75090803, 14213.38819709,  4052.34146983,
      34500.19124244])
```

Prediction using test data:

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

```
[218]: array([11349.35089149,  5884.11059106, 11208.6928275 ,  6641.07786278,
      15565.79920282])
```

Let's perform some model evaluation using our training and testing data separately. First, we import the seaborn and matplotlib library for plotting.

```
[219]: import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

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

```
[220]: Title = 'Distribution Plot of Predicted Value Using Training Data vs Training
      ↪Data Distribution'
DistributionPlot(y_train, yhat_train, "Actual Values (Train)", "Predicted
      ↪Values (Train)", Title)
```

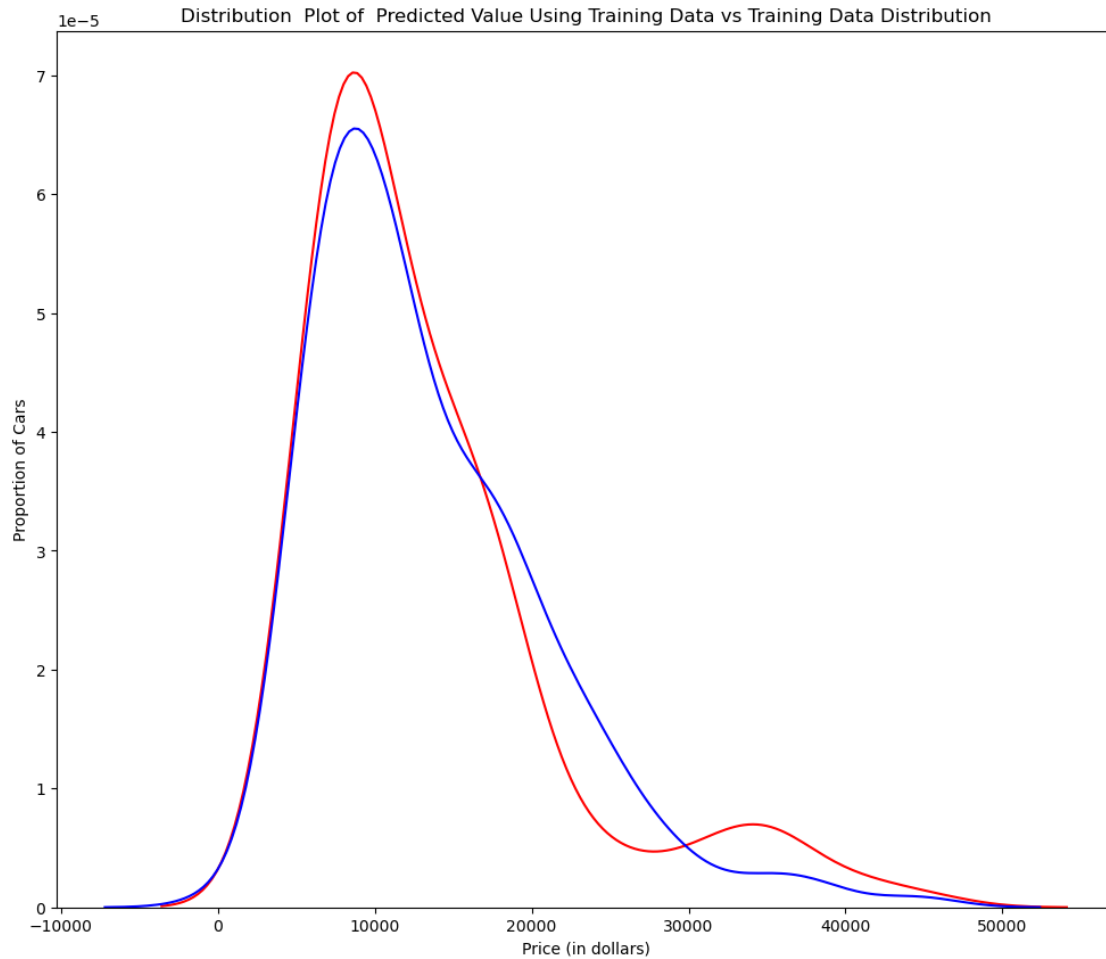
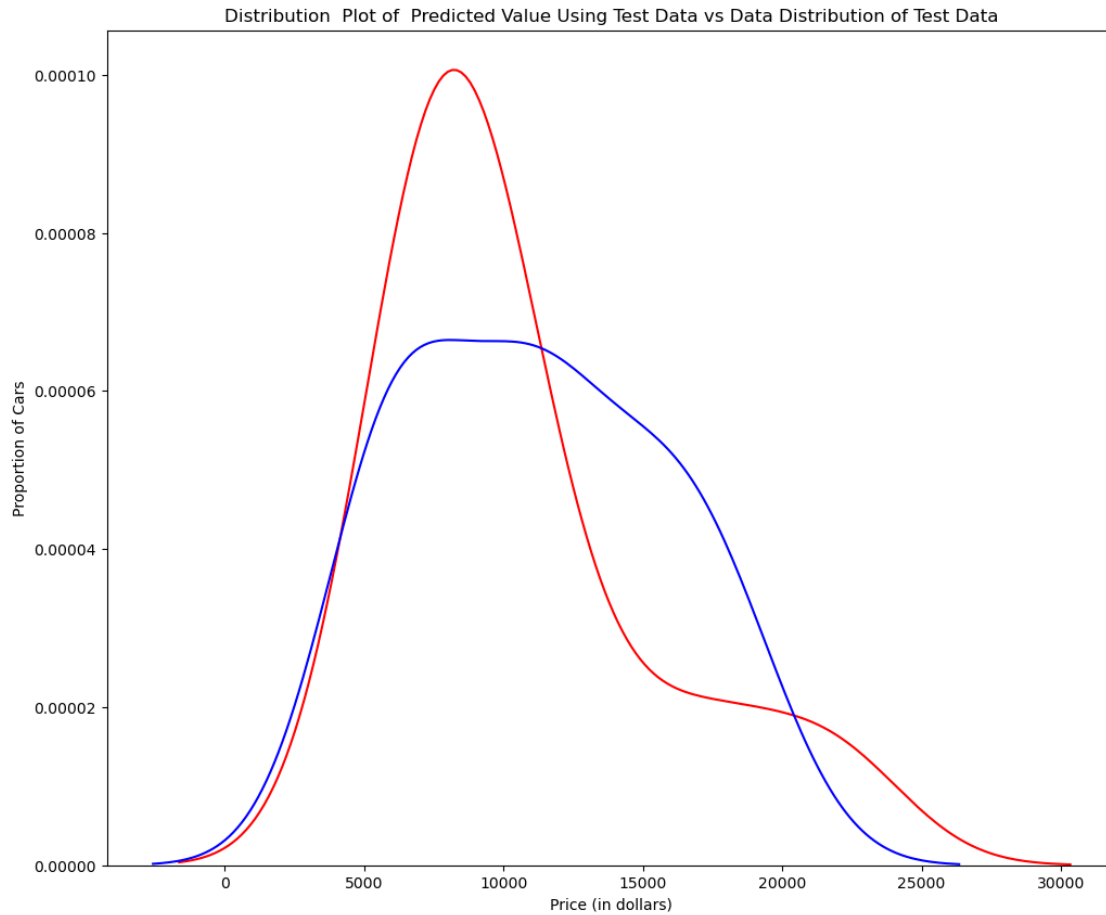


Figure 1: Plot of predicted values using the training data compared to the actual values of the training data.

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.

```
[221]: Title='Distribution Plot of Predicted Value Using Test Data vs Data_
        ↳Distribution of Test Data'
        DistributionPlot(y_test,yhat_test,"Actual Values (Test)","Predicted Values_
        ↳(Test)",Title)
```



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

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

```
[222]: from sklearn.preprocessing import PolynomialFeatures
```

Overfitting

Overfitting occurs when the model fits the noise, but not the underlying process. Therefore, when testing your model using the test set, your model does not perform as well since it is modelling noise, not the underlying process that generated the relationship. Let's create a degree 5 polynomial model.

Let's use 55 percent of the data for training and the rest for testing:

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

We will perform a degree 5 polynomial transformation on the feature ‘horsepower’.

```
[224]: pr = PolynomialFeatures(degree=5)
x_train_pr = pr.fit_transform(x_train[['horsepower']])
x_test_pr = pr.fit_transform(x_test[['horsepower']])
pr
```

```
[224]: PolynomialFeatures(degree=5)
```

Now, let’s create a Linear Regression model “poly” and train it.

```
[225]: poly = LinearRegression()
poly.fit(x_train_pr, y_train)
```

```
[225]: LinearRegression()
```

We can see the output of our model using the method “predict.” We assign the values to “yhat”.

```
[226]: yhat = poly.predict(x_test_pr)
yhat[0:5]
```

```
[226]: array([ 6728.58615619,  7307.91973653, 12213.73734432, 18893.37966315,
        19996.10669225])
```

Let’s take the first five predicted values and compare it to the actual targets.

```
[227]: print("Predicted values:", yhat[0:4])
print("True values:", y_test[0:4].values)
```

```
Predicted values: [ 6728.58615619  7307.91973653 12213.73734432 18893.37966315]
True values: [ 6295. 10698. 13860. 13499.]
```

We will use the function “PollyPlot” that we defined at the beginning of the lab to display the training data, testing data, and the predicted function.

```
[228]: PollyPlot(x_train['horsepower'], x_test['horsepower'], y_train, y_test, poly,pr)
```

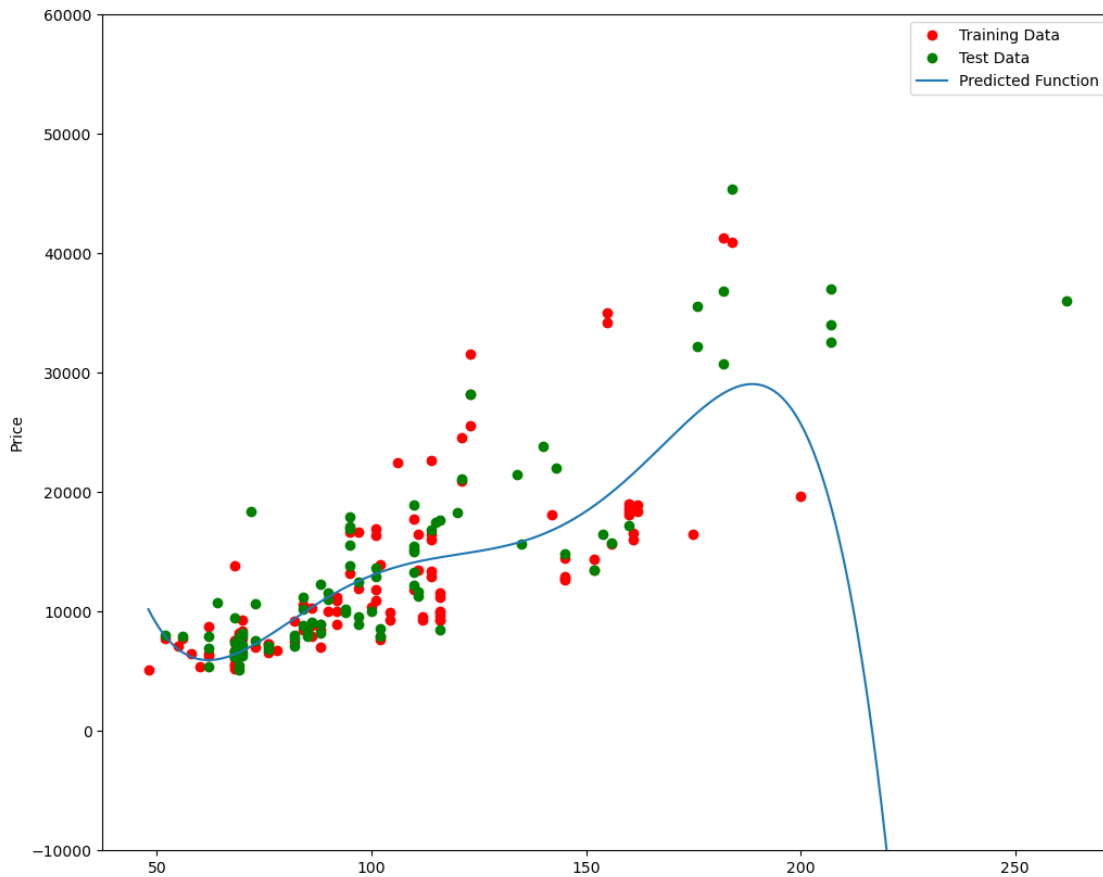


Figure 3: A polynomial regression model where red dots represent training data, green dots represent test data, and the blue line represents the model prediction.

We see that the estimated function appears to track the data but around 200 horsepower, the function begins to diverge from the data points.

R^2 of the training data:

```
[229]: poly.score(x_train_pr, y_train)
```

```
[229]: 0.5567716897727109
```

R^2 of the test data:

```
[230]: poly.score(x_test_pr, y_test)
```

```
[230]: -29.870994900857237
```

We see the R^2 for the training data is 0.750 while the R^2 on the test data was -405.29. 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 the test data for different order polynomials and then plot the results:

```

[231]: 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']])

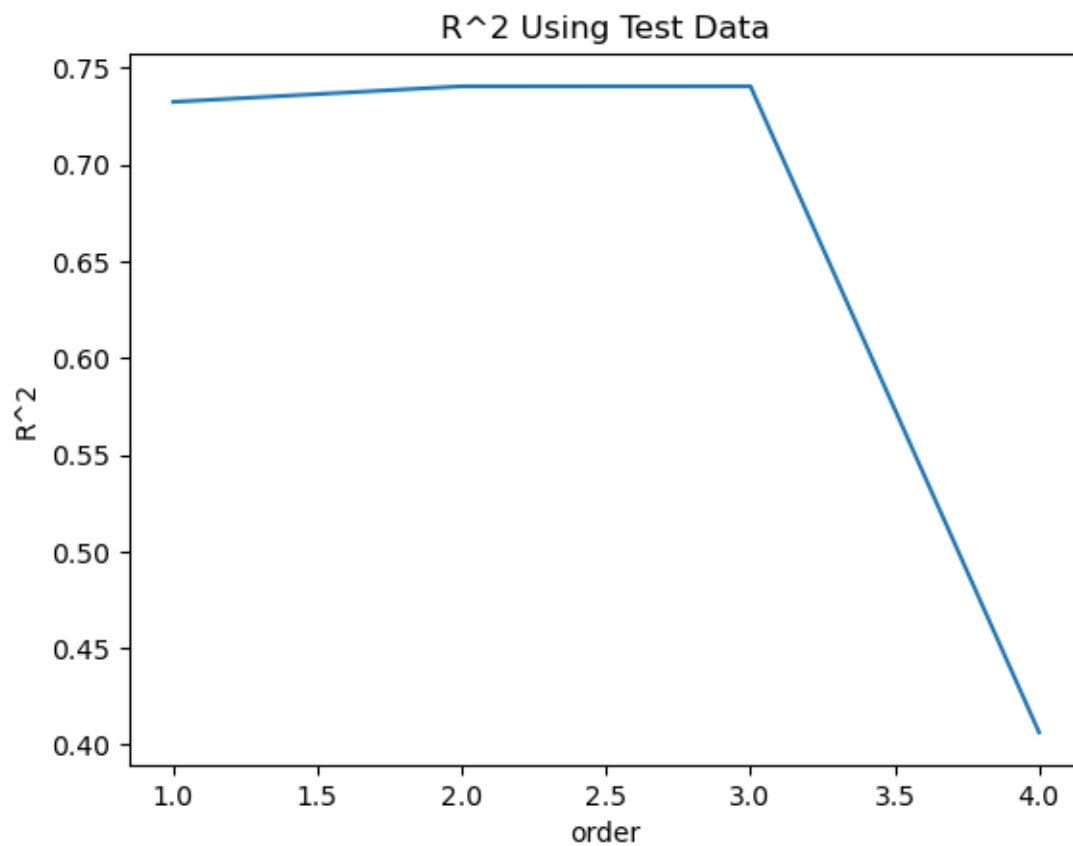
    lr.fit(x_train_pr, y_train)

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

plt.plot(order, Rsqu_test)
plt.xlabel('order')
plt.ylabel('R^2')
plt.title('R^2 Using Test Data')

```

[231]: Text(0.5, 1.0, 'R^2 Using Test Data')



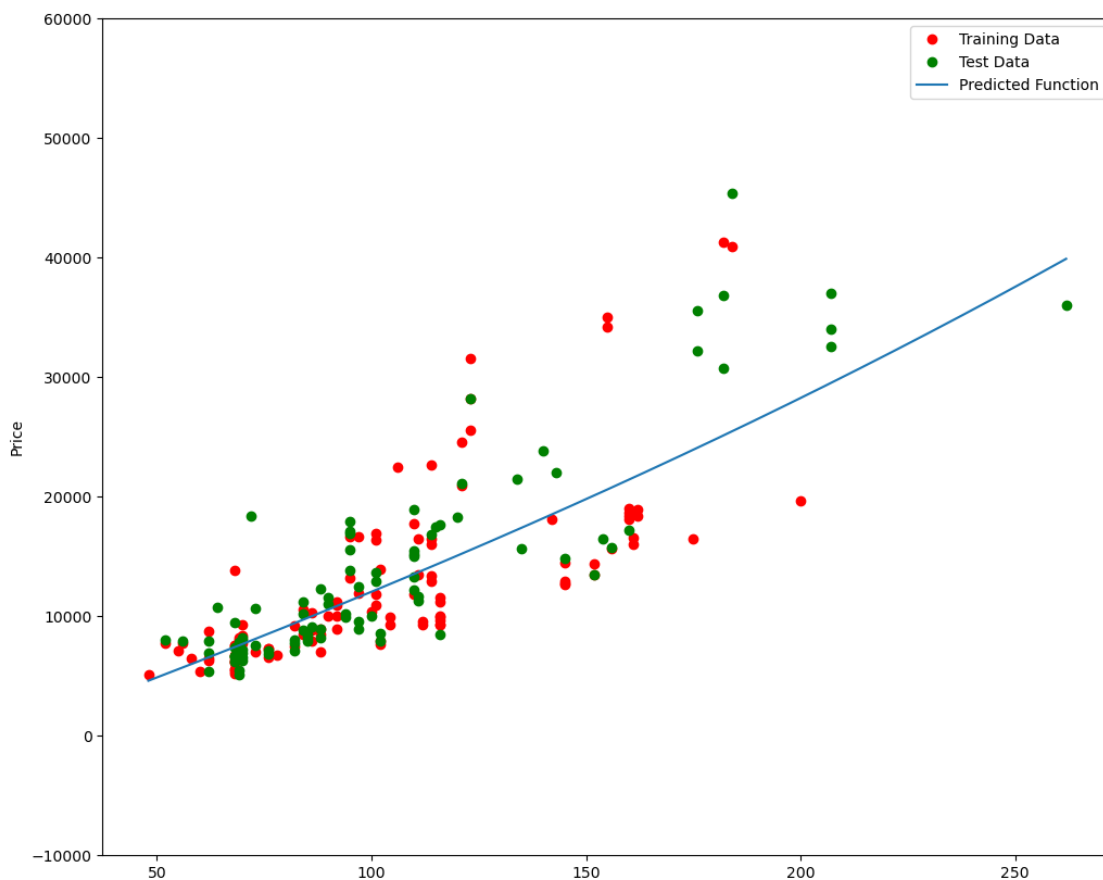
We see the R^2 gradually increases until an order three polynomial is used. Then, the R^2 dramatically decreases at an order four polynomial.

The following function will be used in the next section. Please run the cell below.

```
[232]: def f(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 = 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)  
    PollyPlot(x_train['horsepower'], x_test['horsepower'], y_train, y_test,   
    ↪poly, pr)
```

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

```
[233]: interact(f, order=(0, 6, 1), test_data=(0.05, 0.95, 0.05))
```



```
interactive(children=(IntSlider(value=3, description='order', max=6),  
    ↪FloatSlider(value=0.45, description='tes...
```

[233]: <function __main__.f(order, test_data)>

We can perform polynomial transformations with more than one feature. Create a “PolynomialFeatures” object “pr1” of degree two.

[234]: `pr1=PolynomialFeatures(degree=2)`

Transform the training and testing samples for the features ‘horsepower’, ‘curb-weight’, ‘engine-size’ and ‘highway-mpg’. Hint: use the method “fit_transform”.

[235]: `x_train_pr1=pr1.fit_transform(x_train[['horsepower', 'curb-weight',
 ↪'engine-size', 'highway-mpg']])

x_test_pr1=pr1.fit_transform(x_test[['horsepower', 'curb-weight',
 ↪'engine-size', 'highway-mpg']])`

How many dimensions does the new feature have?

[236]: `x_train_pr1.shape` *#there are now 15 features*

[236]: (110, 15)

Create a linear regression model “poly1”. Train the object using the method “fit” using the polynomial features.

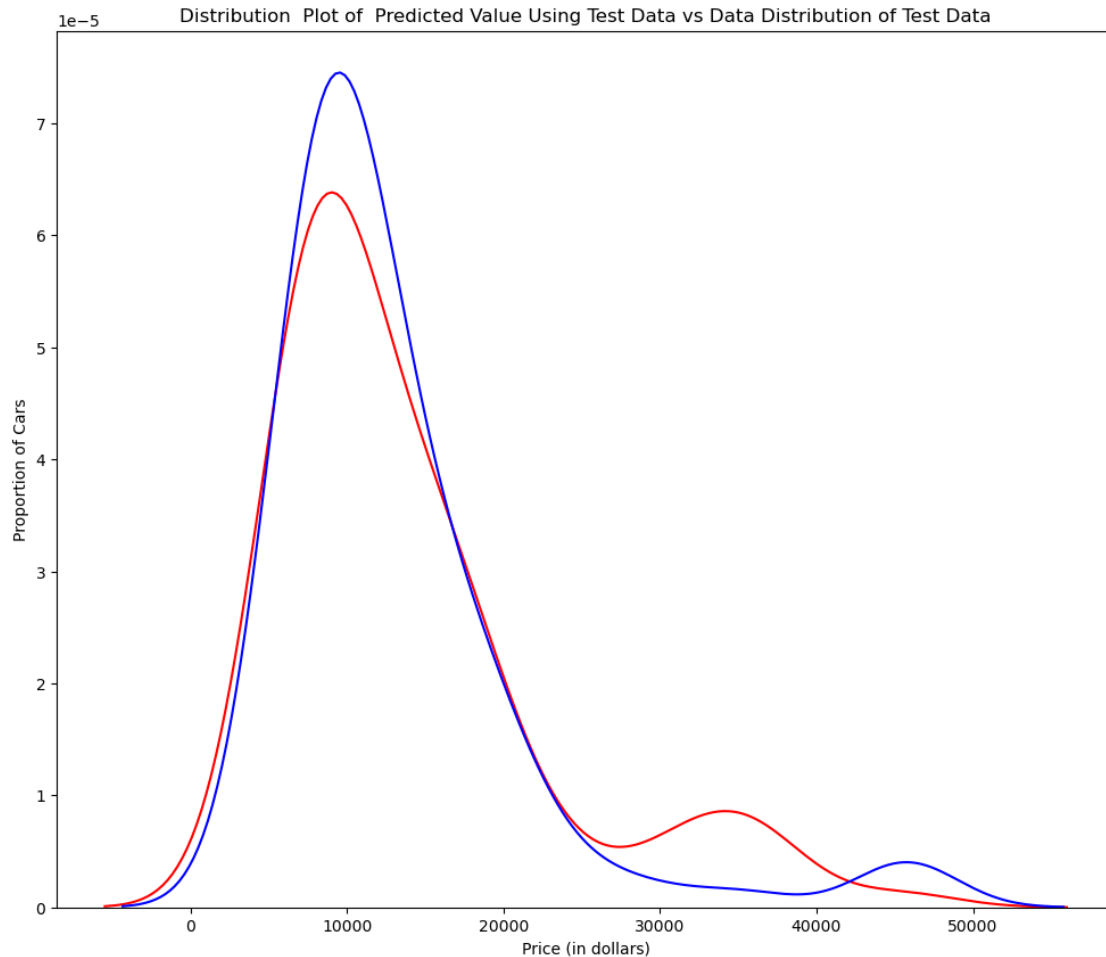
[237]: `poly1=LinearRegression().fit(x_train_pr1,y_train)`

Use the method “predict” to predict an output on the polynomial features, then use the function “DistributionPlot” to display the distribution of the predicted test output vs. the actual test data.

[238]: `yhat_test1=poly1.predict(x_test_pr1)

Title='Distribution Plot of Predicted Value Using Test Data vs Data_
 ↪Distribution of Test Data'

DistributionPlot(y_test, yhat_test1, "Actual Values (Test)", "Predicted Values_
 ↪(Test)", Title)`



Using the distribution plot above, describe (in words) the two regions where the predicted prices are less accurate than the actual prices.

[239]: *#The predicted value is higher than actual value for cars where the price is in the \$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.*

Part 3: Ridge Regression

In this section, we will review Ridge Regression and see how the parameter alpha changes the model. Just a note, here our test data will be used as validation data.

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

[240]: `pr=PolynomialFeatures(degree=2)
x_train_pr=pr.fit_transform(x_train[['horsepower', 'curb-weight',
'engine-size', 'highway-mpg','normalized-losses','symboling']])`

```
x_test_pr=pr.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size', '↪highway-mpg', 'normalized-losses', 'symboling']])
```

Let's import Ridge from the module linear models.

```
[241]: from sklearn.linear_model import Ridge
```

Let's create a Ridge regression object, setting the regularization parameter (alpha) to 0.1

```
[242]: RigeModel=Ridge(alpha=1)
```

Like regular regression, you can fit the model using the method fit.

```
[243]: RigeModel.fit(x_train_pr, y_train)
```

```
[243]: Ridge(alpha=1)
```

Similarly, you can obtain a prediction:

```
[244]: yhat = RigeModel.predict(x_test_pr)
```

Let's compare the first five predicted samples to our test set:

```
[245]: print('predicted:', yhat[0:4])
print('test set :', y_test[0:4].values)
```

```
predicted: [ 6570.82441941  9636.24891471 20949.92322737 19403.60313255]
test set : [ 6295. 10698. 13860. 13499.]
```

We select the value of alpha that minimizes the test error. To do so, we can use a for loop. We have also created a progress bar to see how many iterations we have completed so far.

```
[246]: from tqdm import tqdm

Rsqu_test = []
Rsqu_train = []
dummy1 = []
Alpha = 10 * np.array(range(0,1000))
pbar = tqdm(Alpha)

for alpha in pbar:
    RigeModel = Ridge(alpha=alpha)
    RigeModel.fit(x_train_pr, y_train)
    test_score, train_score = RigeModel.score(x_test_pr, y_test), RigeModel.
    ↪score(x_train_pr, y_train)
    pbar.set_postfix({"Test Score": test_score, "Train Score": train_score})

    Rsqu_test.append(test_score)
    Rsqu_train.append(train_score)
```

100% | 1000/1000 [00:09<00:00,
100.30it/s, Test Score=0.564, Train Score=0.859]

We can plot out the value of R^2 for different alphas:

```
[247]: width = 12  
height = 10  
plt.figure(figsize=(width, height))  
  
plt.plot(Alpha, Rsqu_test, label='validation data ')  
plt.plot(Alpha, Rsqu_train, 'r', label='training Data ')  
plt.xlabel('alpha')  
plt.ylabel('R^2')  
plt.legend()
```

[247]: <matplotlib.legend.Legend at 0x1acd5f49e90>

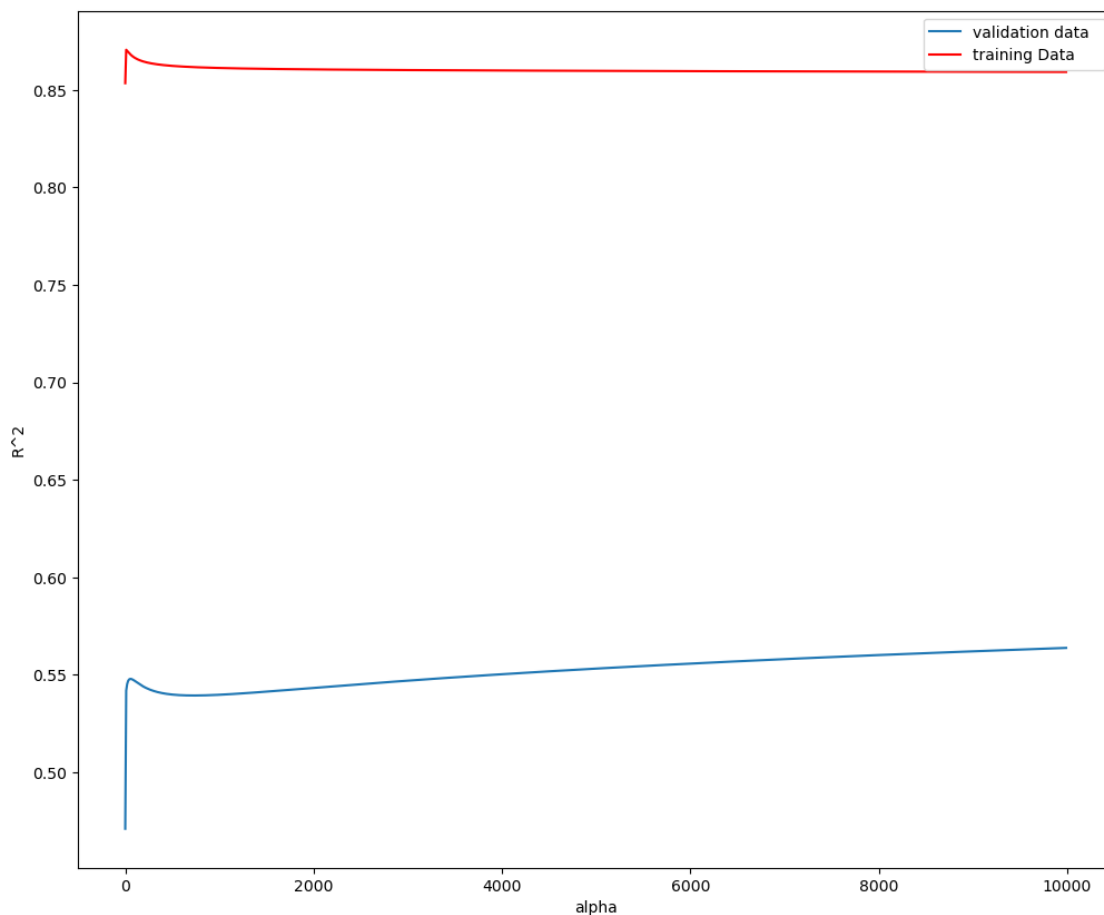


Figure 4: The blue line represents the R^2 of the validation data, and the red line represents the R^2 of the training data. The x-axis represents the different values of Alpha.

Here the model is built and tested on the same data, so the training and test data are the same.

The red line in Figure 4 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 validation data. As the value for alpha increases, the R^2 increases and converges at a point.

Perform Ridge regression. Calculate the R^2 using the polynomial features, use the training data to train the model and use the test data to test the model. The parameter alpha should be set to 10.

```
[248]: RidgeModel = Ridge(alpha=10)
RidgeModel.fit(x_train_pr, y_train)
RidgeModel.score(x_test_pr, y_test)
```

```
[248]: 0.5418576440206405
```

Part 4: 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.

```
[249]: from sklearn.model_selection import GridSearchCV
```

We create a dictionary of parameter values:

```
[250]: parameters1= [{'alpha': [0.001,0.1,1, 10, 100, 1000, 10000, 100000, 100000]}]
parameters1
```

```
[250]: [{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 100000]}]
```

Create a Ridge regression object:

```
[251]: RR=Ridge()
RR
```

```
[251]: Ridge()
```

Create a ridge grid search object:

```
[252]: Grid1 = GridSearchCV(RR, parameters1,cv=4)
```

Fit the model:

```
[253]: Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],
↳y_data)
```

```
[253]: GridSearchCV(cv=4, estimator=Ridge(),
param_grid=[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000,
100000]}])
```

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:

```
[254]: BestRR=Grid1.best_estimator_  
BestRR
```

```
[254]: Ridge(alpha=10000)
```

We now test our model on the test data:

```
[255]: BestRR.score(x_test[['horsepower', 'curb-weight', 'engine-size',  
    ↪ 'highway-mpg']], y_test)
```

```
[255]: 0.841164983103615
```

Perform a grid search for the alpha parameter and the normalization parameter, then find the best values of the parameters:

```
[256]: parameters2 = [{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 100000]}]  
  
Grid2 = GridSearchCV(Ridge(), parameters2, cv=4)  
Grid2.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],  
    ↪ y_data)  
best_alpha = Grid2.best_params_['alpha']  
best_ridge_model = Ridge(alpha=best_alpha)  
best_ridge_model.fit(x_data[['horsepower', 'curb-weight', 'engine-size',  
    ↪ 'highway-mpg']], y_data)
```

```
[256]: Ridge(alpha=10000)
```

```
###
```

© Dhesika