Sheet

import pandas as pd import numpy as np import seaborn as sns %matplotlib inline #import matplotlib.pyplot as plt

```
# Read the online file by the URL provides above, and assign it to variable "df"
df = pd.read_csv("auto.csv", header=None)
#dset = sns.load_dataset("df")
df.head(100)
df.tail(10)
```

	0	1	2	3	4	5	6	7	8	9	...	16	17	18	19	20
195	-1	74	volvo	gas	std	four	wagon	rwd	front	104.3	...	141	mpfi	3.78	3.15	9.5
196	-2	103	volvo	gas	std	four	sedan	rwd	front	104.3	...	141	mpfi	3.78	3.15	9.5
197	-1	74	volvo	gas	std	four	wagon	rwd	front	104.3	...	141	mpfi	3.78	3.15	9.5
198	-2	103	volvo	gas	turbo	four	sedan	rwd	front	104.3	...	130	mpfi	3.62	3.15	7.5
199	-1	74	volvo	gas	turbo	four	wagon	rwd	front	104.3	...	130	mpfi	3.62	3.15	7.5
200	-1	95	volvo	gas	std	four	sedan	rwd	front	109.1	...	141	mpfi	3.78	3.15	9.5
201	-1	95	volvo	gas	turbo	four	sedan	rwd	front	109.1	...	141	mpfi	3.78	3.15	8.7
202	-1	95	volvo	gas	std	four	sedan	rwd	front	109.1	...	173	mpfi	3.58	2.87	8.8
203	-1	95	volvo	diesel	turbo	four	sedan	rwd	front	109.1	...	145	idi	3.01	3.40	23.0
204	-1	95	volvo	gas	turbo	four	sedan	rwd	front	109.1	...	141	mpfi	3.78	3.15	9.5

10 rows × 26 columns

```
#create headr list for columns
headers = ["symboling","normalized-losses","make","fuel-type","aspiration", "num-of-d
         "drive-wheels","engine-location","wheel-base", "length","width","height","cu
         "num-of-cylinders", "engine-size","fuel-system","bore","stroke","compressior
         "peak-rpm","city-mpg","highway-mpg","price"]

print("headers\n", headers)

# we replace header and recheck dataframe
df.columns = headers
df.head()
```

```
#we need to repalce  the "?" symbol with NaN so the dropna() can remove the missing v
df1 = df.replace('NaN', '?', inplace = True)
df.head(200)
```

```
headers
 ['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration', 'num-of-door
```

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	f s
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	r
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	r
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	r
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	r
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	r
...	...	...	...	...	...	...	...	...	...	...	...	...	..
195	-1	74	volvo	gas	std	four	wagon	rwd	front	104.3	...	141	r
196	-2	103	volvo	gas	std	four	sedan	rwd	front	104.3	...	141	r
197	-1	74	volvo	gas	std	four	wagon	rwd	front	104.3	...	141	r
198	-2	103	volvo	gas	turbo	four	sedan	rwd	front	104.3	...	130	r
199	-1	74	volvo	gas	turbo	four	wagon	rwd	front	104.3	...	130	r

200 rows × 26 columns

```
print(df.columns)
#print(df.dtypes)
```

```
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')
```

```
## Describe ###################
# If we would like to get a statistical summary of each column e.g. count, column mea

# this methods provides various summary statiscs excluding NaN values
df.describe()
#df[['length', 'compression-ratio']].describe()
df.describe(include = "all")
```

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine
count	205.000000	205	205	205	205	205	205	205	205	205.000000	...	205.00
unique	NaN	52	22	2	2	3	5	3	2	NaN	...	NaN
top	NaN	?	toyota	gas	std	four	sedan	fwd	front	NaN	...	NaN
freq	NaN	41	32	185	168	114	96	120	202	NaN	...	NaN
mean	0.834146	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	98.756585	...	126.90
std	1.245307	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	6.021776	...	41.642
min	-2.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	86.600000	...	61.000
25%	0.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	94.500000	...	97.000
50%	1.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	97.000000	...	120.00
75%	2.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	102.400000	...	141.00
max	3.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	120.900000	...	326.00

11 rows × 26 columns

```
#You can select the columns of a dataframe by indicating the name of each column. For
df[['length','engine-size']].describe()
```

	length	engine-size
count	205.000000	205.000000
mean	174.049268	126.907317
std	12.337289	41.642693
min	141.100000	61.000000
25%	166.300000	97.000000
50%	173.200000	120.000000
75%	183.100000	141.000000
max	208.100000	326.000000

```
df.to_csv('automobile.csv', index=False)
```

```
# Basic Insight of data types #######
########################################
#There are several ways to obtain essential insights of the data to help us better ur
# Data has variety of type
# The main types stored in Pandas dataframes are object, float, int, bool and dateti

#df.dtypes
# check the data type of data frame "df" by .dtypes
print(df.dtypes)

#  As shown above, it is clear to see that the data type of "symboling" and "curb-wei
#These data types can be changed;
```

```
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
```

```
## Info ##########
# Another method you can use to check your dataset is. df.info()
# It provides a concise summary of your DataFrame.
#This method prints information about a DataFrame including the index dtype and colu
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 26 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   symboling          205 non-null    int64  
 1   normalized-losses  205 non-null    object 
 2   make               205 non-null    object 
 3   fuel-type          205 non-null    object 
 4   aspiration         205 non-null    object 
 5   num-of-doors       205 non-null    object 
 6   body-style         205 non-null    object 
 7   drive-wheels       205 non-null    object 
 8   engine-location    205 non-null    object 
 9   wheel-base         205 non-null    float64
 10  length             205 non-null    float64
 11  width              205 non-null    float64
 12  height             205 non-null    float64
 13  curb-weight        205 non-null    int64  
 14  engine-type        205 non-null    object 
 15  num-of-cylinders   205 non-null    object 
 16  engine-size        205 non-null    int64  
 17  fuel-system        205 non-null    object 
 18  bore               205 non-null    object 
 19  stroke             205 non-null    object 
 20  compression-ratio  205 non-null    float64
 21  horsepower         205 non-null    object 
 22  peak-rpm           205 non-null    object 
 23  city-mpg           205 non-null    int64  
 24  highway-mpg        205 non-null    int64  
 25  price              205 non-null    object 
dtypes: float64(5), int64(5), object(16)
memory usage: 41.8+ KB
```

```
## Data Wraglling ####################
## Handle missing values, correct data formate, standardize and normalize data

#What is the purpose of data wrangling?
    #Data wrangling is the process of converting data from the initial format to a fo
```

```
# Identify and handle missing values
# In the car dataset, missing data comes with the question mark "?". We replace "?" w
# Here we use the function:  df.replace(A, B, inplace =true)
#we need to repalce  the "?" symbol with NaN so the dropna() can remove the missing v
df1 = df.replace('?', np.NaN, inplace = True)
```

```
df.head(200)
```

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	f s
0	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	r
1	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	r
2	1	NaN	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	r
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	r
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	r
...	...	...	...	...	...	...	...	...	...	...	...	...	..
195	-1	74	volvo	gas	std	four	wagon	rwd	front	104.3	...	141	r
196	-2	103	volvo	gas	std	four	sedan	rwd	front	104.3	...	141	r
197	-1	74	volvo	gas	std	four	wagon	rwd	front	104.3	...	141	r
198	-2	103	volvo	gas	turbo	four	sedan	rwd	front	104.3	...	130	r
199	-1	74	volvo	gas	turbo	four	wagon	rwd	front	104.3	...	130	r

200 rows × 26 columns

```
#Evaluating for missing value by using two methods isnull and not null
#True is missng values while false means isn't missing value

missing_data = df.notnull()
missing_data.head(5)
```

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	b
0	True	False	True	True	True	True	True	True	True	True	...	True	True	Tr
1	True	False	True	True	True	True	True	True	True	True	...	True	True	Tr
2	True	False	True	True	True	True	True	True	True	True	...	True	True	Tr
3	True	True	True	True	True	True	True	True	True	True	...	True	True	Tr
4	True	True	True	True	True	True	True	True	True	True	...	True	True	Tr

5 rows × 26 columns

```
# count missing value in each column

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

```
symboling
True    205
Name: symboling, dtype: int64

normalized-losses
True     164
False     41
Name: normalized-losses, dtype: int64

make
True    205
Name: make, dtype: int64

fuel-type
True    205
Name: fuel-type, dtype: int64

aspiration
True    205
Name: aspiration, dtype: int64

num-of-doors
True     203
False      2
Name: num-of-doors, dtype: int64

body-style
True    205
Name: body-style, dtype: int64

drive-wheels
True    205
Name: drive-wheels, dtype: int64

engine-location
True    205
Name: engine-location, dtype: int64
```

```
wheel-base
True    205
Name: wheel-base, dtype: int64

length
True    205
Name: length, dtype: int64

width
True    205
Name: width, dtype: int64

height
True    205
Name: height, dtype: int64

curb-weight
True    205
Name: curb-weight, dtype: int64

engine-type
True    205
Name: engine-type, dtype: int64

num-of-cylinders
True    205
Name: num-of-cylinders, dtype: int64

engine-size
True    205
Name: engine-size, dtype: int64

fuel-system
True    205
Name: fuel-system, dtype: int64

bore
True     201
False      4
Name: bore, dtype: int64

stroke
True     201
False      4
Name: stroke, dtype: int64

compression-ratio
True    205
Name: compression-ratio, dtype: int64

horsepower
True     203
False      2
Name: horsepower, dtype: int64
```

```
peak-rpm
True     203
False      2
Name: peak-rpm, dtype: int64

city-mpg
True    205
Name: city-mpg, dtype: int64

highway-mpg
True    205
Name: highway-mpg, dtype: int64

price
True     201
False      4
Name: price, dtype: int64
```

```
###############   Data Wrangling #########################

# Deal with missing data
# How to deal with missing data
    # Drop Data -> drop whole column and Raw
    # Replace Data -> Replce it by mean , frequency, and other funcation
# Calculate the mean value for the normalization-lossess column

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

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

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

# Calculate the mean value for the "bore" column
avg_bore = df['bore'].astype("float").mean(axis=0)
print("Avergae of bore:", avg_bore)

# Replace "NaN" with mean value in bore
df["bore"].replace(np.nan, avg_bore, inplace=True)


## Calculating mean value for stroke column ##
avg_stroke = df['stroke'].astype("float").mean(axis=0)
print("Average of stroke:", avg_stroke)

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

## Calcualting mean value for price column ##
avg_price = df['price'].astype("float").mean(axis=0)
```

```
print("Average of price:", avg_price)

## replace "Nan" with mean value in price
df["price"].replace(np.nan, avg_price, inplace=True)

df.head(50)
```

```
Average of normalized-losses: 122.0
Avergae of bore: 3.3297512437810943
Average of stroke: 3.255422885572139
Average of price: 13207.129353233831
```

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	f s
0	3	122.0	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	r
1	3	122.0	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	r
2	1	122.0	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	r
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	r
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	r
5	2	122.0	audi	gas	std	two	sedan	fwd	front	99.8	...	136	r
6	1	158	audi	gas	std	four	sedan	fwd	front	105.8	...	136	r
7	1	122.0	audi	gas	std	four	wagon	fwd	front	105.8	...	136	r
8	1	158	audi	gas	turbo	four	sedan	fwd	front	105.8	...	131	r
9	0	122.0	audi	gas	turbo	two	hatchback	4wd	front	99.5	...	131	r
10	2	192	bmw	gas	std	two	sedan	rwd	front	101.2	...	108	r
11	0	192	bmw	gas	std	four	sedan	rwd	front	101.2	...	108	r
12	0	188	bmw	gas	std	two	sedan	rwd	front	101.2	...	164	r
13	0	188	bmw	gas	std	four	sedan	rwd	front	101.2	...	164	r
14	1	122.0	bmw	gas	std	four	sedan	rwd	front	103.5	...	164	r
15	0	122.0	bmw	gas	std	four	sedan	rwd	front	103.5	...	209	r
16	0	122.0	bmw	gas	std	two	sedan	rwd	front	103.5	...	209	r
17	0	122.0	bmw	gas	std	four	sedan	rwd	front	110.0	...	209	r
18	2	121	chevrolet	gas	std	two	hatchback	fwd	front	88.4	...	61	2
19	1	98	chevrolet	gas	std	two	hatchback	fwd	front	94.5	...	90	2
20	0	81	chevrolet	gas	std	four	sedan	fwd	front	94.5	...	90	2
21	1	118	dodge	gas	std	two	hatchback	fwd	front	93.7	...	90	2
22	1	118	dodge	gas	std	two	hatchback	fwd	front	93.7	...	90	2

23	1	118	dodge	gas	turbo	two	hatchback	fwd	front	93.7	...	98	
24	1	148	dodge	gas	std	four	hatchback	fwd	front	93.7	...	90	
25	1	148	dodge	gas	std	four	sedan	fwd	front	93.7	...	90	
26	1	148	dodge	gas	std	four	sedan	fwd	front	93.7	...	90	
27	1	148	dodge	gas	turbo	NaN	sedan	fwd	front	93.7	...	98	
28	-1	110	dodge	gas	std	four	wagon	fwd	front	103.3	...	122	
29	3	145	dodge	gas	turbo	two	hatchback	fwd	front	95.9	...	156	
30	2	137	honda	gas	std	two	hatchback	fwd	front	86.6	...	92	
31	2	137	honda	gas	std	two	hatchback	fwd	front	86.6	...	92	
32	1	101	honda	gas	std	two	hatchback	fwd	front	93.7	...	79	
33	1	101	honda	gas	std	two	hatchback	fwd	front	93.7	...	92	
34	1	101	honda	gas	std	two	hatchback	fwd	front	93.7	...	92	
35	0	110	honda	gas	std	four	sedan	fwd	front	96.5	...	92	
36	0	78	honda	gas	std	four	wagon	fwd	front	96.5	...	92	
37	0	106	honda	gas	std	two	hatchback	fwd	front	96.5	...	110	
38	0	106	honda	gas	std	two	hatchback	fwd	front	96.5	...	110	
39	0	85	honda	gas	std	four	sedan	fwd	front	96.5	...	110	
40	0	85	honda	gas	std	four	sedan	fwd	front	96.5	...	110	
41	0	85	honda	gas	std	four	sedan	fwd	front	96.5	...	110	
42	1	107	honda	gas	std	two	sedan	fwd	front	96.5	...	110	
43	0	122.0	isuzu	gas	std	four	sedan	rwd	front	94.3	...	111	
44	1	122.0	isuzu	gas	std	two	sedan	fwd	front	94.5	...	90	
45	0	122.0	isuzu	gas	std	four	sedan	fwd	front	94.5	...	90	
46	2	122.0	isuzu	gas	std	two	hatchback	rwd	front	96.0	...	119	
47	0	145	jaguar	gas	std	four	sedan	rwd	front	113.0	...	258	
48	0	122.0	jaguar	gas	std	four	sedan	rwd	front	113.0	...	258	
49	0	122.0	jaguar	gas	std	two	sedan	rwd	front	102.0	...	326	

50 rows × 26 columns

```
# To see which values are present in a particular column, we can use the ".value_cou
#df['num-of-doors'].value_counts()

# We can see that four doors are the most common type. we can also use the "idmax() '
df['num-of-doors'].value_counts().idxmax()
```

```
NameError: name 'df' is not defined
```

```
# To see which values are present in a particular column, we can use the ".value_cou
df['num-of-doors'].value_counts() # Result : four:116, two:89

# We can see that four doors are the most common type. we can also use the "idmax() '
df['num-of-doors'].value_counts().idxmax()  # result : Four

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

```
# finally let's drop all rows that do not have price data
# simply drop whole row with NaN in "price" column
df.dropna(subset=["price"], axis=0, inplace=True)

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

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

5 rows × 26 columns

```
### Correct data format ###
# .dtype() to check the data type and .astype() to change the data type
# conver data type to the correct format

df[["bore", "stroke"]] = df[["bore", "stroke"]].astype("float")
df[["normalized-losses"]] = df[["normalized-losses"]].astype("int")
df[["price"]] = df[["price"]].astype("float")
df[["peak-rpm"]] = df[["peak-rpm"]].astype("float")
df.dtypes
```

```
###   Data Standardization ####
###  Data is usually collected from different agencies in different formats. ###
### Standardization is the process of transforming data into a common format, allowi
# data transformation to transform mpg into L/100km

# # Convert mpg to L/100km by mathematical operation (235 divided by mpg)
df['city-L/100km'] = 235/df["highway-mpg"]
# check your transformed data
## rename column name from "highway-mpg" to "highway-L/100km"
df.rename(columns={'highway-mpg':'highway-L/100km'}, inplace=True)

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

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	fuel-system	bore
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	mpfi	3.47
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	mpfi	3.47
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	mpfi	2.68
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	mpfi	3.19
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	mpfi	3.19

5 rows × 27 columns

```
df.dropna(subset=["city-L/100km"], axis=0, inplace=False)
```

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	fuel-system
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	mpfi
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	mpfi
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	mpfi
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	mpfi
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	mpfi
...	...	...	...	...	...	...	...	...	...	...	...	...
200	-1	95	volvo	gas	std	four	sedan	rwd	front	109.1	...	mpfi
201	-1	95	volvo	gas	turbo	four	sedan	rwd	front	109.1	...	mpfi
202	-1	95	volvo	gas	std	four	sedan	rwd	front	109.1	...	mpfi
203	-1	95	volvo	diesel	turbo	four	sedan	rwd	front	109.1	...	idi
204	-1	95	volvo	gas	turbo	four	sedan	rwd	front	109.1	...	mpfi

205 rows × 27 columns

```
df.head()
```

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	fuel-system	bore
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	mpfi	3.47
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	mpfi	3.47
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	mpfi	2.68
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	mpfi	3.19
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	mpfi	3.19

5 rows × 27 columns

```
## Data Normalization ###
## Why Normalization ##
### Normalization is the process of transforming values of several variables into a s
# scaling the variable so the variance is 1, or scaling the variable so the variable
```

```
# Example

#To demonstrate normalization, let's say we want to scale the columns "length", "widt

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

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

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

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

	length	width	height
0	0.811148	0.886584	0.816054
1	0.811148	0.886584	0.816054
2	0.822681	0.905947	0.876254
3	0.848630	0.915629	0.908027
4	0.848630	0.918396	0.908027

```
### Bining ############
# Why bining -> Binning is a process of transforming continuous numerical variables i
# Example -> In our dataset, "horsepower" is a real valued variable ranging from 48 t
# What if we only care about the price difference between cars with high horsepower,
#we will use the pandas method 'cut' to segment the 'horsepower' column into 3 bins
```

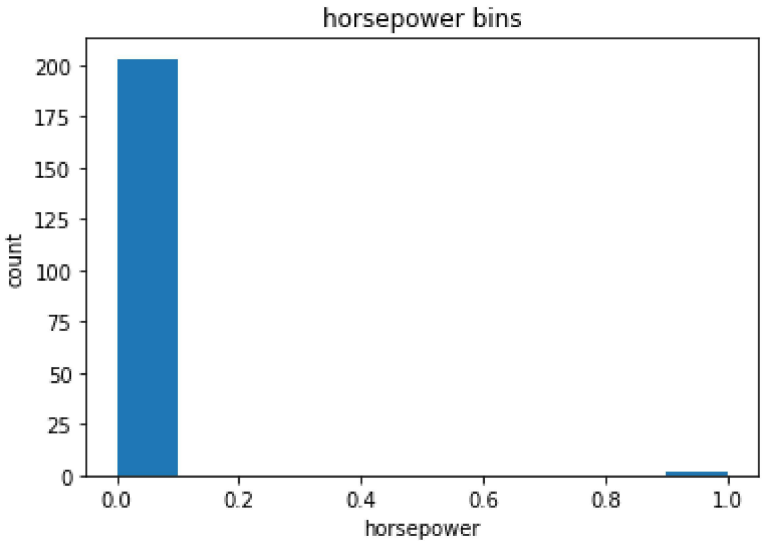
```
df['horsepower'] = df['horsepower'].isnull()
df['horsepower'] = df['horsepower'].astype(int, copy=True)
```

```
# Example
# Convert data to correct format:
#df["horsepower"] = df["horsepower"].astype("int")
df["horsepower"]=df["horsepower"].astype(int, copy=True)

# Let's plot the histogram of horsepower to see what the distribution of horsepower l
%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")
```

```
Text(0.5, 1.0, 'horsepower bins')
```

Download




```
#We would like 3 bins of equal size bandwidth so we use numpy's linspace(start_value,
#Since we want to include the minimum value of horsepower, we want to set start_value
#Since we want to include the maximum value of horsepower, we want to set end_value =
#Since we are building 3 bins of equal length, there should be 4 dividers, so numbers
#We build a bin array with a minimum value to a maximum value by using the bandwidth
        #The values will determine when one bin ends and another begins.

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

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

```
df['horsepower'] = df['horsepower-binned'].isnull()
df["horsepower"]=df['horsepower-binned'].astype(float, copy=True)
```

```
KeyError: 'horsepower-binned'
```

```
# We apply the function "cut" to determine what each value of df['horsepower'] belong
df['horsepower-binned'] = pd.cut(df['horsepower'], bins, labels=group_names, include_
df[['horsepower','horsepower-binned']].head(30)
```

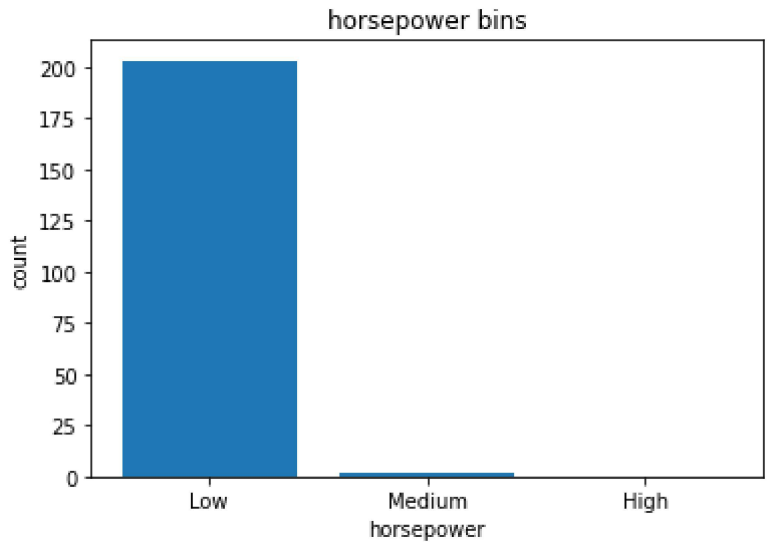
```
ValueError: Bin edges must be unique: array([0., 0., 0., 0.]).
You can drop duplicate edges by setting the 'duplicates' kwarg
```

```
#Let's see the number of vehicles in each bin:
df["horsepower-binned"].value_counts()
```

```
#Let's plot the distribution of each bin:
%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")

# Look at the dataframe above carefully. You will find that the last column provides
# We successfully narrowed down the intervals from 59 to 3!
```

```
Text(0.5, 1.0, 'horsepower bins')
```

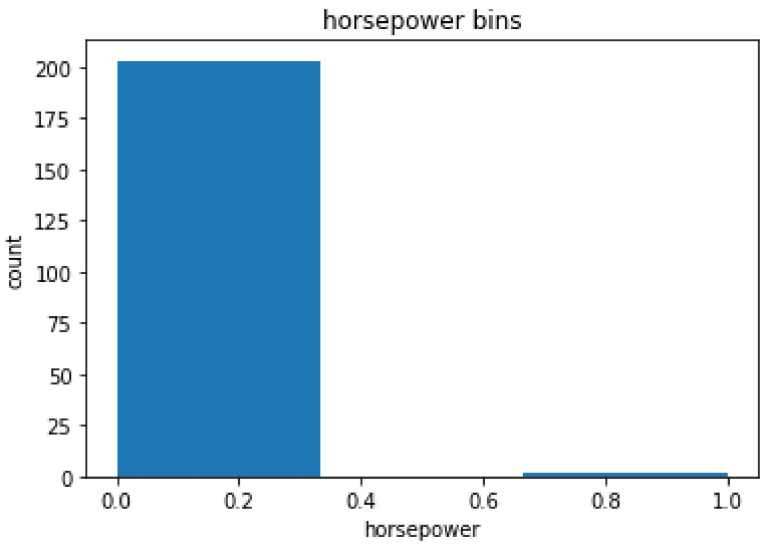
Download



```
# Bins Visualization
# Normally, a histogram is used to visualize the distribution of bins we created abov

%matplotlib inline
import matplotlib as plt
from matplotlib import pyplot


# draw historgram of attribute "horsepower" with bins = 3
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")
```

```
Text(0.5, 1.0, 'horsepower bins')
```

Download




```
## Indicator variable ######
###################################

# what is Indicator varaible?
 # An indicator variable (or dummy variable) is a numerical variable used to label ca
  # They are called 'dummies' because the numbers themselves don't have inherent mean
# Why we use indicator variables?
    #We use indicator variables so we can use categorical variables for regression an
# Example, We see the column "fuel-type" has two unique values: "gas" or "diesel". Re
# We will use pandas' method 'get_dummies' to assign numerical values to different ca


df.columns

# get indicator variables and assign it to data frame "dummy_varaible_1"
```

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

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

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

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

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

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

```
df.head()
```

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

5 rows × 29 columns

```
#df.to_csv('clean_df.csv')
df.to_csv('New_data.csv')
```

```
## Exploratary DataAnalyst ####
### Analyzing Individual Feature Patterns Using Visualization ####
#Import visualization packages "Matplotlib" and "Seaborn". Don't forget about "%matpl

# How to choose the right visualization method?¶
   #Ans # When visualizing individual variables, it is important to first understand

#print(df.dtypes)

# What is the data type of the column "peak-rpm"?
df["peak-rpm"] = df["peak-rpm"].dtypes
print(df["peak-rpm"])
```

```
0      float64
1      float64
2      float64
3      float64
4      float64
        ...
200    float64
201    float64
202    float64
203    float64
204    float64
Name: peak-rpm, Length: 205, dtype: object
```

```
###### Exploratory Data Analysis ###########
# Analyzing individual Feature Patterns Using Visualization #

#df['peak-rpm'].astype

# Example, we can calculate the correlation between variables of type "int64" or "flo
df.corr()
```

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore
symboling	1.000000	0.465190	-0.531954	-0.357612	-0.232919	-0.541038	-0.227691	-0.105790	-0.13008
normalized-losses	0.465190	1.000000	-0.056518	0.019209	0.084195	-0.370706	0.097785	0.110997	-0.02926
wheel-base	-0.531954	-0.056518	1.000000	0.874587	0.795144	0.589435	0.776386	0.569329	0.488760
length	-0.357612	0.019209	0.874587	1.000000	0.841118	0.491029	0.877728	0.683360	0.606462
width	-0.232919	0.084195	0.795144	0.841118	1.000000	0.279210	0.867032	0.735433	0.559152
height	-0.541038	-0.370706	0.589435	0.491029	0.279210	1.000000	0.295572	0.067149	0.171101
curb-weight	-0.227691	0.097785	0.776386	0.877728	0.867032	0.295572	1.000000	0.850594	0.648485
engine-size	-0.105790	0.110997	0.569329	0.683360	0.735433	0.067149	0.850594	1.000000	0.583798
bore	-0.130083	-0.029266	0.488760	0.606462	0.559152	0.171101	0.648485	0.583798	1.000000
stroke	-0.008689	0.054929	0.160944	0.129522	0.182939	-0.055351	0.168783	0.203094	-0.05590
compression-ratio	-0.178515	-0.114525	0.249786	0.158414	0.181129	0.261214	0.151362	0.028971	0.005201
horsepower	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
city-mpg	-0.035823	-0.218749	-0.470414	-0.670909	-0.642704	-0.048640	-0.757414	-0.653658	-0.58450
highway-L/100km	0.034606	-0.178221	-0.544082	-0.704662	-0.677218	-0.107358	-0.797465	-0.677470	-0.58699
price	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
city-L/100km	-0.030190	0.178527	0.578128	0.711597	0.728044	0.085892	0.836742	0.777077	0.551943
fuel-type-diesel	-0.194311	-0.101437	0.308346	0.212679	0.233880	0.284631	0.217275	0.069594	0.054457
fuel-type-gas	0.194311	0.101437	-0.308346	-0.212679	-0.233880	-0.284631	-0.217275	-0.069594	-0.05445

```
df['horsepower'] = df['horsepower'].isnull()
df["horsepower"]=df["horsepower"].astype(float, copy=True)
```

```
# Q2 find the core relastion between the following columns: broke, stroke, compressio
# Hint: if you would like to select those columns, use the following syntax: df[['bor
```

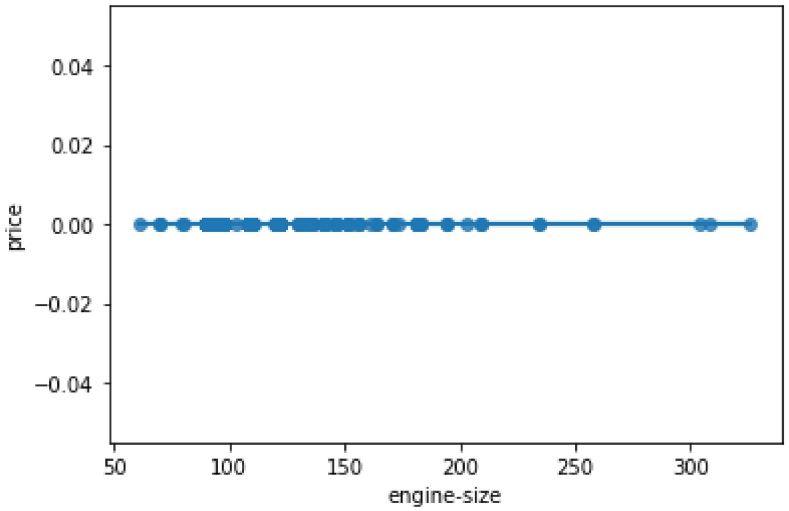
```
df[['bore', 'stroke','compression-ratio', 'horsepower']].corr()
```

	bore	stroke	compression-ratio	horsepower
bore	1.000000	-0.055909	0.005201	NaN
stroke	-0.055909	1.000000	0.186105	NaN
compression-ratio	0.005201	0.186105	1.000000	NaN
horsepower	NaN	NaN	NaN	NaN

```
#Continus numerical variablen
# Postive linear relastionship
# Let's find the scatterplot of "engine-size" and "price".
#umerical_var = sns.load_dataset('df')
sns.regplot(x="engine-size", y="price", data = df)
```

```
<AxesSubplot:xlabel='engine-size', ylabel='price'>
```

Download



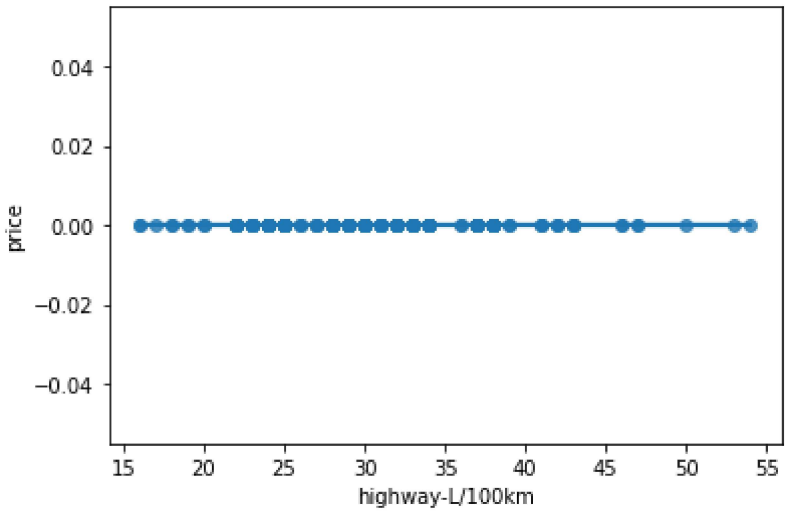

```
# Find core relastion between x = "stroke and y = "price"

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

	engine-size	price
engine-size	1.0	NaN
price	NaN	NaN

```
# Highway mpg is a potential predictor variable of price. Let's find the scatterplot
sns.regplot(x="highway-L/100km", y="price", data=df)
```

```
<AxesSubplot:xlabel='highway-L/100km', ylabel='price'>
```

Download

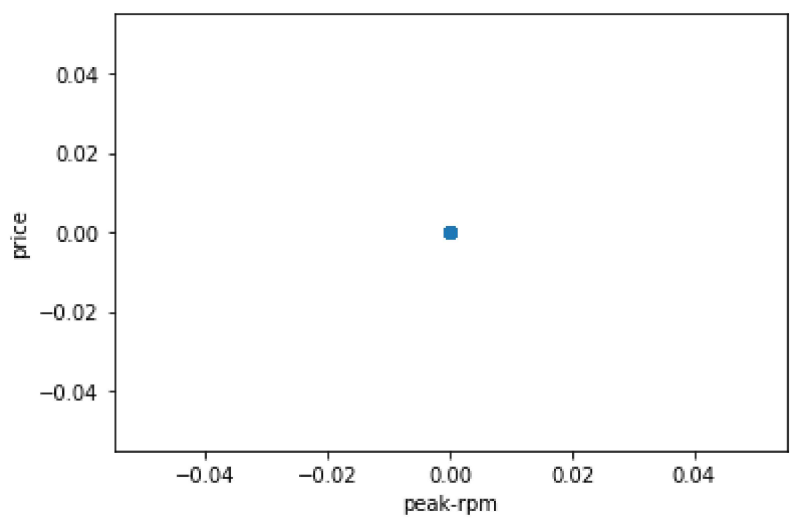



```
df["peak-rpm"] = df["peak-rpm"].isnull()
df["peak-rpm"] = df["peak-rpm"].astype(float, copy=True)
```

```
## Waek linear relastionship #######
## Let's see if "peak-rpm" is a predictor variable of "price".#

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

```
<AxesSubplot:xlabel='peak-rpm', ylabel='price'>
```

Download



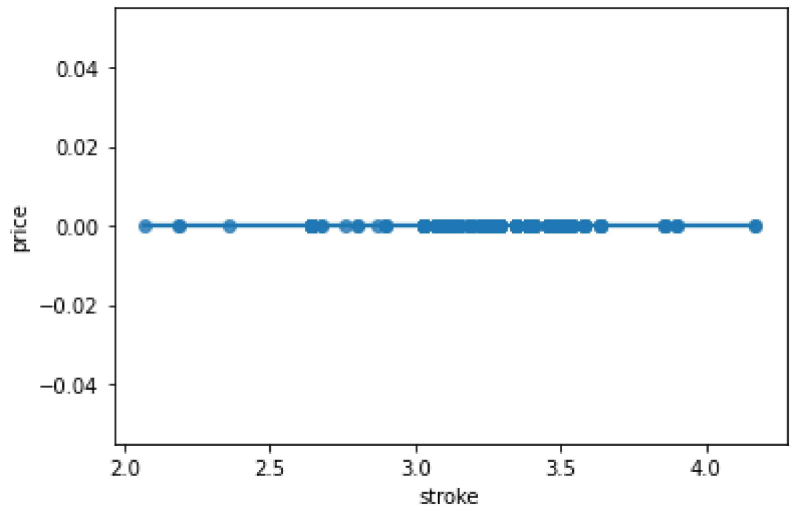

```
df[['peak-rpm','price']].corr()
```

	peak-rpm	price
peak-rpm	NaN	NaN
price	NaN	NaN

```
sns.regplot(x="stroke", y="price", data=df)
```

```
<AxesSubplot:xlabel='stroke', ylabel='price'>
```

Download



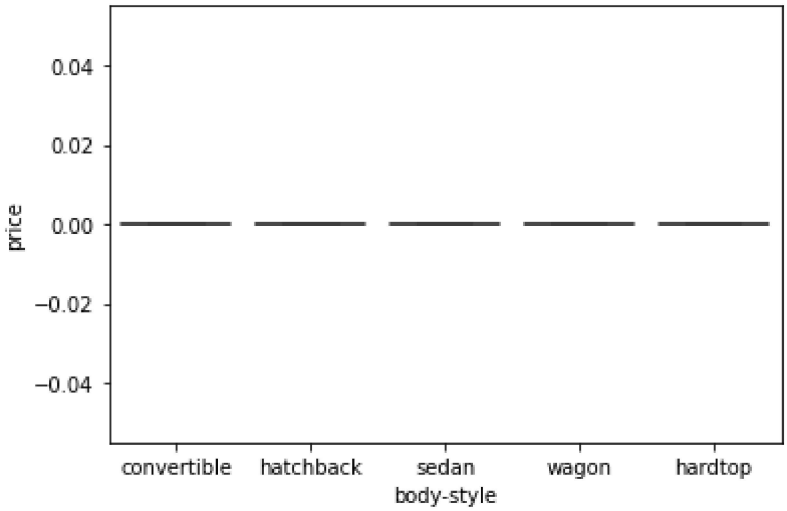

```
# Corr relastionship between sroke and price
df[["stroke", "price"]].corr()
```

	stroke	price
stroke	1.0	NaN
price	NaN	NaN

```
# Categorical variable ####
## These are variables that describe a 'characteristic' of a data unit, and are selec
# A good way to visualize categorical variables is by using boxplots.

# Let's look at the relationship between "body-style" and "price".
sns.boxplot(x="body-style", y="price", data=df)

# We see that the distributions of price between the different body-style categories
# so body-style would not be a good predictor of price. Let's examine engine "engine-
```

```
<AxesSubplot:xlabel='body-style', ylabel='price'>
```

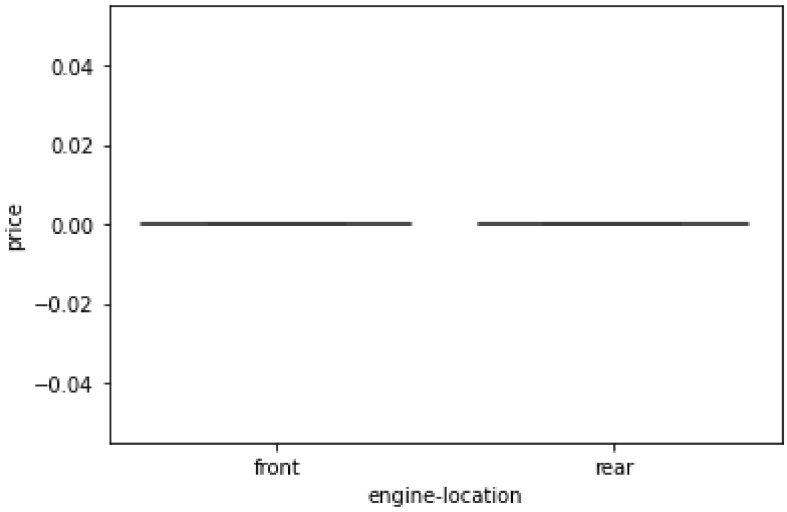
Download




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

#Here we see that the distribution of price between these two engine-location categor
# are distinct enough to take engine-location as a potential good predictor of price.
```

```
<AxesSubplot:xlabel='engine-location', ylabel='price'>
```

Download




```
## Descriptive static analyst ####
##The describe function automatically computes basic statistics for all continuous va
# Any NaN values are automatically skipped in these statistics.

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

	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	engine-type	num-of-cylinders	fuel-system
count	205	205	205	205	205	205	205	205	205
unique	22	2	2	5	3	2	7	7	8
top	toyota	std	four	sedan	fwd	front	ohc	four	mpfi
freq	32	168	116	96	120	202	148	159	94

```
## Value Counts #####
# Value counts is a good way of understanding how many units of each characteristic/v
# 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 data
# As a result, we only include one bracket df['drive-wheels'], not two brackets df[[

#df['drive-wheels'].value_counts()

# We can convert the series to a dataframe as follows
df['drive-wheels'].value_counts().to_frame()
```

	drive-wheels

```
# Let's repeat the above steps but save the results to the dataframe "drive_wheels_co
drive_wheels_counts = df["drive-wheels"].value_counts().to_frame()
drive_wheels_counts.rename(columns={'drive-wheels': 'value_counts'}, inplace=True) #
drive_wheels_counts
```

	value_counts
fwd	120
rwd	76
4wd	9

```
drive_wheels_counts.index.name = 'drive-wheels'
drive_wheels_counts
```

	value_counts
drive-wheels	
fwd	120
rwd	76
4wd	9

```
# Example for enginee
#df.info()
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


# After examining the value counts of the engine location, we see that engine locatio
# This is because we only have three cars with a rear engine and 198 with an engine i
```

	value_counts
engine-location	
front	202
rear	3

```
## Basic of grouping ###
# The "groupby" method groups data by different categories.
# The data is grouped based on one or several variables, and analysis is performed or
# For example, let's group by the variable "drive-wheels". We see that there are 3 di

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

```
# If we want to know, on average, which type of drive wheel is most valuable, we can
# We can select the columns 'drive-wheels', 'body-style' and 'price', then assign it
df_group_one = df[['drive-wheels','body-style','price']]

# We can then calculate the average price for each of the different categories of dat
# grouping results
group_one = df_group_one.groupby(['drive-wheels', 'body-style'],as_index=False).mean(
group_one

# 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
# We can convert the dataframe to a pivot table using the method "pivot" to create a
```

	drive-wheels	body-style	price
0	4wd	hatchback	0.0
1	4wd	sedan	0.0
2	4wd	wagon	0.0
3	fwd	convertible	0.0
4	fwd	hardtop	0.0
5	fwd	hatchback	0.0
6	fwd	sedan	0.0
7	fwd	wagon	0.0
8	rwd	convertible	0.0
9	rwd	hardtop	0.0
10	rwd	hatchback	0.0
11	rwd	sedan	0.0
12	rwd	wagon	0.0

```
# we will leave the drive-wheels variable as the rows of the table, and pivot body-st

grouped_pivot = group_one.pivot(index= 'drive-wheels',columns='body-style')
grouped_pivot
```

	price				
body-style	convertible	hardtop	hatchback	sedan	wagon
drive-wheels					
4wd	NaN	NaN	0.0	0.0	0.0
fwd	0.0	0.0	0.0	0.0	0.0
rwd	0.0	0.0	0.0	0.0	0.0

```
#
grouped_pivot = grouped_pivot.fillna(0) #fill missing values with 0
grouped_pivot
```

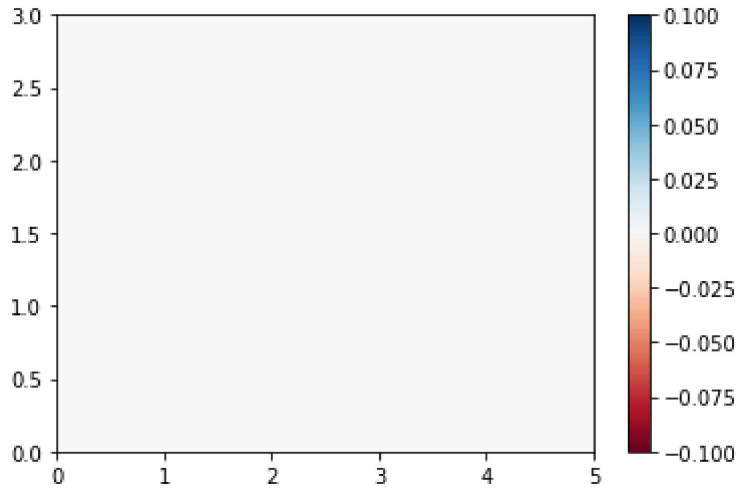
	price				
body-style	convertible	hardtop	hatchback	sedan	wagon
drive-wheels					
4wd	0.0	0.0	0.0	0.0	0.0
fwd	0.0	0.0	0.0	0.0	0.0
rwd	0.0	0.0	0.0	0.0	0.0

```
# use group by funcation to fian an average "price" of each car baed on body-style
df_gtest2 = df[["body-style", "price"]]
group_test_bodystyle = df_gtest2.groupby(['body-style'], as_index= False).mean()
group_test_bodystyle
```

	body-style	price
0	convertible	0.0
1	hardtop	0.0
2	hatchback	0.0
3	sedan	0.0
4	wagon	0.0

```
# Variables: Drive Wheels and Body Style vs. Price

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

%matplotlib inline
import matplotlib.pyplot as plt

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

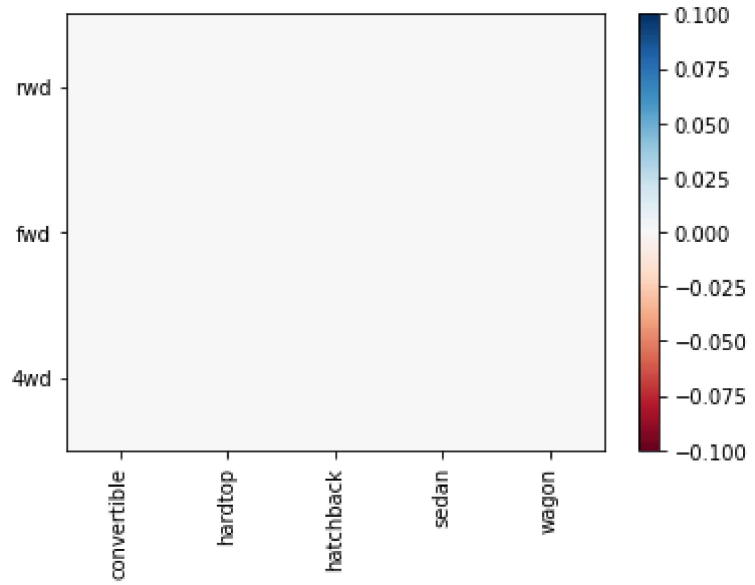
Download




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

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()
```

Download




```
### Correlation and causstion

# 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: h

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

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

  # The Pearson Correlation measures the linear dependence between two variables X ar

      # 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.

df.corr()
```

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore
symboling	1.000000	0.465190	-0.531954	-0.357612	-0.232919	-0.541038	-0.227691	-0.105790	-0.130083
normalized-losses	0.465190	1.000000	-0.056518	0.019209	0.084195	-0.370706	0.097785	0.110997	-0.029266
wheel-base	-0.531954	-0.056518	1.000000	0.874587	0.795144	0.589435	0.776386	0.569329	0.488760
length	-0.357612	0.019209	0.874587	1.000000	0.841118	0.491029	0.877728	0.683360	0.606462
width	-0.232919	0.084195	0.795144	0.841118	1.000000	0.279210	0.867032	0.735433	0.559152
height	-0.541038	-0.370706	0.589435	0.491029	0.279210	1.000000	0.295572	0.067149	0.171101
curb-weight	-0.227691	0.097785	0.776386	0.877728	0.867032	0.295572	1.000000	0.850594	0.648485
engine-size	-0.105790	0.110997	0.569329	0.683360	0.735433	0.067149	0.850594	1.000000	0.583798
bore	-0.130083	-0.029266	0.488760	0.606462	0.559152	0.171101	0.648485	0.583798	1.000000
stroke	-0.008689	0.054929	0.160944	0.129522	0.182939	-0.055351	0.168783	0.203094	-0.055909
compression-ratio	-0.178515	-0.114525	0.249786	0.158414	0.181129	0.261214	0.151362	0.028971	0.005201
horsepower	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
peak-rpm	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
city-mpg	-0.035823	-0.218749	-0.470414	-0.670909	-0.642704	-0.048640	-0.757414	-0.653658	-0.584508
highway-L/100km	0.034606	-0.178221	-0.544082	-0.704662	-0.677218	-0.107358	-0.797465	-0.677470	-0.586992
price	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
city-L/100km	-0.030190	0.178527	0.578128	0.711597	0.728044	0.085892	0.836742	0.777077	0.551943
fuel-type-diesel	-0.194311	-0.101437	0.308346	0.212679	0.233880	0.284631	0.217275	0.069594	0.054457
fuel-type-gas	0.194311	0.101437	-0.308346	-0.212679	-0.233880	-0.284631	-0.217275	-0.069594	-0.054457

```
###  P - value  ###
 # What is this P-value? The P-value is the probability value that the correlation be
# which means that we are 95% confident that the correlation between the variables is

# 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
```

```
from scipy import stats

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

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 =
```

```
The Pearson Correlation Coefficient is nan  with a P-value of P = nan


/opt/python/envs/default/lib/python3.8/site-packages/scipy/stats/stats.py:4023: Pea
  warnings.warn(PearsonRConstantInputWarning())
```

```
## Anova: Analyst of variance  ###
# The Analysis of Variance (ANOVA) is a statistical method used to test whether there
# F-test score: ANOVA assumes the means of all groups are the same, calculates how mu
 #and reports it as the F-test score. A larger score means there is a larger differen
  # P-value: P-value tells how statistically significant our calculated score value i

# If our price variable is strongly correlated with the variable we are analyzing,
  #we expect ANOVA to return a sizeable F-test score and a small p-value.

#Since ANOVA analyzes the difference between different groups of the same variable, t
 # Because the ANOVA algorithm averages the data automatically, we do not need to tak
```

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

grouped_test2= df_group_one[['drive-wheels', 'price']].groupby(['drive-wheels'])
grouped_test2.head(2)
```

	drive-wheels	price
0	rwd	0.0
1	rwd	0.0
3	fwd	0.0
4	4wd	0.0
5	fwd	0.0
9	4wd	0.0

```
df_group_one
```

	drive-wheels	body-style	price
0	rwd	convertible	0.0
1	rwd	convertible	0.0
2	rwd	hatchback	0.0
3	fwd	sedan	0.0
4	4wd	sedan	0.0
...	...	...	...
200	rwd	sedan	0.0
201	rwd	sedan	0.0
202	rwd	sedan	0.0
203	rwd	sedan	0.0
204	rwd	sedan	0.0

205 rows × 3 columns

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

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

```
# We can use the function 'f_oneway' in the module 'stats' to obtain the F-test score

f_val, p_val = stats.f_oneway(grouped_test2.get_group('fwd')['price'], grouped_test2.

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

 # This is a great result with a large F-test score showing a strong correlation and
  #But does this mean all three tested groups are all this highly correlated?
```

```
ANOVA results: F= nan , P = nan


/opt/python/envs/default/lib/python3.8/site-packages/scipy/stats/stats.py:3650: F_c
  warnings.warn(F_onewayConstantInputWarning())
```

```
# fwd and rwd
f_val, p_val = stats.f_oneway(grouped_test2.get_group('fwd')['price'], grouped_test2.
```

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

```
ANOVA results: F= 129.41115759339715 , P = 2.6524240289951807e-23
```

```
# 4wd and rwd

f_val, p_val = stats.f_oneway(grouped_test2.get_group('4wd')['price'], grouped_test2.

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

```
ANOVA results: F= 8.879065438652509 , P = 0.00378273257813761
```

```
################################################
```

```
##        Lab-4 Model Development   #####
# After complete this lab you will able to Develope predication model

# In this section, we will develop several models that will predict the price of the
# This is just an estimate but should give us an objective idea of how much the car s

# some question we want to ask in this model
 # Do i know if the delear offerin 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 observa
```

```
##    Liner regrestion and Multiple Linear regrestion  ####
#  Simple Linear Regrestion ###
 # Simple Linear Regression is a method to help us understand the relationship betwee
    #  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 (de
# Linear Funcation  :  Ythat = a + bx
   #  a refers to the intercept of the regression line, in other words: the value of
   # b refers to the slope of the regression line, in other words: the value with whi
```

```
# Load the models for liniar regrations
```

```
from sklearn.linear_model import LinearRegression
lm = LinearRegression()     # create Linear regression object lm
lm
```

```
LinearRegression()
```

```
# if we want to look at how highway-mpg can help us predict car price.

x = df[['highway-L/100km']] # create a linear function with "highway-mpg" as the pred
y = df['price']         # "price" as the response variable

# fit the linear model using the highway-mpg
lm.fit(x,y)

# output predication
Yhat=lm.predict(x)
Yhat[0:5]
```

```
lm.intercept_   # What is the value of the intercept (a)?
```

```
37470.66014138312
```

```
lm.coef_        # What is the value of the slope (b)?
```

```
#  What is the equation of the predicted line? You can use x and yhat or "engine-size
#yhat = 38423 + (-821.7333783219254)*x
# price = 38423 + (-821.7333783219254)*highway-mpg
```

```
# Multiple Linear regrestion  ###
  # 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 Mu
  # Multiple Linear Regression is very similar to Simple Linear Regression, but this
  # Most of the real-world regression models involve multiple predictors
# Equtions: Yhat=a+b_1X_1+b_2X_2+b_3X_3+b_4X_4
```

```
from sklearn.linear_model import LinearRegression
lm = LinearRegression()  # create linear regrestion model
lm
```

```
LinearRegression()
```

```
df['price'] = df['price'].isnull()
df["price"]=df["price"].astype(float, copy=True)
```

```
df.dtypes
```

```
# Let's develop a model using these variables as the predictor variables.
Z = df[['horsepower','curb-weight', 'engine-size', 'highway-L/100km']]
# print(df)
#X = df[['price']]
lm.fit(Z,df['price'])      # Fit the linear model using the four above-mentioned vari
lm.intercept_   # What is the value of the intercept(a)?
```

```
0.0
```

```
lm.coef_         # What are the values of the coefficients (b1, b2, b3, b4)?
```

```
# Create and train a Multiple Linear Regression model "lm2" where the response variab
# and the predictor variable is "normalized-losses" and "highway-mpg


lm2 = LinearRegression()
lm2.fit(df[['normalized-losses', 'highway-L/100km']], df['price'])
```

```
LinearRegression()
```

```
lm2.coef_
```

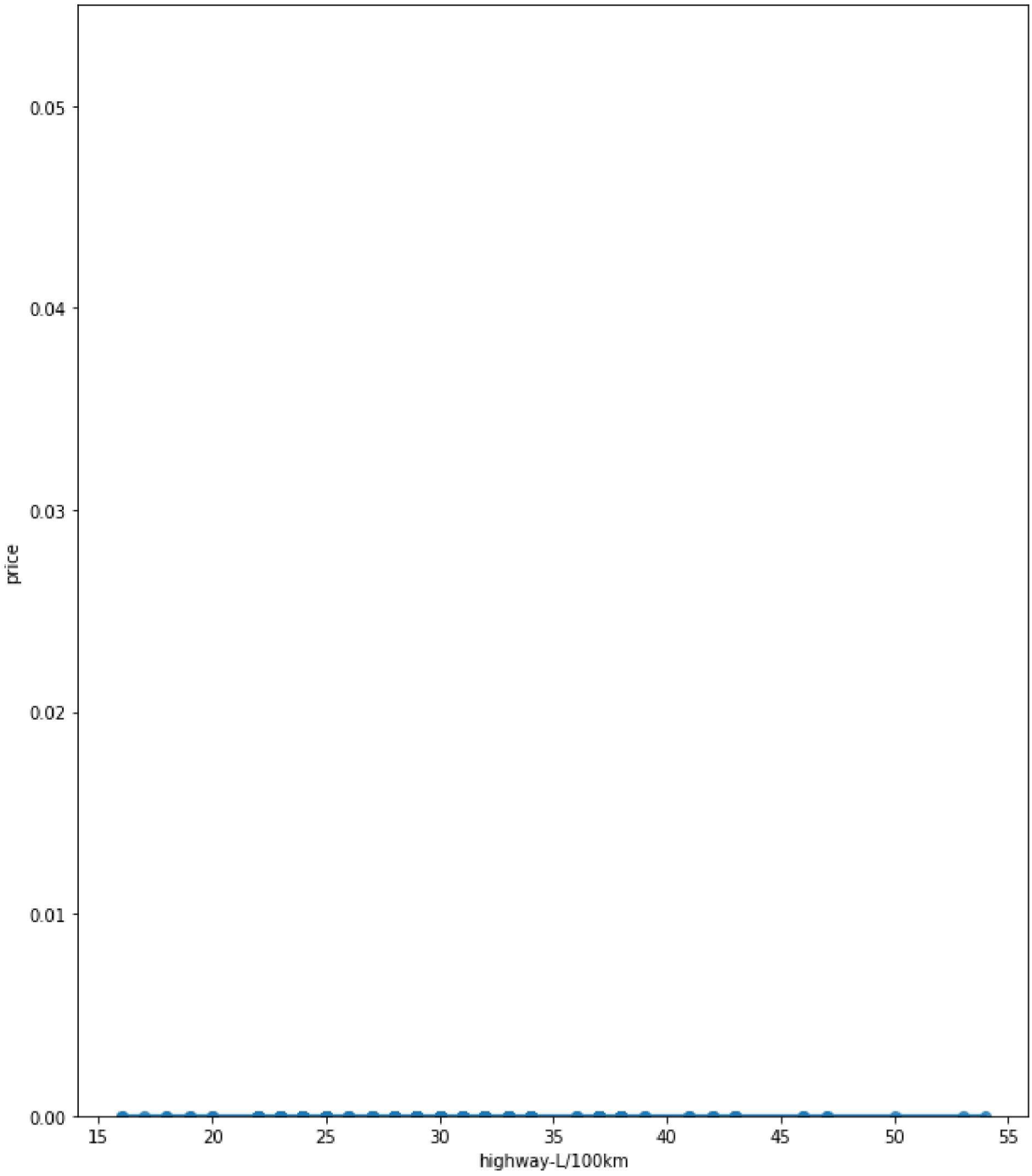
```
################################

# Model evaluation using visualiz
```

```
    # how do we evaluate our own models and choose the best one ?
    # one way to do this is by using a visualization, import the visualization package

#import seaborn as sns
# %matplotlib inline

# Regrastion plot ###
```

```
# Let the visualize highway-mpg as  potential predicator variable of price

import seaborn as sns
%matplotlib inline

width = 10
height = 12
plt.figure(figsize=(width,height))
sns.regplot(x="highway-L/100km", y="price", data=df)
plt.ylim(0,)

# What is this plot telling us?
    # We can see from this residual plot that the residuals are not randomly spread ar
      #leading us to believe that maybe a non-linear model is more appropriate for thi
```

```
(0.0, 0.05500000000000001)
```

Download

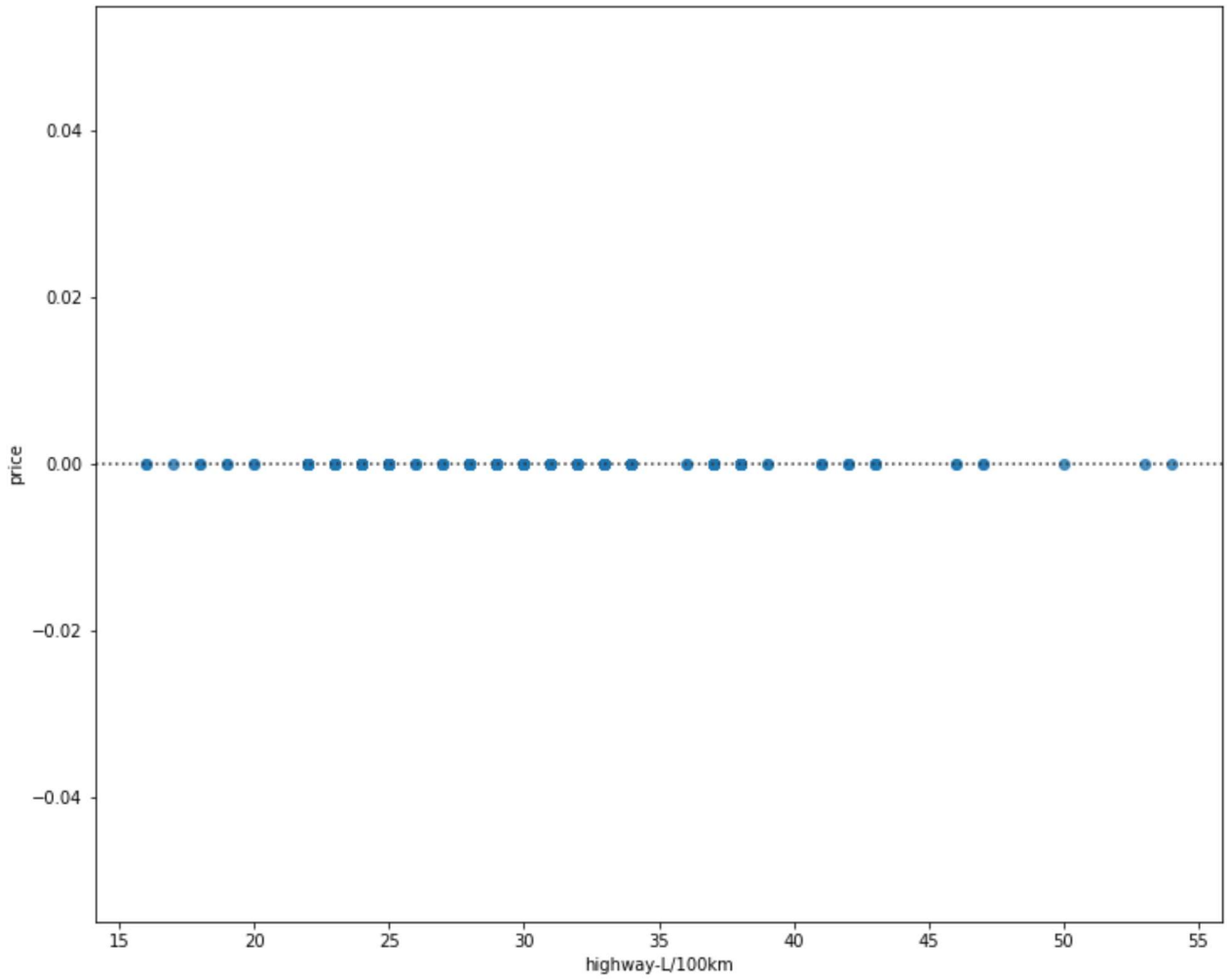



```
# Given the regression plots above, is "peak-rpm" or "highway-mpg" more strongly corr
# Use the method ".corr()" to verify your answer.
df[['peak-rpm', 'highway-L/100km', 'price']].corr()

# The variable "highway-mpg" has a stronger correlation with "price", it is approxima
#You can verify it using the following comma
```

	peak-rpm	highway-L/100km	price

```
import seaborn as sns
%matplotlib inline

width = 12
height = 10
plt.figure(figsize=(width, height))
sns.residplot(df['highway-L/100km'], df['price'])
plt.show()
```

Download




```
/opt/python/envs/default/lib/python3.8/site-packages/seaborn/_decorators.py:36: F
  warnings.warn(
```

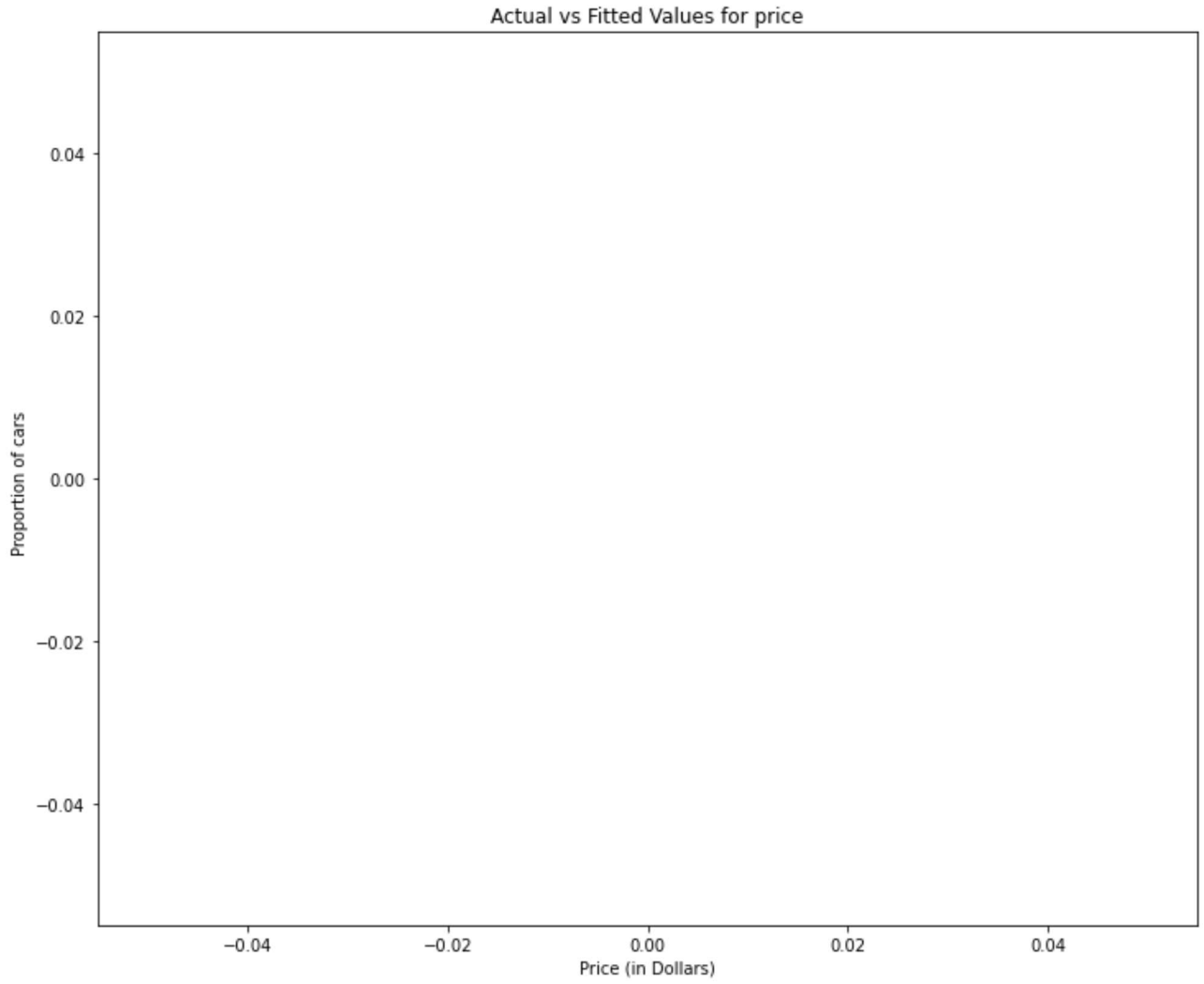
```
# Multiple Linear Regression ####
 # How do we visualize a model for Multiple Linear Regression?
   #This gets a bit more complicated because you can't visualize it with regression
   # one way to look at the fit of the model is by looking at the distribution plot.
```

```
# first let make predication
```

```
Y_hat = lm.predict(Z)
```

```
import seaborn as sns
%matplotlib inline


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()


# We can see that the fitted values are reasonably close to the actual values since t
# However, there is definitely some room for improvement.
```

Download




```
/opt/python/envs/default/lib/python3.8/site-packages/seaborn/distributions.py:2619:
  warnings.warn(msg, FutureWarning)
/opt/python/envs/default/lib/python3.8/site-packages/seaborn/distributions.py:316:
  warnings.warn(msg, UserWarning)
/opt/python/envs/default/lib/python3.8/site-packages/seaborn/distributions.py:2619:
  warnings.warn(msg, FutureWarning)
/opt/python/envs/default/lib/python3.8/site-packages/seaborn/distributions.py:316:
  warnings.warn(msg, UserWarning)
```

```
# We saw earlier that a linear model did not provide the best fit while using "highwa
        # Let's see if we can try fitting a polynomial model to the data instead
# We will use the following function to plot the data:

def plotpolly(model, independent_variable, dependent_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))
```

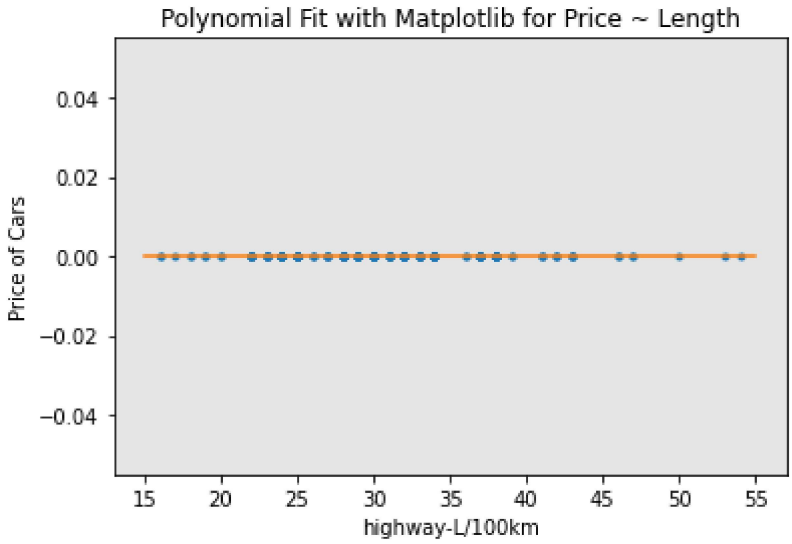
```
    fig = plt.gcf()
    plt.xlabel(Name)
    plt.ylabel('Price of Cars')

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

```
# Let's get the varaible

x = df['highway-L/100km']
y = df['price']

# Let's fit the polynomial using the function polyfit, then use the function poly1d t
 # # Here we use a polynomial of the 3rd order (cubic)

f = np.polyfit(x,y,11)  # Create 11 order polynomial model with the variables x and y
p = np.poly1d(f)
print(f)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

```
# Let the plot the funcation
plotpolly(p,x,y,'highway-L/100km')
```

Download




```
np.polyfit(x,y,11)

# Conclusion ###
```

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

```
# We can perform a polynomial transform on multiple features. First, we import the mo
from sklearn.preprocessing import PolynomialFeatures

# We create a PolynomialFeatures object of degree 2:

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

```
PolynomialFeatures()
```

```
Z_pr=pr.fit_transform(Z)

# n the original data, there are 201 samples and 4 features
Z.shape  # result :(205,4)

# er the transformation, there are 205 samples and 15 features.
Z_pr.shape
```

```
(205, 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

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 mode
# Create a pipeline that standardizes the data, then produce a prediction using a lin
Input=[('scale',StandardScaler()), ('polynomial', PolynomialFeatures(include_bias=Fal

#  We input the list as an argument to the pipeline constructor
pipe=Pipeline(Input)
pipe
```

```
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 m

# Then, we can normalize the data, perform a transform and fit the model simultaneous

Z = Z.astype(float)
pipe.fit(Z,y)
```

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

```
# Similarly, we can normalize the data, perform a transform and produce a prediction
ypipe=pipe.predict(Z)
ypipe[0:4]
```

```
## Mesure for in-sample Evaluation  ###

# R-squared

# R squared, also known as the coefficient of determination, is a measure to indicate

# The value of the R-squared is the percentage of variation of the response variable

# Mean Squared Error (MSE)

# The Mean Squared Error measures the average of the squares of errors. That is, the
```

```
x_train = x_train.reshape(-1, 1)
x_test = x_test.reshape(-1,1)
```

```
AttributeError: 'DataFrame' object has no attribute 'reshape'
```

```
#highway_mpg_fit
lm.fit(x,y)
# Find the R^2
print('The R-square is: ', lm.score(x, y))

# We can say that ~49.659% of the variation of the price is explained by this simple
```

```
ValueError: Expected 2D array, got 1D array instead:
array=[27 27 26 30 22 25 25 25 20 22 29 29 28 28 25 22 22 20 53 43 43 41 38 30
 38 38 38 30 30 24 54 38 42 34 34 34 34 33 33 33 33 28 31 29 43 43 29 19
```

```
 19 17 31 38 38 38 38 23 23 23 23 32 32 32 32 42 32 27 39 25 25 25 25 18
 18 16 16 24 41 38 38 30 30 32 24 24 24 32 32 30 30 37 50 37 37 37 37 37
 37 37 37 34 34 22 22 25 25 23 25 24 33 24 25 24 33 24 25 24 33 24 41 30
 38 38 38 30 24 27 25 25 25 28 31 31 28 28 28 28 26 26 36 31 31 37 33 32
 25 29 32 31 29 23 39 38 38 37 32 32 37 37 36 47 47 34 34 34 34 29 29 30
 30 30 30 30 30 34 33 32 32 32 24 24 24 24 46 34 46 34 34 42 32 29 29 24
 38 31 28 28 28 28 22 22 28 25 23 27 25].
Reshape your data either using array.reshape(-1, 1) if your data has a single featu
```

```
# Let's calculate the MSE:

# We can predict the output i.e., "yhat" using the predict method, where X is the inp

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

```
ValueError: Expected 2D array, got 1D array instead:
array=[27 27 26 30 22 25 25 25 20 22 29 29 28 28 25 22 22 20 53 43 43 41 38 30
 38 38 38 30 30 24 54 38 42 34 34 34 34 33 33 33 33 28 31 29 43 43 29 19
 19 17 31 38 38 38 38 23 23 23 23 32 32 32 32 42 32 27 39 25 25 25 25 18
 18 16 16 24 41 38 38 30 30 32 24 24 24 32 32 30 30 37 50 37 37 37 37 37
 37 37 37 34 34 22 22 25 25 23 25 24 33 24 25 24 33 24 25 24 33 24 41 30
 38 38 38 30 24 27 25 25 25 28 31 31 28 28 28 28 26 26 36 31 31 37 33 32
 25 29 32 31 29 23 39 38 38 37 32 32 37 37 36 47 47 34 34 34 34 29 29 30
 30 30 30 30 30 34 33 32 32 32 24 24 24 24 46 34 46 34 34 42 32 29 29 24
 38 31 28 28 28 28 22 22 28 25 23 27 25].
Reshape your data either using array.reshape(-1, 1) if your data has a single featu
```

```
# Let's import the function mean_squared_error from the module metrics:
from sklearn.metrics import mean_squared_error

        # We can compare the predicted results with the actual results
mse = mean_squared_error(df['price'], Y_hat)
print('The mean square error of price and predicted value is: ', mse)
```

```
The mean square error of price and predicted value is:  0.0
```

```
# Module 2  Multiple Linear regresion 2
# Let's calculate the R^2:

# 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:  1.0
```

```
# Let's calculate the MSE.We produce a prediction
Y_predict_multifit = lm.predict(Z)

#  We compare the predicted results with the actual results:
print('The mean square error of price and predicted value using multifit is: ', mean_
```

```
The mean square error of price and predicted value using multifit is:  0.0
```

```
# Modul 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 differ

from sklearn.metrics import r2_score
# We apply the function to get the value of R^2:

r_squared = r2_score(y, p(x))
print('The R-square value is: ', r_squared)
# We can say that ~67.419 % of the variation of price is explained by this polynomial
# Wrong result by used wrong data, but equation and method right
```

```
The R-square value is:  1.0
```

```
# MSE
# We can also calculte the MSE:
mean_squared_error(df['price'], p(x))
```

```
0.0
```

```
# Predication and Decision making
  # 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
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline

# create new input
new_input=np.arange(1, 100, 1).reshape(-1, 1)
```

```
# Fit the method
lm.fit(x,y)
lm
```

```
ValueError: Expected 2D array, got 1D array instead:
array=[27 27 26 30 22 25 25 25 20 22 29 29 28 28 25 22 22 20 53 43 43 41 38 30
 38 38 38 30 30 24 54 38 42 34 34 34 34 33 33 33 33 28 31 29 43 43 29 19
 19 17 31 38 38 38 38 23 23 23 23 32 32 32 32 42 32 27 39 25 25 25 25 18
 18 16 16 24 41 38 38 30 30 32 24 24 24 32 32 30 30 37 50 37 37 37 37 37
 37 37 37 34 34 22 22 25 25 23 25 24 33 24 25 24 33 24 25 24 33 24 41 30
 38 38 38 30 24 27 25 25 25 28 31 31 28 28 28 28 26 26 36 31 31 37 33 32
 25 29 32 31 29 23 39 38 38 37 32 32 37 37 36 47 47 34 34 34 34 29 29 30
 30 30 30 30 30 34 33 32 32 32 24 24 24 24 46 34 46 34 34 42 32 29 29 24
 38 31 28 28 28 28 22 22 28 25 23 27 25].
Reshape your data either using array.reshape(-1, 1) if your data has a single featu
```

```
Y_hat=lm.predict(new_input)
Y_hat[0:1]
```

```
ValueError: X has 1 features, but LinearRegression is expecting 4 features as input
```

```
plt.plot(new_input, Y_hat)
plt.show()
```

```
ValueError: x and y must have same first dimension, but have shapes (99, 1) and (20
```

```
## Modle Evaluation ####
```

```
import pandas as pd
import numpy as np
import seaborn as sns
%matplotlib inline
#import matplotlib.pyplot as plt
```

```
# Read the online file by the URL provides above, and assign it to variable "df"
df = pd.read_csv("New_data.csv")
#dset = sns.load_dataset("df")
df.head(100)
df.tail(10)
```

	Unnamed: 0	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	compressio ratio
195	195	-1	74	volvo	std	four	wagon	rwd	front	104.3	...	9.5
196	196	-2	103	volvo	std	four	sedan	rwd	front	104.3	...	9.5
197	197	-1	74	volvo	std	four	wagon	rwd	front	104.3	...	9.5
198	198	-2	103	volvo	turbo	four	sedan	rwd	front	104.3	...	7.5
199	199	-1	74	volvo	turbo	four	wagon	rwd	front	104.3	...	7.5
200	200	-1	95	volvo	std	four	sedan	rwd	front	109.1	...	9.5
201	201	-1	95	volvo	turbo	four	sedan	rwd	front	109.1	...	8.7
202	202	-1	95	volvo	std	four	sedan	rwd	front	109.1	...	8.8
203	203	-1	95	volvo	turbo	four	sedan	rwd	front	109.1	...	23.0
204	204	-1	95	volvo	turbo	four	sedan	rwd	front	109.1	...	9.5

10 rows × 30 columns

```
df=df._get_numeric_data()
df.head()
```

	Unnamed: 0	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore	stroke	compres ratio
0	0	3	122	88.6	0.811148	0.886584	0.816054	2548	130	3.47	2.68	9.0
1	1	3	122	88.6	0.811148	0.886584	0.816054	2548	130	3.47	2.68	9.0
2	2	1	122	94.5	0.822681	0.905947	0.876254	2823	152	2.68	3.47	9.0
3	3	2	164	99.8	0.848630	0.915629	0.908027	2337	109	3.19	3.40	10.0
4	4	2	164	99.4	0.848630	0.918396	0.908027	2824	136	3.19	3.40	8.0

```
# Library for plotting
from ipywidgets import interact, interactive, fixed, interact_manual

# Funcation plotting
def DistributionPlot(RedFunction, BlueFunction, RedName, BlueName, Title):
    width = 12
    height = 10
    plt.figure(figsize=(width, height))

    ax1 = sns.distplot(RedFunction, hist=False, color="r", label=RedName)
```

```
    ax2 = sns.distplot(BlueFunction, hist=False, color="b", label=BlueName, ax=ax1)

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

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

```
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='Pr
    plt.ylim([-10000, 60000])
    plt.ylabel('Price')
    plt.legend()
```

```
## Part 1. Traning and Testing

    #  An important step in testing your model is to split your data into training ar
    # We will place the target data price in a separate dataframe y_data

y_data = df['price']

# Drop price data in dataframe x_data:
x_data=df.drop('price',axis=1)
```

```
# Now, we randomly split our data into training and testing data using the function t

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=0.20, r

print("number of test sample:", y_test.shape[0])
print("number of training sample:", y_train.shape[0])

# The test_size parameter sets the proportion of data that is split into the testing
```

```
number of test sample: 41
number of training sample: 164
```

```
# Exercise 1:
 # Use the function "train_test_split" to split up the dataset such that 40% of the d
  # The output of the function should be the following: "x_train1" , "x_test1", "y_tr

from sklearn.model_selection import train_test_split

x_train1, x_test1, y_train1, y_test1 = train_test_split (x_data, y_data, test_size=0.

print("number of test sample:", y_test1.shape[0])
print("number of training sample:", y_train1.shape[0])
```

```
number of test sample: 82
number of training sample: 123
```

```
# Let import LinearRegrastion from the module linear_model

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

from sklearn.linear_model import LinearRegression

# Let we create Linear regrastion object

lre = LinearRegression()

# We fit the model using the feture horsepower

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

# Let calculate the R^2 on the test data

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

```
# result : -0.001755664252970135

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

# As Result we can see R^2 is much smaller using the test data compared to the traing
```

```
0.0
```

```
## Cross validation score ###
##############################################

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

Rcross
```

```
# We can calculate the average and standard deviation of our estimate

print("The mean of the folds are", Rcross.mean(), "and the standard deviation is" , F
```

```
The mean of the folds are -0.04278412307512103 and the standard deviation is 0.0648
```

```
# We can use negative squared error as a score by setting the parameter 'scoring' met

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

```
# Cross-Validation predict ####
# 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

from sklearn.model_selection import cross_val_predict

yhat= cross_val_predict(lre, x_data[['horsepower']], y_data, cv=16)
yhat[0:5]
```

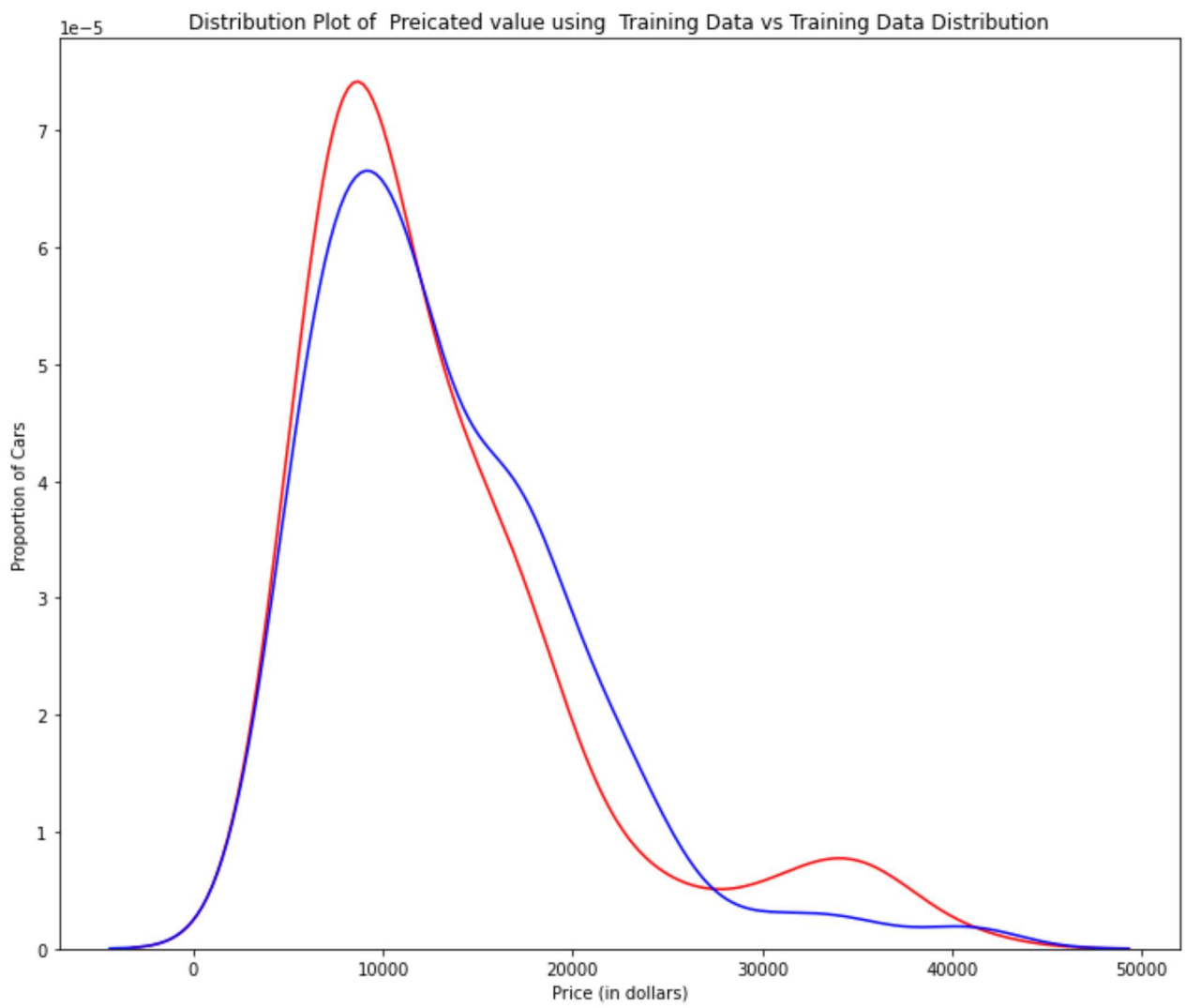
```
# We input the object, the feature "horsepower", and the target data y_data. The para
```

```
#### part 2: Overfiting, Underfiting, and Model selection ###
########################################################################

# Let's create Multiple Linear Regression objects and train the model using 'horsepow
lr = LinearRegression()
lr.fit(x_train[['horsepower', 'curb-weight','engine-size','highway-L/100km']], y_trai

# Prediction using training data
yhat_train = lr.predict(x_train[['horsepower','curb-weight','engine-size','highway-L/
yhat_train[0:10]
```

```
# Predication using test data

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

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

import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

# Let eximine the distribute of the predicted value of trainig data

Title = 'Distribution Plot of  Preicated value using  Training Data vs Training Data
DistributionPlot(y_train, yhat_train, "Actual values(Train)", "Predicted Values(Trair
```

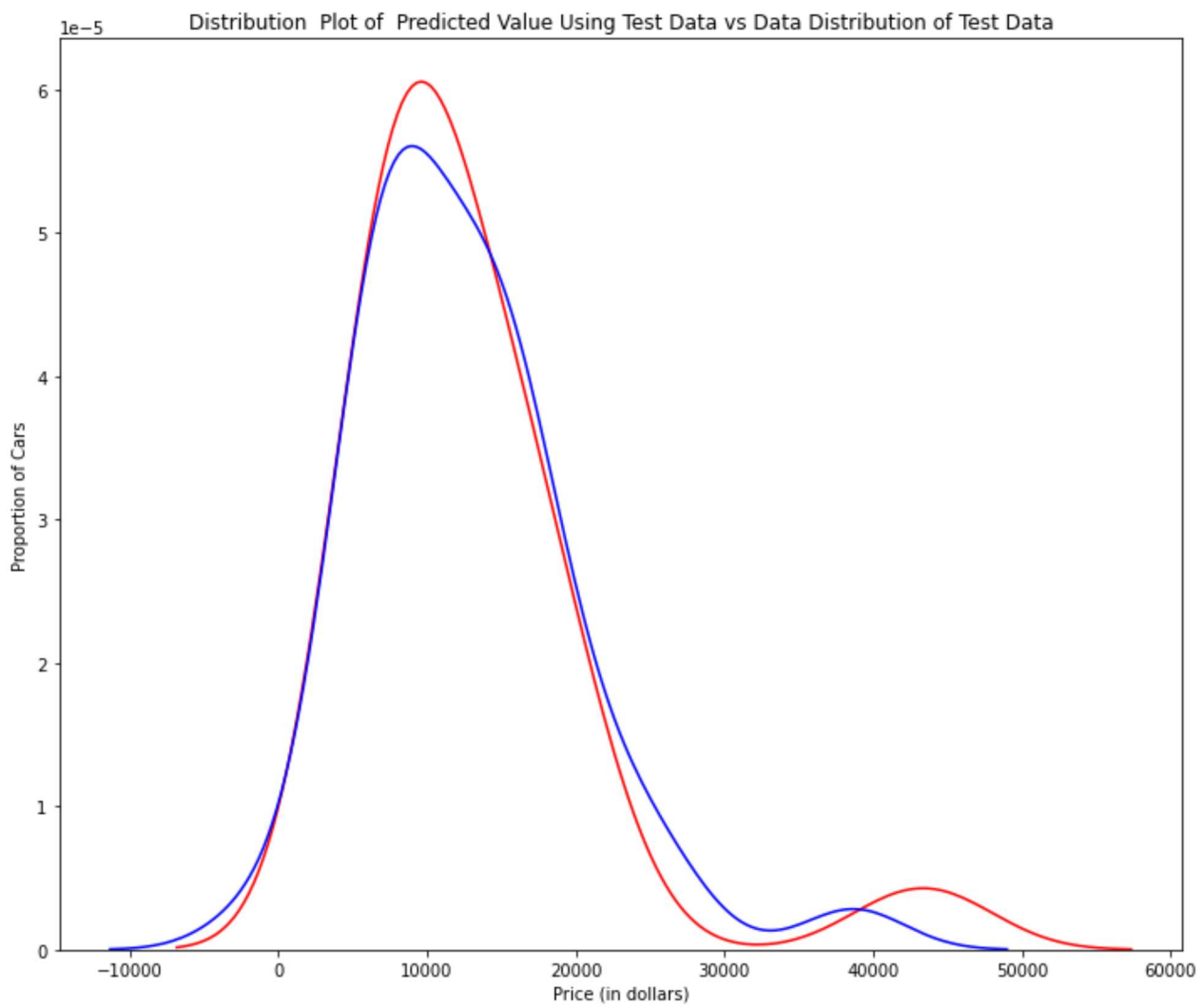
Download




```
/opt/python/envs/default/lib/python3.8/site-packages/seaborn/distributions.py:261
  warnings.warn(msg, FutureWarning)
/opt/python/envs/default/lib/python3.8/site-packages/seaborn/distributions.py:261
  warnings.warn(msg, FutureWarning)
```

```
# Figure 2: Plot of predicted value using the test data compared to the actual values
Title='Distribution  Plot of  Predicted Value Using Test Data vs Data Distribution of
DistributionPlot(y_test,yhat_test,"Actual Values (Test)","Predicted Values (Test)",Ti
```

Download




```
/opt/python/envs/default/lib/python3.8/site-packages/seaborn/distributions.py:261
  warnings.warn(msg, FutureWarning)
/opt/python/envs/default/lib/python3.8/site-packages/seaborn/distributions.py:261
  warnings.warn(msg, FutureWarning)
```

```
# Ovefitting

from sklearn.preprocessing import PolynomialFeatures

# 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
        # Let's create a degree 5 polynomial model.

x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.45, r

# We will perform a degree 5 polynomial transformation on the feature 'horsepower'.
# create a polynomialFeatures Object "pr" of degree 5
pr = PolynomialFeatures(degree=5)

# Tranform the training and testing samples for the features 'horsepower'
```

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

# How many dimenstions does the new features have?
x_train_pr # result (112,6)
```

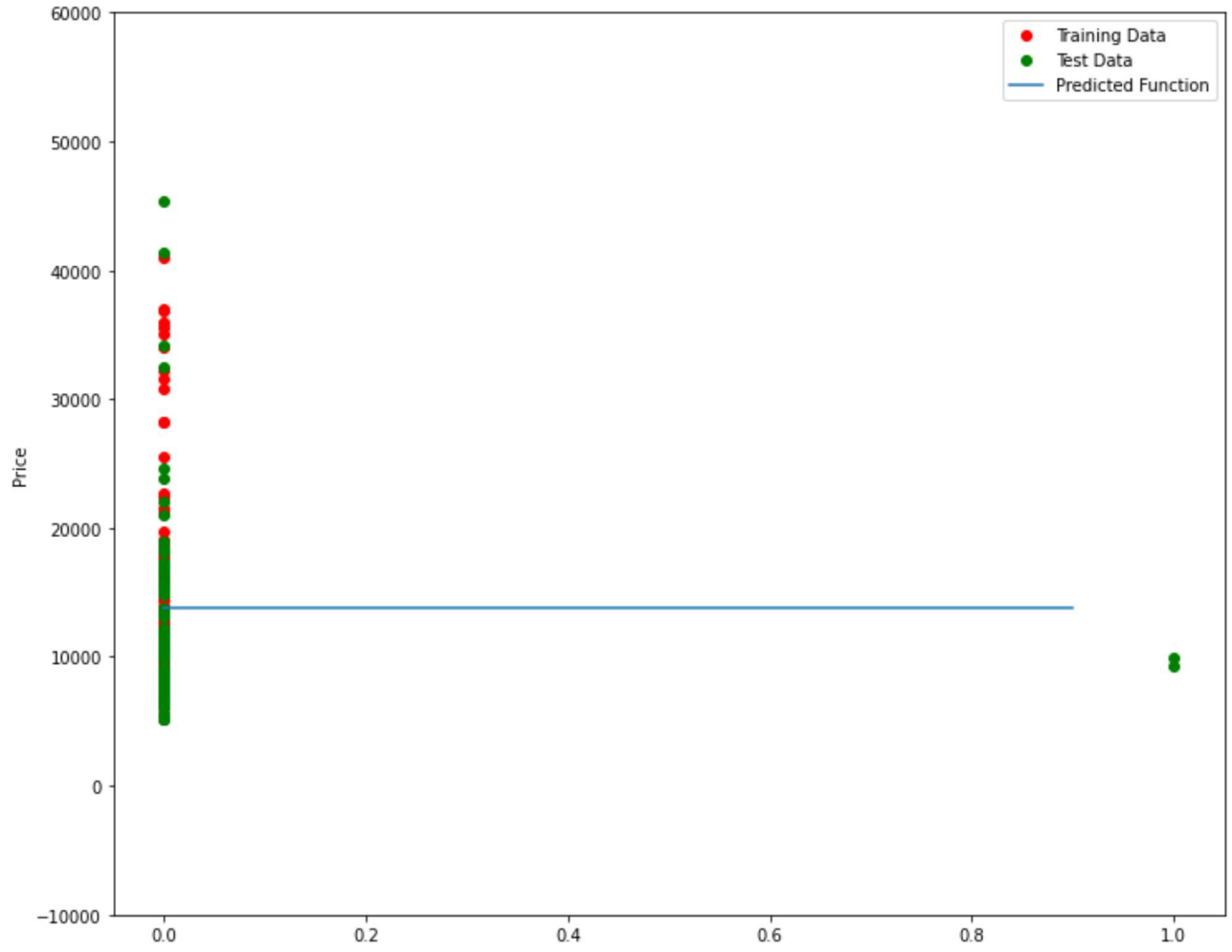
```
# Now, let's create a Linear Regression model "poly" and train the object using the m

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

# We can see the output of our model using the method "predict." We assign the values
yhat= poly.predict(x_test_pr)
yhat[0:5]
```

```
# Let's take the first five predicted values and compare it to the actual targets
print("Predicted values:", yhat[0:5])
print("True values:", y_test[0:5].values)
```

```
Predicted values: [13740.01008351 13740.01008351 13740.01008351 13740.01008351
 13740.01008351]
True values: [ 6795. 15750. 15250.  5151.  9995.]
```

```
# We will use the function "PollyPlot" that we defined at the beginning of the lab to

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, gre
```

Download




```
poly.score(x_train_pr, y_train)  # As result 0
poly.score(x_test_pr, y_test)   # Result: -0.027067250301669565

# The lower the R^2, the worse the model. A negative R^2 is a sign of overfitting.
```

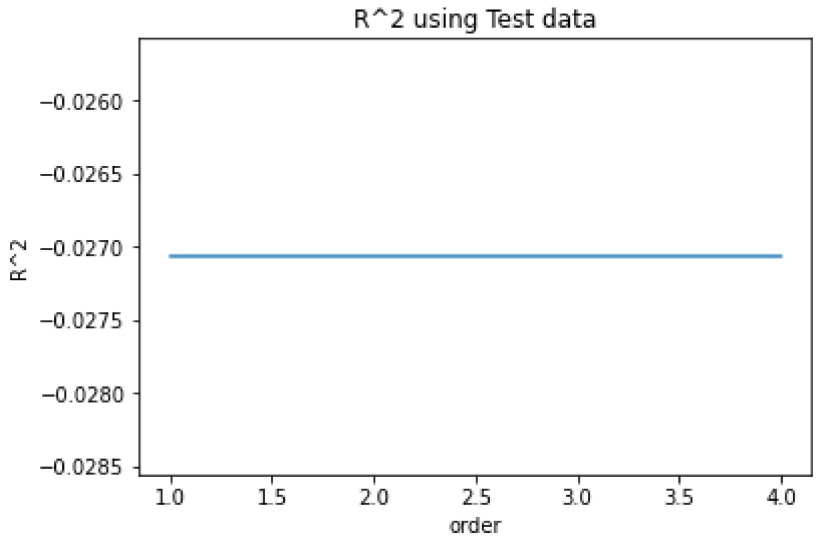
```
-0.027067250301669565
```

```
# Let's see how the R^2 changes on the test data for different order polynomials and
Rsqu_test =[]

order = [1,2,3,4]
for n in order:
    pr = PolynomialFeatures(degree=1)
    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')
plt.text(3,0.75, 'Maximum R^2')

# We see the R^2 gradually increases until an order three polynomial is used. Then, t
```

```
Text(3, 0.75, 'Maximum R^2')
```

Download

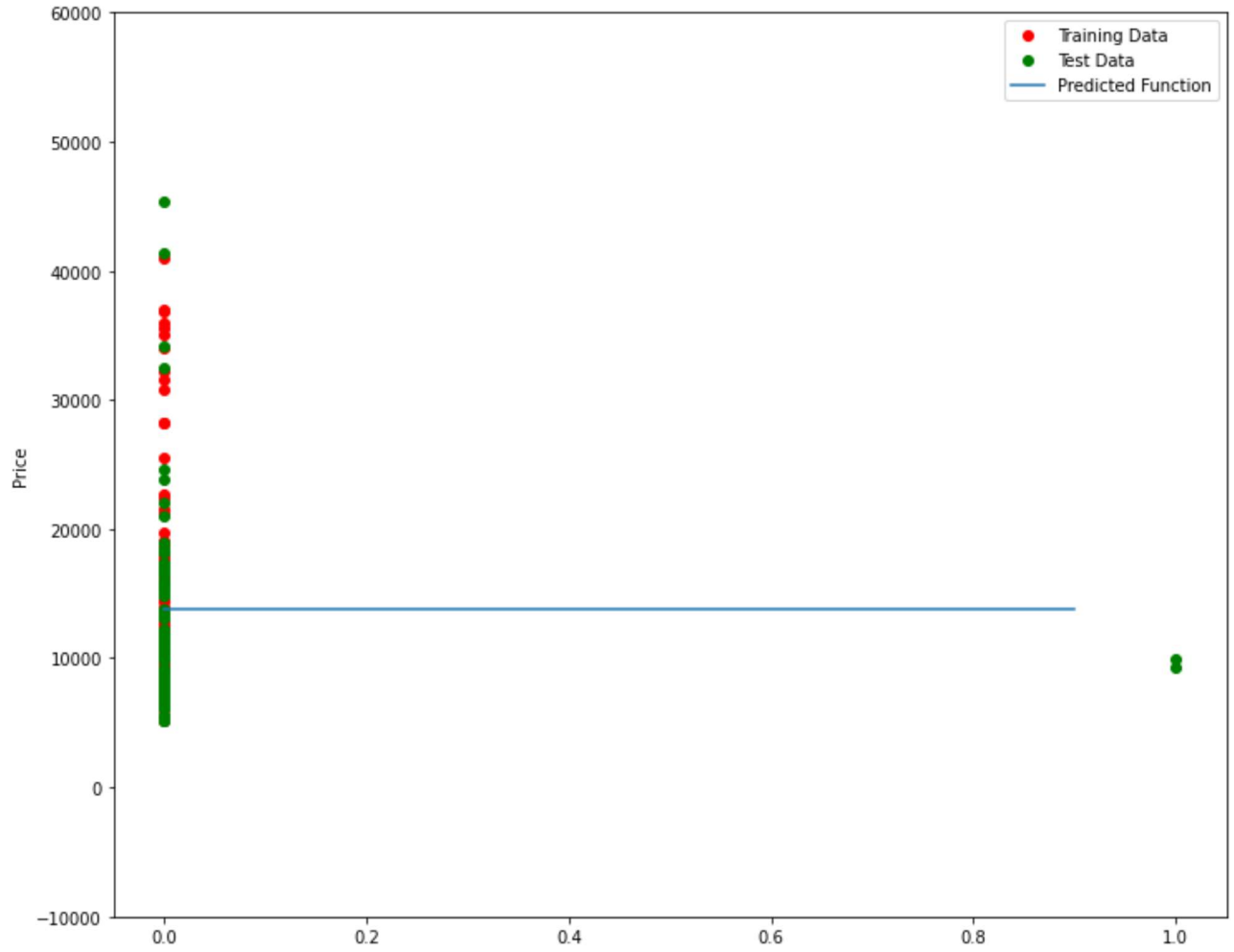



```
# The following interface allows you to experiment with different polynomial orders c

def f(order, test_data):
    x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=tes
    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,
```

```
interact(f, order=(0, 6, 1), test_data=(0.05, 0.95, 0.05))
```

```
<function __main__.f(order, test_data)>
```

Download




```
# Use the method "predict" to predict an output on the polynomial features,
# then use the function "DistributionPlot" to display the distribution of the predict

yhat_test = poly.predict(x_test_pr)

Title = 'Distribution plot of predictd value using test data vs  data distributed of
DistributionPlot(y_test, yhat_test, "Actual values (Test)","Preddicated values(Test)'
```

```
ValueError: X has 2 features, but LinearRegression is expecting 6 features as input
```

```
# 4F Using the distribution plot above, describe (in words) the two regions where the

# The predicted value is higher than actual value for cars where the price $10,000 ra
```

```
### Part 3 : Ridge Rigrastion #####

# we will review Ridge Regression and see how the parameter alpha changes the model.
```

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

pr = PolynomialFeatures(degree=2)
x_train_pr=pr.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size', 'hig
x_test_pr=pr.fit_transform(x_test[['horsepower', 'curb-weight','engine-size', 'highwa
```

```
# Let import ridge from the module the linear models

from sklearn.linear_model import Ridge

# Let create ridge regression object, setting the regularization parameter alpha to 0

RigeModel=Ridge(alpha=1)

# Like regular regression, you can fit the model using the fit method
RigeModel.fit(x_train_pr,y_train)
```

```
Ridge(alpha=1)
```

```
# Similarly, you can obtain a prediction

yhat = RigeModel.predict(x_test_pr)

# Let's compare the first five predicted samples to our test set:
print('predicted:', yhat[0:4])
print('test set :', y_test[0:4].values)
```

```
predicted: [ 5854.60634601 19329.67660502 15386.61896507  3243.12328324]
test set : [ 6795. 15750. 15250.  5151.]
```

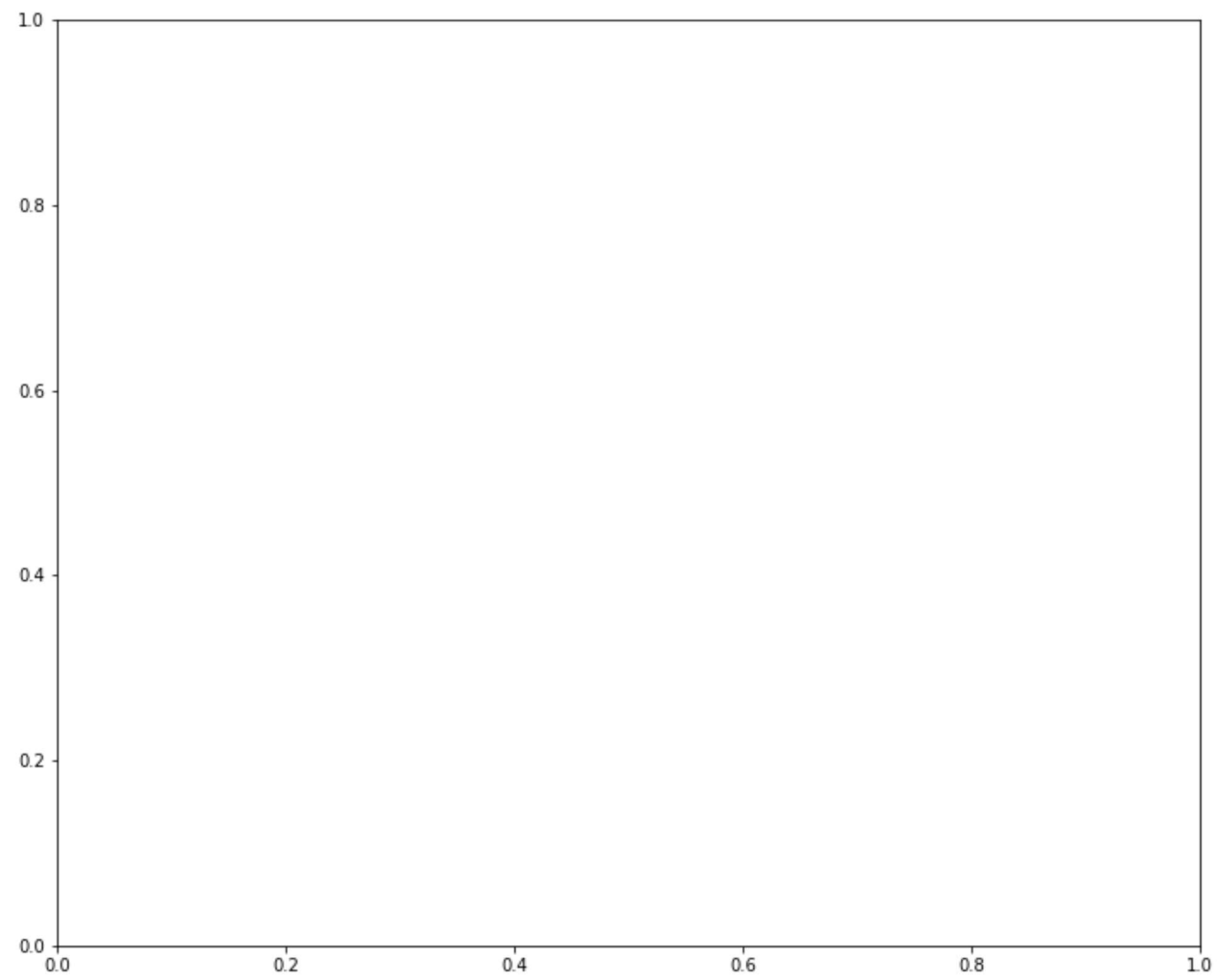
```
# We select the value of alpha that minimizes the test error.
# To do so, we can use a for loop. We have also created a progress bar to see how mar

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_tr

pbar.set_postfix({"The score": test_score, "Train Score": train_score})

Rsqu_test.append(test_score)
Rsqu_train.append(train_score)
```

```
  0%|          | 0/1000 [00:00<?, ?it/s]  6%|▌         | 60/1000 [00:00<00:01, 599.
```

```
# We can plot out the value of R^2 for different alphas:

width = 12
height = 10
plt.figure(figsize=(width,height))

plt.plot(Alpha, Rsqu_test, label='Validation Data')
plt.plot(Alpha, Rsqu_train, label='training data')
plt.xlabel('Alpha')
plt.ylabel('R^2')
plt.legend()
```

Download




```
ValueError: x and y must have same first dimension, but have shapes (1000,) and (
```

```
## Part 4 Grid Search ####

# The term alpha is a hyperparameter.
# Sklearn has the class GridSearchCV to make the process of finding the best hyperpar
```

```
# Let's import GridSearchCV from the module model_selection
from sklearn.model_selection import GridSearchCV

# We create a dictionalry for parameter values
parameters1 = [{'alpha': [0.001,0.1,1, 10, 100, 1000, 10000, 100000, 100000]}]
parameters1
```

```
[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 100000]}]
```

```
# Create ridge regrestion object
```

```
RR = Ridge()
RR

# Create a ridge grid search object:
Grid1 = GridSearchCV(RR, parameters1, cv=4)

# Fit the model
Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-L/100km']], y_
```

```
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 = Grid1.best_estimator_
BestRR
```

```
Ridge(alpha=1000)
```

```
# We now test our model on the test data
BestRR.score(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-L/100km']],
```

```
0.7141122250975075
```