Advance Regression Techniques

In this assignment you will learn a lot on various advance regression techniques like lasso, ridge, ElasticNet, polynomial regression, and also you will learn hyperparameter tuning technique called GridSearchCV

So buddy role up your sleeves and get ready for various fun activities

Problem statement: Car Price Prediction

The solution is divided into the following sections:

- Data understanding and exploration
- Data cleaning
- Data preparation
- Model building and evaluation

1. Data Understanding and Exploration

points= 20

Let's first have a look at the dataset and understand the size, attribute names etc.

In [1]:

```
# Importing necessary libraries

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import linear_model
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.model_selection import GridSearchCV

# import os

# hide warnings
import warnings
warnings.filterwarnings('ignore')
```

In [2]:

```
# reading the CarPrice_Assignment
cars = 
```

In [3]:

```
# summary of the dataset: 205 rows, 26 columns, no null values
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 26 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   car_ID            205 non-null    int64  
 1   symboling         205 non-null    int64  
 2   CarName           205 non-null    object 
 3   fueltype          205 non-null    object 
 4   aspiration        205 non-null    object 
 5   doornumber        205 non-null    object 
 6   carbody           205 non-null    object 
 7   drivewheel        205 non-null    object 
 8   enginelocation    205 non-null    object 
 9   wheelbase         205 non-null    float64
 10  carlength         205 non-null    float64
 11  carwidth          205 non-null    float64
 12  carheight         205 non-null    float64
 13  curbweight        205 non-null    int64  
 14  enginetype        205 non-null    object 
 15  cylindernumber    205 non-null    object 
 16  enginesize        205 non-null    int64  
 17  fuelsystem        205 non-null    object 
 18  boreratio         205 non-null    float64
 19  stroke            205 non-null    float64
 20  compressionratio  205 non-null    float64
 21  horsepower        205 non-null    int64  
 22  peakrpm           205 non-null    int64  
 23  citympg           205 non-null    int64  
 24  highwaympg        205 non-null    int64  
 25  price             205 non-null    float64
dtypes: float64(8), int64(8), object(10)
memory usage: 41.8+ KB
None
```

In [4]:

```
# head
```

Out[4]:

	car_ID	symboling	CarName	fueltype	aspiration	doornumber	carbody	drivewheel	en
0	1	3	alfa-romero giulia	gas	std	two	convertible	rwd	
1	2	3	alfa-romero stelvio	gas	std	two	convertible	rwd	
2	3	1	alfa-romero Quadrifoglio	gas	std	two	hatchback	rwd	
3	4	2	audi 100 ls	gas	std	four	sedan	fwd	
4	5	2	audi 100ls	gas	std	four	sedan	4wd	

5 rows × 26 columns

Understanding the unique value distribution

Here we will check various attributes in a feature and its contribution in the dataset.

In [5]:

```
#Check each symboling attribute's count
```

Out[5]:

```
0     67
1     54
2     32
3     27
-1    22
-2     3
Name: symboling, dtype: int64
```

From above output we can see that symboling parameter in cars daraset shows -2 (least risky) to +3 most risky but most of the cars are 0,1,2.

In [6]:

```
#Check each aspiration attribute's count
```

Out[6]:

```
std      168
turbo     37
Name: aspiration, dtype: int64
```

aspiration: An (internal combustion) engine property showing whether the oxygen intake is through standard (atmospheric pressure)or through turbocharging (pressurised oxygen intake)

In [7]:

```
#Check each drivewheel attribute's count
```

Out[7]:

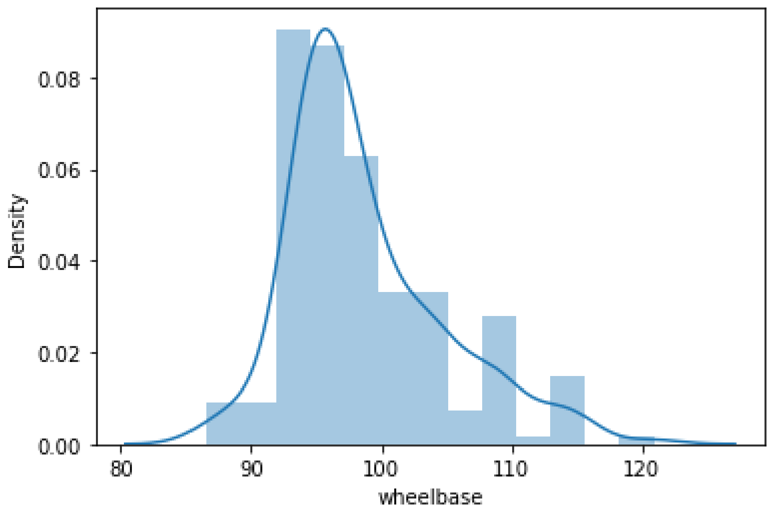
```
fwd    120
rwd     76
4wd      9
Name: drivewheel, dtype: int64
```

drivewheel: frontwheel, rarewheel or four-wheel drive

Now plot distribution plot for **wheelbase**: distance between centre of front and rarewheels

In [8]:

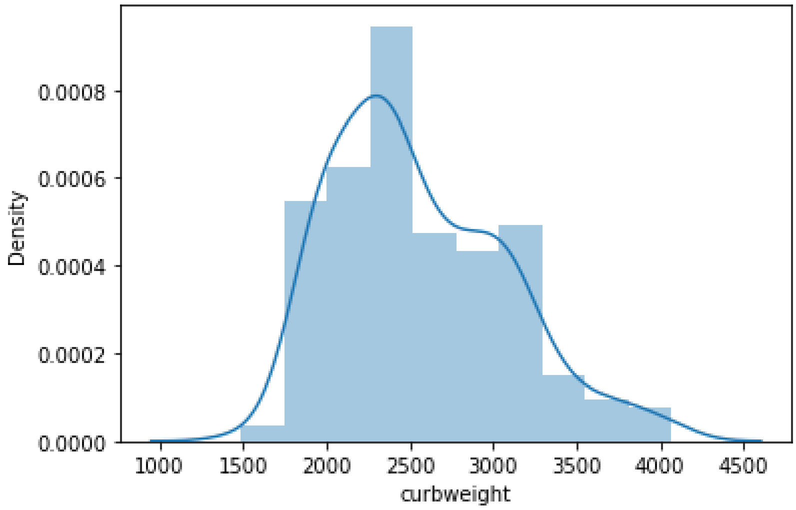
```
# plot wheetbase distribution
```



plot distribution plot for **curbweight**: weight of car without occupants or baggage

In [9]:

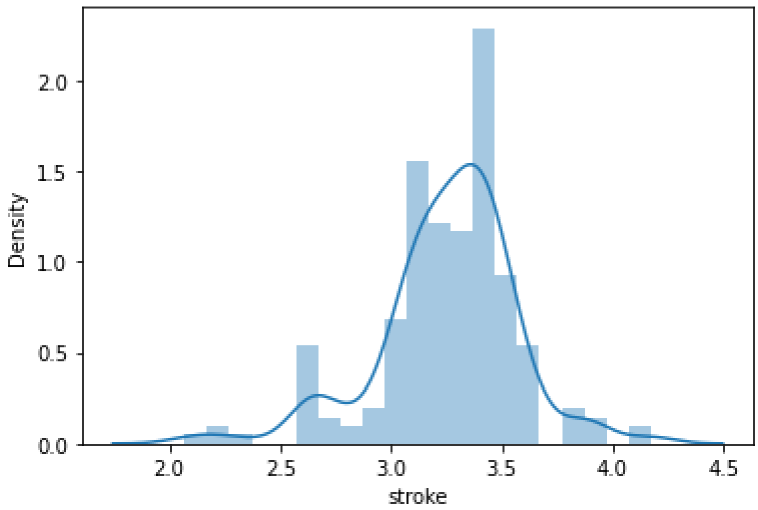
```
# plot curbweight distribution
```



plot distribution plot for **stroke**: volume of the engine (the distance traveled by the piston in each cycle)

In [10]:

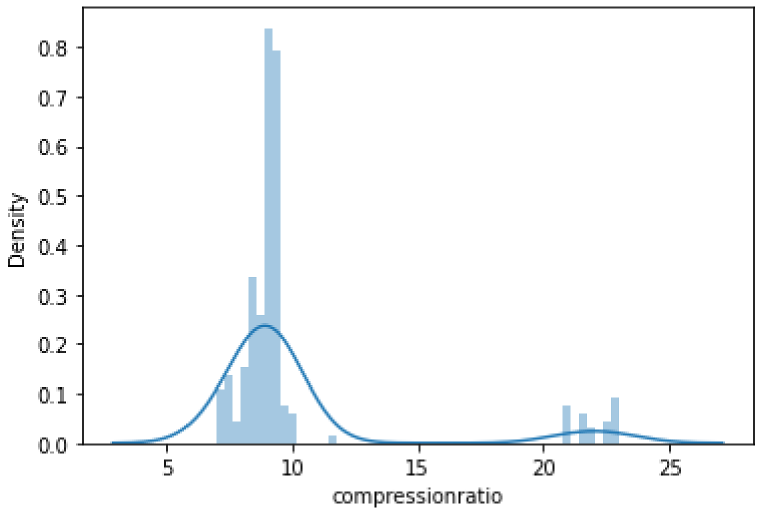
```
# plot stroke dsitribution
```



Now plot distribution plot for **compressionration**: ratio of volume of compression chamber at largest capacity to least capacity

In [11]:

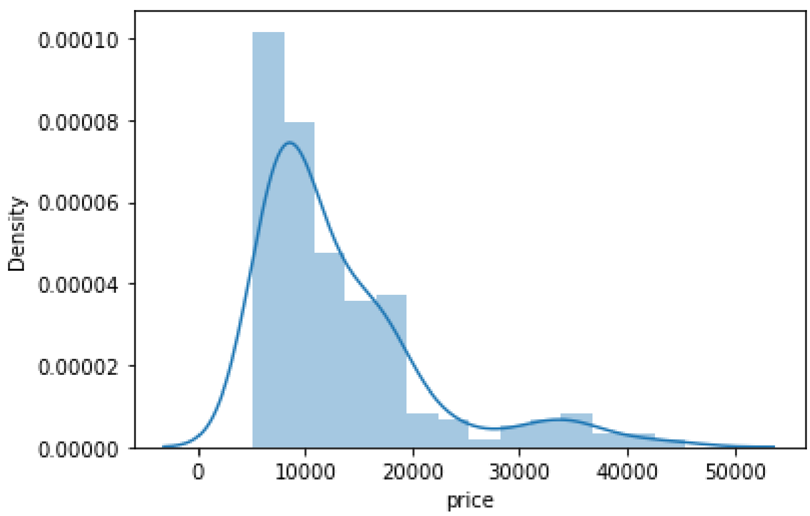
```
# plot compressionratio distribution
```



Now lets see distribution plot for target variable: **price** of car

In [12]:

```
# Price distribution
```



Data Exploration

To perform linear regression, the (numeric) target variable should be linearly related to *at least one another numeric variable*. Let's see whether that's true in this case.

We'll first subset the list of all (independent) numeric variables, and then make a **pairwise plot**.

In [13]:

```
# all numeric (float and int) variables in the dataset
cars_numeric =

#head
cars_numeric.head()
```

Out[13]:

	car_ID	symboling	wheelbase	carlength	carwidth	carheight	curbweight	enginesize	borer
0	1	3	88.6	168.8	64.1	48.8	2548	130	
1	2	3	88.6	168.8	64.1	48.8	2548	130	
2	3	1	94.5	171.2	65.5	52.4	2823	152	
3	4	2	99.8	176.6	66.2	54.3	2337	109	
4	5	2	99.4	176.6	66.4	54.3	2824	136	

Here, although the variable `symboling` is numeric (int), we'd rather treat it as categorical since it has only 6 discrete values. Also, we do not want 'car_ID'.

In [14]:

```
# dropping symboling and car_ID


cars_numeric.head()
```

Out[14]:

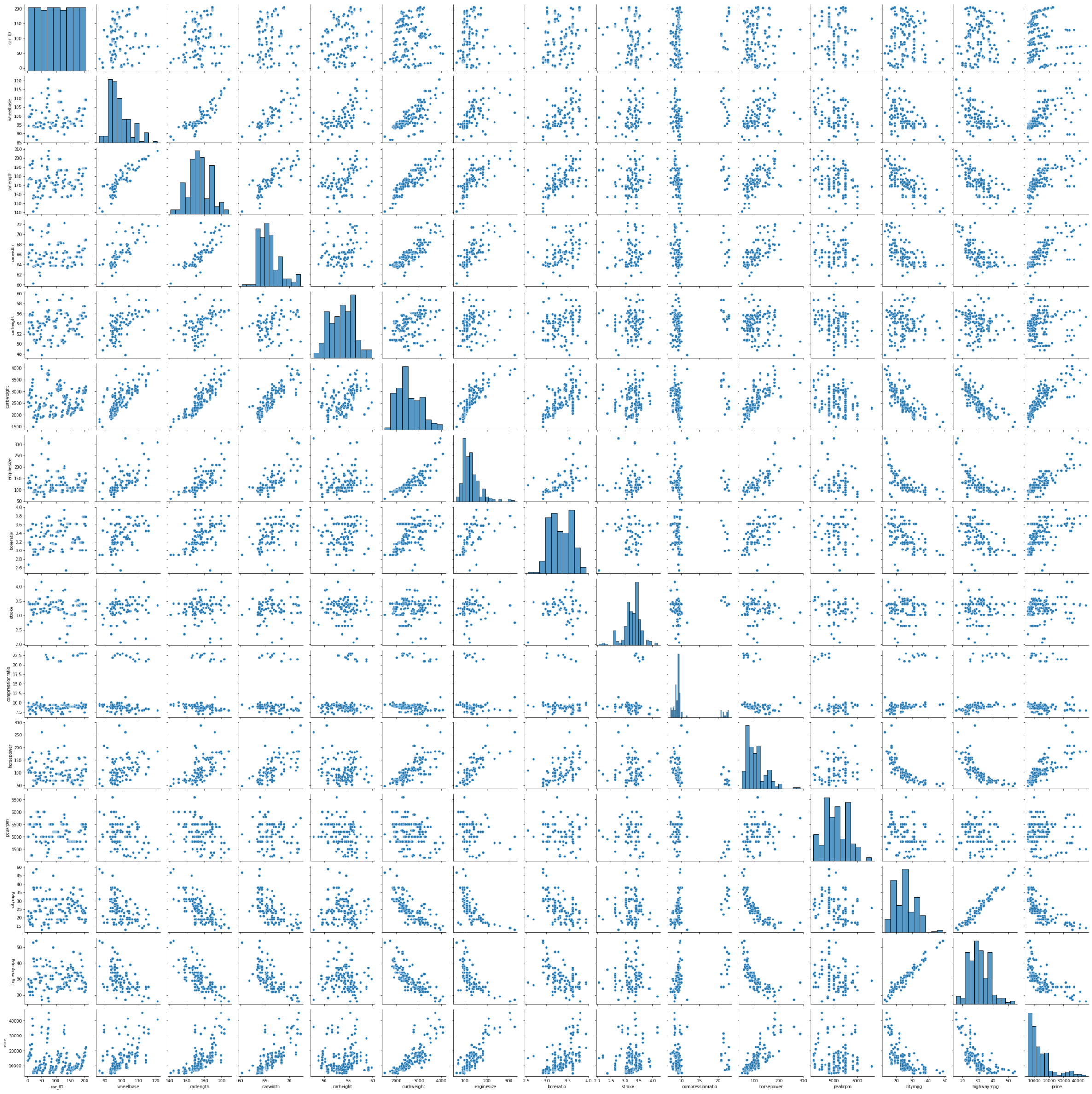
	car_ID	wheelbase	carlength	carwidth	carheight	curbweight	enginesize	boreratio	stroke
0	1	88.6	168.8	64.1	48.8	2548	130	3.47	2.68
1	2	88.6	168.8	64.1	48.8	2548	130	3.47	2.68
2	3	94.5	171.2	65.5	52.4	2823	152	2.68	3.47
3	4	99.8	176.6	66.2	54.3	2337	109	3.19	3.40
4	5	99.4	176.6	66.4	54.3	2824	136	3.19	3.40

Let's now make a pairwise scatter plot and observe linear relationships.

In [15]:

```
# paiwise scatter plot for all variables in cars_numeric
```

<Figure size 1440x720 with 0 Axes>



This is quite hard to read, and we can rather plot correlations between variables. Also, a heatmap is pretty useful to visualise multiple correlations in one plot.

In [16]:

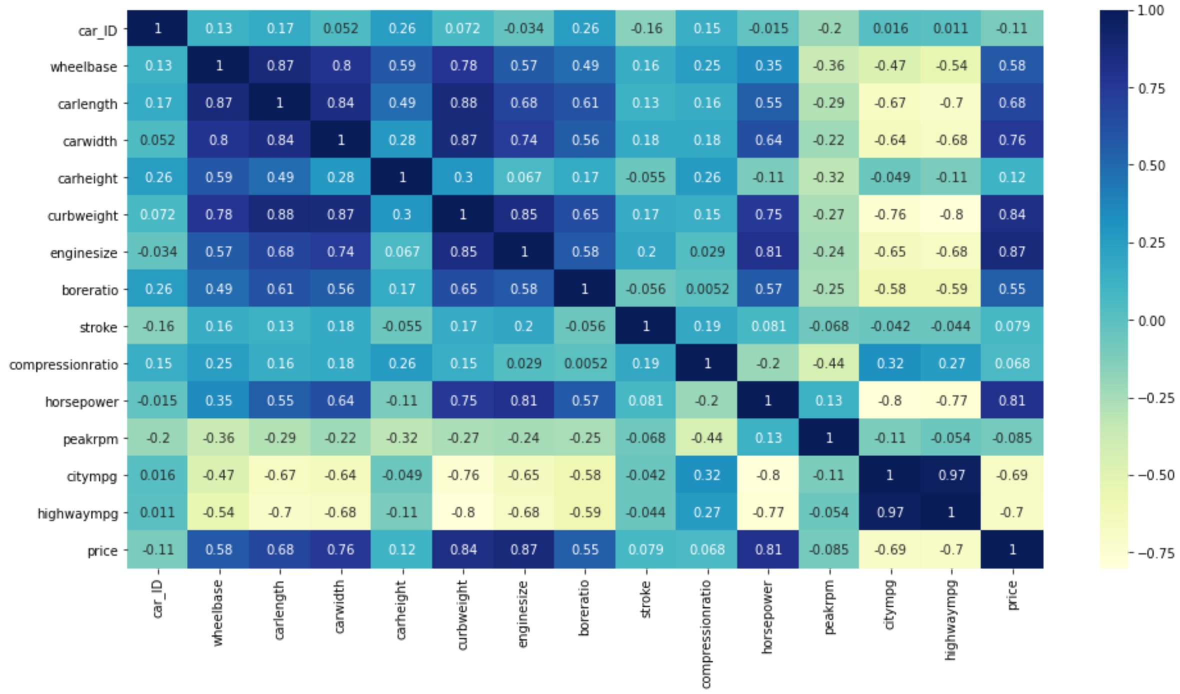
```
# correlation matrix
cor = 
#print cor
```

Out[16]:

	car_ID	wheelbase	carlength	carwidth	carheight	curbweight	enginesize
car_ID	1.000000	0.129729	0.170636	0.052387	0.255960	0.071962	-0.033930
wheelbase	0.129729	1.000000	0.874587	0.795144	0.589435	0.776386	0.569329
carlength	0.170636	0.874587	1.000000	0.841118	0.491029	0.877728	0.683360
carwidth	0.052387	0.795144	0.841118	1.000000	0.279210	0.867032	0.735433
carheight	0.255960	0.589435	0.491029	0.279210	1.000000	0.295572	0.067149
curbweight	0.071962	0.776386	0.877728	0.867032	0.295572	1.000000	0.850594
enginesize	-0.033930	0.569329	0.683360	0.735433	0.067149	0.850594	1.000000
boreratio	0.260064	0.488750	0.606454	0.559150	0.171071	0.648480	0.583774
stroke	-0.160824	0.160959	0.129533	0.182942	-0.055307	0.168790	0.203129
compressionratio	0.150276	0.249786	0.158414	0.181129	0.261214	0.151362	0.028971
horsepower	-0.015006	0.353294	0.552623	0.640732	-0.108802	0.750739	0.809769
peakrpm	-0.203789	-0.360469	-0.287242	-0.220012	-0.320411	-0.266243	-0.244660
citympg	0.015940	-0.470414	-0.670909	-0.642704	-0.048640	-0.757414	-0.653658
highwaympg	0.011255	-0.544082	-0.704662	-0.677218	-0.107358	-0.797465	-0.677470
price	-0.109093	0.577816	0.682920	0.759325	0.119336	0.835305	0.874145

In [17]:

```
# plotting correlations on a heatmap

# figure size
plt.figure(figsize=(16,8))

# heatmap


plt.show()
```



The heatmap shows some useful insights:

Correlation of price with independent variables:

- Price is highly (positively) correlated with wheelbase, carlength, carwidth, curbweight, enginesize, horsepower (notice how all of these variables represent the size/weight/engine power of the car)
- Price is negatively correlated to `citympg` and `highwaympg` (-0.70 approximately). This suggest that cars having high mileage may fall in the 'economy' cars category, and are priced lower (think Maruti Alto/Swift type of cars, which are designed to be affordable by the middle class, who value mileage more than horsepower/size of car etc.)

Correlation among independent variables:

- Many independent variables are highly correlated (look at the top-left part of matrix): wheelbase, carlength, curbweight, enginesize etc. are all measures of 'size/weight', and are positively correlated

Thus, while building the model, we'll have to pay attention to multicollinearity (especially linear models, such as linear and logistic regression, suffer more from multicollinearity).

2. Data Cleaning

points= 15

Let's now conduct some data cleaning steps.

We've seen that there are no missing values in the dataset. We've also seen that variables are in the correct format, except `symboling` , which should rather be a categorical variable (so that dummy variable are created for the categories).

Note that it *can* be used in the model as a numeric variable also.

In [18]:

```
# variable formats
```

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

In [19]:

```
# converting symboling to categorical by changing its datatype to object

#printing cars basic informatiomn
```

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

Next, we need to extract the company name from the column `CarName` .

In [20]:

```
# CarName: first few entries (upto 30)
```

Out[20]:

```
0             alfa-romero giulia
1            alfa-romero stelvio
2       alfa-romero Quadrifoglio
3                    audi 100 ls
4                     audi 100ls
5                       audi fox
6                     audi 100ls
7                      audi 5000
8                      audi 4000
9            audi 5000s (diesel)
10                      bmw 320i
11                      bmw 320i
12                        bmw x1
13                        bmw x3
14                        bmw z4
15                        bmw x4
16                        bmw x5
17                        bmw x3
18              chevrolet impala
19         chevrolet monte carlo
20            chevrolet vega 2300
21                 dodge rampage
22           dodge challenger se
23                    dodge d200
24            dodge monaco (sw)
25             dodge colt hardtop
26                dodge colt (sw)
27          dodge coronet custom
28             dodge dart custom
29     dodge coronet custom (sw)
Name: CarName, dtype: object
```

Notice that the carname is what occurs before a space, e.g. alfa-romero, audi, chevrolet, dodge, bmx etc.

Thus, we need to simply extract the string before a space. There are multiple ways to do that.

In [21]:

```
# Extracting carname

# Method 1: str.split() by space
carnames =

# Print CarName: first few entries (upto 30)
```

Out[21]:

```
0     alfa-romero
1     alfa-romero
2     alfa-romero
3            audi
4            audi
5            audi
6            audi
7            audi
8            audi
9            audi
10            bmw
11            bmw
12            bmw
13            bmw
14            bmw
15            bmw
16            bmw
17            bmw
18      chevrolet
19      chevrolet
20      chevrolet
21          dodge
22          dodge
23          dodge
24          dodge
25          dodge
26          dodge
27          dodge
28          dodge
29          dodge
Name: CarName, dtype: object
```

In [22]:

```
# Method 2: Use regular expressions
import re

# regex: any alphanumeric sequence before a space, may contain a hyphen
p =

#apply above regex pattern to CarName
carnames =

#print carnames
print()
```

```
0      alfa-romero
1      alfa-romero
2      alfa-romero
3             audi
4             audi
          ...
200          volvo
201          volvo
202          volvo
203          volvo
204          volvo
Name: CarName, Length: 205, dtype: object
```

Let's create a new column to store the compnay name and check whether it looks okay.

In [23]:

```
# New column car_company
cars['car_company'] =
```

In [24]:

```
# look at all values under car_company
```

Out[24]:

```
toyota         31
nissan         17
mazda          15
mitsubishi     13
honda          13
subaru         12
volvo          11
peugeot        11
volkswagen      9
dodge           9
buick           8
bmw             8
plymouth        7
audi            7
saab            6
porsche         4
isuzu           4
chevrolet       3
alfa-romero     3
jaguar          3
vw              2
maxda           2
renault         2
mercury         1
porcshce        1
toyouta         1
vokswagen       1
Nissan          1
Name: car_company, dtype: int64
```

Notice that **some car-company names are misspelled** - vw and vokswagen should be volkswagen, porcshce should be porsche, toyouta should be toyota, Nissan should be nissan, maxda should be mazda etc.

This is a data quality issue, let's solve it.

Reference:[https://kanoki.org/2019/07/17/pandas-how-to-replace-values-based-on-conditions/ (https://kanoki.org/2019/07/17/pandas-how-to-replace-values-based-on-conditions/)](https://kanoki.org/2019/07/17/pandas-how-to-replace-values-based-on-conditions/)

In [25]:

```
# replacing misspelled car_company names using loc

# volkswagen

# porsche

# toyota

# nissan

# mazda
```

In [26]:

```
# again print all the values under car_company
```

Out[26]:

```
toyota         32
nissan         18
mazda          17
honda          13
mitsubishi     13
subaru         12
volkswagen     12
volvo          11
peugeot        11
dodge           9
buick           8
bmw             8
plymouth        7
audi            7
saab            6
porsche         5
isuzu           4
alfa-romero     3
chevrolet       3
jaguar          3
renault         2
mercury         1
Name: car_company, dtype: int64
```

The `car_company` variable looks okay now. Let's now drop the car name variable.

In [27]:

```
# drop carname variable
cars =
```

In [28]:

```
# car basic information
```

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

In [29]:

```
# cars statistical discription
```

Out[29]:

	car_ID	wheelbase	carlength	carwidth	carheight	curbweight	enginesize	
count	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	2
mean	103.000000	98.756585	174.049268	65.907805	53.724878	2555.565854	126.907317	
std	59.322565	6.021776	12.337289	2.145204	2.443522	520.680204	41.642693	
min	1.000000	86.600000	141.100000	60.300000	47.800000	1488.000000	61.000000	
25%	52.000000	94.500000	166.300000	64.100000	52.000000	2145.000000	97.000000	
50%	103.000000	97.000000	173.200000	65.500000	54.100000	2414.000000	120.000000	
75%	154.000000	102.400000	183.100000	66.900000	55.500000	2935.000000	141.000000	
max	205.000000	120.900000	208.100000	72.300000	59.800000	4066.000000	326.000000	

3. Data Preparation

points= 8

Data Preparation

Let's now prepare the data and build the model.

split into X and y

In [30]:

```
#Define X
X =

# Define y
y =
```

Creating dummy variables for categorical variables

In [31]:

```
# subset all categorical variables
cars_categorical = 

# cars_categorical head
cars_categorical.head()
```

Out[31]:

	symboling	fueltype	aspiration	doornumber	carbody	drivewheel	enginelocation	enginet
0	3	gas	std	two	convertible	rwd	front	d
1	3	gas	std	two	convertible	rwd	front	d
2	1	gas	std	two	hatchback	rwd	front	o
3	2	gas	std	four	sedan	fwd	front	
4	2	gas	std	four	sedan	4wd	front	

In [32]:

```
# convert into dummies
cars_dummies = 

# cars_dummies head
cars_dummies.head()
```

Out[32]:

	symboling_-1	symboling_0	symboling_1	symboling_2	symboling_3	fueltype_gas	aspiratio
0	0	0	0	0	1	1	
1	0	0	0	0	1	1	
2	0	0	1	0	0	1	
3	0	0	0	1	0	1	
4	0	0	0	1	0	1	

5 rows × 55 columns

In [33]:

```
# drop categorical variables from X
X = 
```

In [34]:

```
# concat dummy variables with X
X = 
```

Scaling the features

points= 4

In [35]:

```
from sklearn.preprocessing import scale

# storing column names in cols, since column names are (annoyingly) lost after
# scaling (the df is converted to a numpy array)
cols =

# scaling X and converting to Dtaframe
X =

#renaming X columns as cols
X.columns =

#print columns in X
```

Out[35]:

```
Index(['wheelbase', 'carlength', 'carwidth', 'carheight', 'curbweight',
       'enginesize', 'boreratio', 'stroke', 'compressionratio', 'horsepowe
r',
       'peakrpm', 'citympg', 'highwaympg', 'symboling_-1', 'symboling_0',
       'symboling_1', 'symboling_2', 'symboling_3', 'fueltype_gas',
       'aspiration_turbo', 'doornumber_two', 'carbody_hardtop',
       'carbody_hatchback', 'carbody_sedan', 'carbody_wagon', 'drivewheel_fw
d',
       'drivewheel_rwd', 'enginelocation_rear', 'enginetype_dohcv',
       'enginetype_l', 'enginetype_ohc', 'enginetype_ohcf', 'enginetype_ohc
v',
       'enginetype_rotor', 'cylindernumber_five', 'cylindernumber_four',
       'cylindernumber_six', 'cylindernumber_three', 'cylindernumber_twelv
e',
       'cylindernumber_two', 'fuelsystem_2bbl', 'fuelsystem_4bbl',
       'fuelsystem_idi', 'fuelsystem_mfi', 'fuelsystem_mpfi',
       'fuelsystem_spdi', 'fuelsystem_spfi', 'car_company_audi',
       'car_company_bmw', 'car_company_buick', 'car_company_chevrolet',
       'car_company_dodge', 'car_company_honda', 'car_company_isuzu',
       'car_company_jaguar', 'car_company_mazda', 'car_company_mercury',
       'car_company_mitsubishi', 'car_company_nissan', 'car_company_peugeo
t',
       'car_company_plymouth', 'car_company_porsche', 'car_company_renault',
       'car_company_saab', 'car_company_subaru', 'car_company_toyota',
       'car_company_volkswagen', 'car_company_volvo'],
      dtype='object')
```

Splitting into test train

points= 1

In [36]:

```
# split into train and test with train_size=70% and random_state=100
```

3. Model Building and Evaluation

points= 30

Reference video: https://www.youtube.com/watch?v=9lRv01HDU0s (https://www.youtube.com/watch?v=9lRv01HDU0s)
https://www.youtube.com/watch?v=uiL5Q64yKYE (https://www.youtube.com/watch?v=uiL5Q64yKYE)

Reference site: https://towardsdatascience.com/linear-regression-models-4a3d14b8d368 (https://towardsdatascience.com/linear-regression-models-4a3d14b8d368)

Ridge, Lasso and ElasticNet Regression

Let's now try predicting car prices, a dataset used in simple linear regression, to perform ridge, lasso and elasticNet regression.

To understand there differences please check: https://medium.com/analytics-vidhya/understanding-difference-between-regularization-methods-ridge-lasso-and-elasticnet-in-python-996185296ed2 (https://medium.com/analytics-vidhya/understanding-difference-between-regularization-methods-ridge-lasso-and-elasticnet-in-python-996185296ed2)

Ridge Regression

In [37]:

```
# list of alphas to tune
params = {'alpha': [0.0001, 0.001, 0.01, 0.05, 0.1,
 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 2.0, 3.0,
 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 20, 50, 100, 500, 1000 ]}

#initialising Ridge() function
ridge = 


# defining cross validation folds as 5
folds = 
```

Cross validation and Hyperparameter tuning: GridSearchCV

Reference: https://www.youtube.com/watch?v=0yI0-r3Ly40 (https://www.youtube.com/watch?v=0yI0-r3Ly40)

initialising GridSearchCV function with folowing attributes:

```
estimator = ridge
param_grid = params
scoring= 'neg_mean_absolute_error'
cv = folds
return_train_score=True
verbose = 1
```

In [38]:

```
# Define GridSearchCV


# fit GridSearchCV() with X_train and y_train
```

```
Fitting 5 folds for each of 28 candidates, totalling 140 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent work
ers.
[Parallel(n_jobs=1)]: Done 140 out of 140 | elapsed:    2.2s finished
```

Out[38]:

```
GridSearchCV(cv=5, estimator=Ridge(),
             param_grid={'alpha': [0.0001, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3,
                                   0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 2.0,
3.0,
                                   4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 20, 5
0,
                                   100, 500, 1000]},
             return_train_score=True, scoring='neg_mean_absolute_error',
             verbose=1)
```

In [39]:

```
# Save GridSearchCV results into a dataframe
cv_results =

# filter cv_results with all param_alpha less than or equal to 200
cv_results =

# cv_results head
```

Out[39]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_alpha	params	split0_t
0	0.004426	0.003863	0.008375	0.002172	0.0001	{'alpha': 0.0001}	-28
1	0.003427	0.004212	0.005474	0.004098	0.001	{'alpha': 0.001}	-28
2	0.006818	0.003441	0.007890	0.003982	0.01	{'alpha': 0.01}	-27
3	0.008410	0.002235	0.004733	0.003991	0.05	{'alpha': 0.05}	-26
4	0.005798	0.002851	0.004201	0.003818	0.1	{'alpha': 0.1}	-25

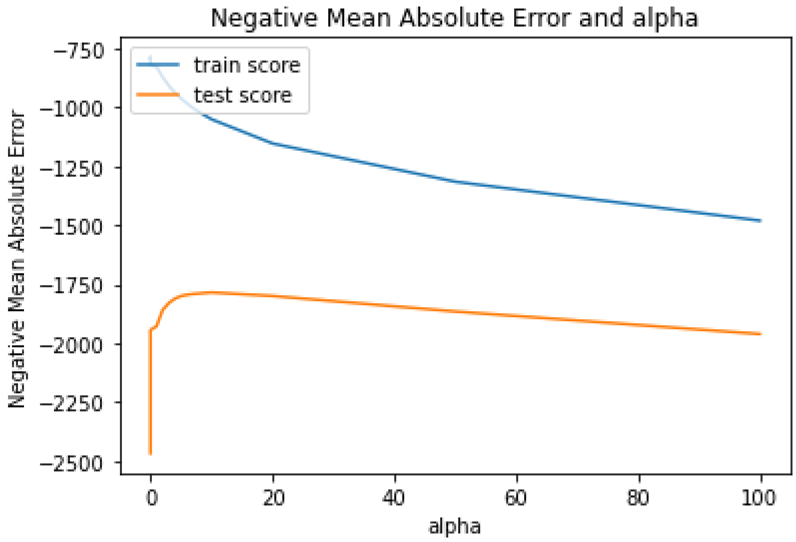
5 rows × 21 columns

Note: The training results depend on the way the train data is splitted in cross validation. Each time you run, the data is splitted randomly and hence you observe minor differences in your answer

plotting mean test and train scores with alpha

In [40]:

```
# change datatype of 'param_alpha' into int
cv_results['param_alpha'] =

# plotting
```



In [41]:

```
# checking best alpha from model_cv
```

Out[41]:

```
{'alpha': 10.0}
```

As you can see that trai and test scores start to become parallel to each other after apha crosses 10. So lets check our ridge model on alpha 10.

In [42]:

```
#sel alpha as 10
alpha =

# Initialise Ridge() with above alpha
ridge =

#fit model

#print ridge coeficients
```

Out[42]:

```
array([ 3.66439600e+02, -3.84733269e+01,  1.48385910e+03, -4.28871390e+02,
        1.32508938e+03,  1.53232524e+03, -1.32353686e+02, -3.43961178e+02,
       -3.85991151e+01,  1.00274451e+03,  4.08530524e+02,  3.06226713e+01,
       -3.86573031e+01,  2.80260386e+02,  2.25689703e+02,  1.15232435e+02,
        3.11172714e+01,  2.20999290e+02, -2.36604555e+02,  4.37146732e+02,
        8.11201095e+01, -4.17761691e+01, -6.36303725e+02, -3.89525755e+02,
       -2.67865922e+02, -2.25694801e+02,  2.17304590e+02,  1.01331104e+03,
       -7.99719800e+01, -2.11299602e+02,  3.80633116e+02,  5.02569921e+01,
       -2.68490049e+01,  1.31276809e+02, -5.38964122e+02, -5.61768987e+02,
       -2.24243763e+02,  3.30958997e+02, -3.41292337e+02,  1.31276809e+02,
        9.03941997e+01, -1.92932138e+02,  2.36604555e+02, -2.70565207e-28,
       -6.35149689e+01, -1.25417650e+02, -2.69360576e-28,  3.66335476e+02,
        1.54082286e+03,  1.04052995e+03, -1.83880719e+02, -4.32144537e+02,
       -3.65503794e+02, -6.64424355e+01,  8.30476815e+02, -2.68159702e+02,
       -2.25623296e-28, -7.10103895e+02, -4.24309992e+02, -3.22453130e+02,
       -3.35256307e+02,  5.94116597e+02, -2.27221245e+02,  2.12510138e+02,
       -4.62583518e+02, -5.92202827e+02, -1.24467623e+02,  1.72803694e+00])
```

Lasso

Cross validation and Hyperparameter tuning: GridSearchCV

In [43]:

```
# Initialise Lasso()
lasso = 

# cross validation and Hyperparameter tuning using lasso
#use same attributes used for Ridge tuning except estimator here would be lasso

#fit model_cv
```

```
Fitting 5 folds for each of 28 candidates, totalling 140 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent work
ers.
[Parallel(n_jobs=1)]: Done 140 out of 140 | elapsed:    5.3s finished
```

Out[43]:

```
GridSearchCV(cv=5, estimator=Lasso(),
             param_grid={'alpha': [0.0001, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3,
                                   0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 2.0,
3.0,
                                   4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 20, 5
0,
                                   100, 500, 1000]},
             return_train_score=True, scoring='neg_mean_absolute_error',
             verbose=1)
```

In [44]:

```
# Save model_cv results into a dataframe
cv_results = 

# cv_results head
```

Out[44]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_alpha	params	split0_t
0	0.041280	0.004229	0.004618	0.004130	0.0001	{'alpha': 0.0001}	-23
1	0.035186	0.002802	0.000826	0.001012	0.001	{'alpha': 0.001}	-23
2	0.038177	0.003033	0.001065	0.001216	0.01	{'alpha': 0.01}	-23
3	0.032661	0.003882	0.005488	0.004900	0.05	{'alpha': 0.05}	-23
4	0.034194	0.000630	0.004873	0.002689	0.1	{'alpha': 0.1}	-24

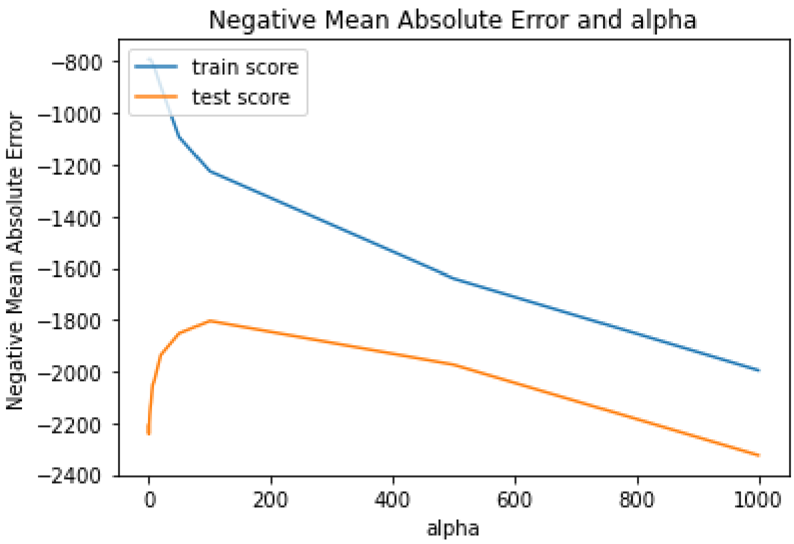
5 rows × 21 columns

Note: The training results depend on the way the train data is splitted in cross validation. Each time you run, the data is splitted randomly and hence you observe minor differences in your answer

plotting mean test and train scores with alpha

In [45]:

```
# change param_alpha datatype to float
cv_results['param_alpha'] =

# plotting
```



In [46]:

```
# Checking best  alpha from model_cv
```

Out[46]:

```
{'alpha': 100}
```

As you can see that trai and test scores start to become parallel to each other after apha crosses 100. So lets check our Lasso model on alpha 100.

In [47]:

```
# Set alpha =100
alpha =

# Define lasso with above alpha
lasso =

# fit lasso
```

Out[47]:

```
Lasso(alpha=100)
```

In [48]:

```
# print lasso coeficients
```

Out[48]:

```
array([   0.        ,   -0.        , 1747.1052243 ,  -82.23183774,
       1780.64173078,  788.28807799,   -0.        ,   -0.        ,
          0.        , 1017.48820119,   84.89633333,    0.        ,
         -0.        ,    0.        ,   -0.        ,   -0.        ,
          0.        ,  246.519852  ,  -73.38572878,  120.56790634,
          0.        ,    0.        , -187.60748943,    0.        ,
        -96.25412649, -134.39227325,  294.27227486, 1218.02281069,
          0.        ,   -0.        ,    0.        ,   -0.        ,
         -0.        ,    0.        ,   -0.        , -202.47407284,
         -0.        ,  197.70712322,   -0.        ,    0.        ,
         -0.        ,   -0.        ,   58.81424436,   -0.        ,
          0.        ,   -0.        ,   -0.        ,  186.35685239,
       1805.30123983, 1210.72936345,    0.        ,   -0.        ,
         -0.        ,   78.54297249,  796.29612837,    0.        ,
         -0.        , -397.80411254,  -58.198149  , -377.78256238,
         -0.        ,  592.06274204, -163.73847377,   95.37139425,
       -198.09298955, -233.82794826,    0.        ,  206.40038676])
```

ElasticNet Regression

Cross validation and Hyperparameter tuning: GridSearchCV

In [49]:

```
from sklearn.linear_model import ElasticNet

# Initialise ElasticNet()
elasticnet =

# cross validation and Hyperparameter tuning using ElasticNet
#use same attributes used for Ridge tuning except estimator here would be ElasticNet


#fit model_cv
```

```
Fitting 5 folds for each of 28 candidates, totalling 140 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent work
ers.
[Parallel(n_jobs=1)]: Done 140 out of 140 | elapsed:    2.8s finished
```

Out[49]:

```
GridSearchCV(cv=5, estimator=ElasticNet(),
             param_grid={'alpha': [0.0001, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3,
                                   0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 2.0,
3.0,
                                   4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 20, 5
0,
                                   100, 500, 1000]},
             return_train_score=True, scoring='neg_mean_absolute_error',
             verbose=1)
```

In [50]:

```
# Save model_cv results into a dataframe
cv_results =

# cv_results head
```

Out[50]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_alpha	params	split0_t
0	0.031865	0.004191	0.002702	0.003027	0.0001	{'alpha': 0.0001}	-23
1	0.034055	0.005297	0.005555	0.003183	0.001	{'alpha': 0.001}	-25
2	0.035958	0.002333	0.001599	0.003198	0.01	{'alpha': 0.01}	-24
3	0.015941	0.002919	0.005264	0.004376	0.05	{'alpha': 0.05}	-24
4	0.012075	0.003206	0.004834	0.003947	0.1	{'alpha': 0.1}	-25

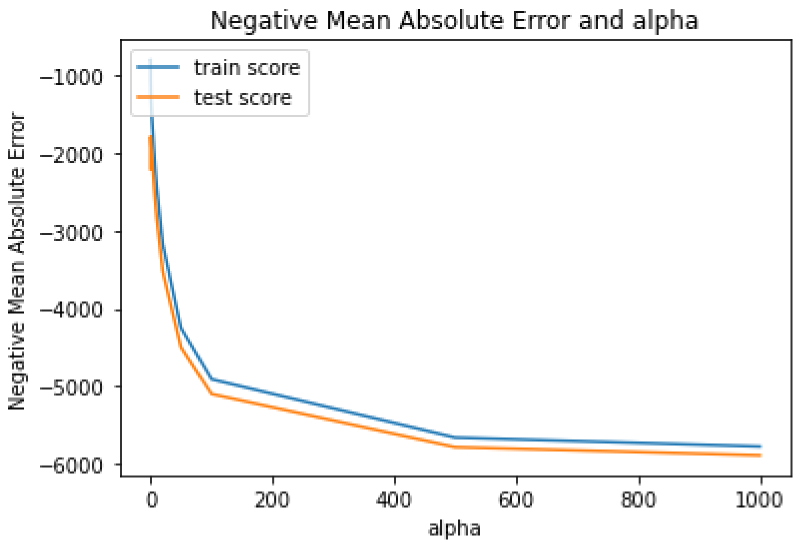
5 rows × 21 columns

Note: The training results depend on the way the train data is splitted in cross validation. Each time you run, the data is splitted randomly and hence you observe minor differences in your answer

plotting mean test and train scores with alpha

In [51]:

```
# change param_alpha datatype to float
cv_results['param_alpha'] =

# plotting
```



In [52]:

```
# Checking best  alpha from model_cv
```

Out[52]:

```
{'alpha': 0.2}
```

As you can see that train and test scores start to become parallel to each other after apha crosses 0.2. So lets check our Elastic model on alpha 0.2.

In [53]:

```
# Set alpha =0.2
alpha =

# Define ElasticNet with above alpha
elasticnet =

# fit elastic net
```

Out[53]:

```
ElasticNet(alpha=0.2)
```

In [54]:

```
# print ElasticNet coeficients
```

Out[54]:

```
array([ 3.52875034e+02,  8.04719724e+01,  1.36197491e+03, -3.45930153e+02,
        1.18273011e+03,  1.35196759e+03, -2.41227489e+01, -2.91628086e+02,
        6.93627991e+00,  9.37481847e+02,  3.32618529e+02, -1.39882008e+01,
       -7.85819335e+01,  2.27493274e+02,  1.72030594e+02,  6.62782224e+01,
       -1.27923967e+01,  2.27565797e+02, -2.14336655e+02,  4.14688503e+02,
        9.90597634e+01,  1.91288777e+01, -5.40401108e+02, -2.83936435e+02,
       -2.27849088e+02, -2.42012667e+02,  2.56159923e+02,  9.31561443e+02,
       -2.60504970e+01, -2.17386627e+02,  3.19978445e+02,  2.58362347e+01,
        1.42903295e-01,  9.85928492e+01, -4.37823888e+02, -5.79072467e+02,
       -1.18311947e+02,  2.97005265e+02, -2.47647701e+02,  9.85847326e+01,
        5.63230424e+01, -1.80422952e+02,  2.14349861e+02, -0.00000000e+00,
       -9.91244238e+00, -1.35054652e+02, -0.00000000e+00,  3.40417804e+02,
        1.47689704e+03,  1.05813935e+03, -1.36588735e+02, -3.58406577e+02,
       -2.90372308e+02, -4.22939256e+01,  8.29953210e+02, -2.34457779e+02,
       -0.00000000e+00, -6.11365043e+02, -4.03350969e+02, -3.18645092e+02,
       -2.65444081e+02,  6.17824181e+02, -2.20222992e+02,  1.92201885e+02,
       -4.46272064e+02, -5.73637157e+02, -1.19797636e+02,  0.00000000e+00])
```

Model evaluation

points= 5

Lets compare all three model result using error term . Here we will check RMSE.

In [55]:

```
# Calculate all 3 predictions
pred_l =
pred_r =
pred_en =
```

In [56]:

```
# import mean_squared_error module
from sklearn.metrics import mean_squared_error

# print RMSE for all 3 techniques
```

```
Lasso RMSE 2494.78187042361
Ridge RMSE 2330.4249922504914
ElasticNet RMSE 2387.770286974708
```

As you can see for our problem statement Ridge as a regularization technique gave us the best result. You can also check for other metrics also, so that you choose the best model.

Generalised Regression using Polynomial regression

points= 18

In this section, we will build a generalised regression model on the electricity consumption dataset. The dataset contains two variables - year and electricity consumption.

Refrence: https://www.analyticsvidhya.com/blog/2020/03/polynomial-regression-python/ (https://www.analyticsvidhya.com/blog/2020/03/polynomial-regression-python/)?

In [57]:

```
#importing libraries
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
```

In [58]:

```
#fetching data
elec_cons =

#printing head
```

Out[58]:

	Year	Consumption
0	1920	57125
1	1921	53656
2	1922	61816
3	1923	72113
4	1924	76651

In [59]:

```
# number of observations: 51
```

Out[59]:

```
(51, 2)
```

In [60]:

```
# checking NA
# there are no missing values in the dataset
```

Out[60]:

```
False
```

In [61]:

```
#Defining length of elec_cons index
size =

# Defining custom index which ranges from 0 to size and step size as  5
index =

#train will not have same index which is is defined above
train =

#test will have same index which is is defined above
test =
```

In [62]:

```
#print train and test length
```

```
40
11
```

In [63]:

```
# converting X to a two dimensional array, as required by the learning algorithm
#Making X_train two dimensional
X_train =

#Defining y_train
y_train =

#Making X_test two dimensional
X_test =

#Defining y_test
y_test =
```

Doing a polynomial regression: Comparing linear, quadratic and cubic fits

Pipeline helps you associate two models or objects to be built sequentially with each other, in this case, the objects are PolynomialFeatures() and LinearRegression()

In [64]:

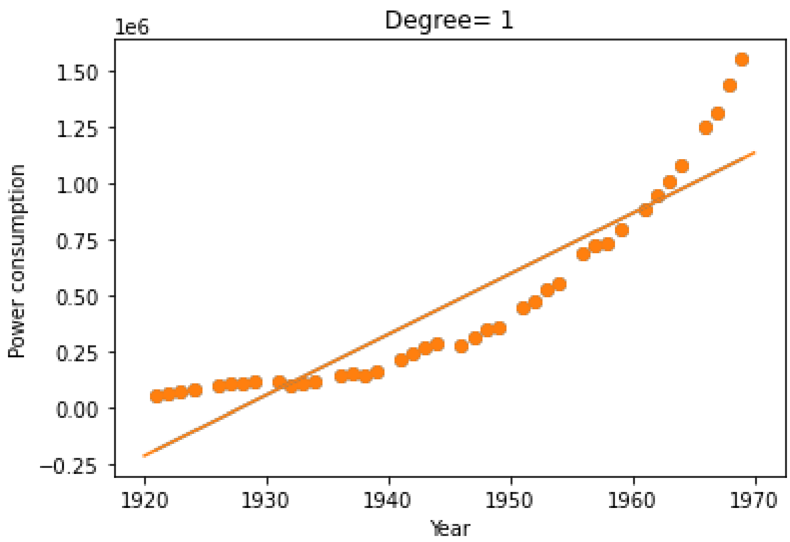
```
# Defining empty list r2_train and r2_test



#Define degrees as list with 1,2 and 3 as elements
degrees =
```

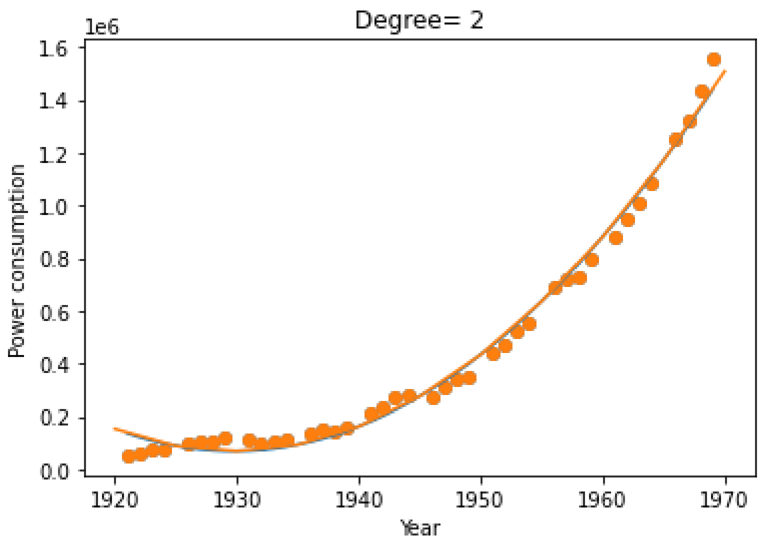
To check how pipeline work: [https://www.youtube.com/watch?v=w9IGkBfOoic (https://www.youtube.com/watch?v=w9IGkBfOoic)](https://www.youtube.com/watch?v=w9IGkBfOoic)

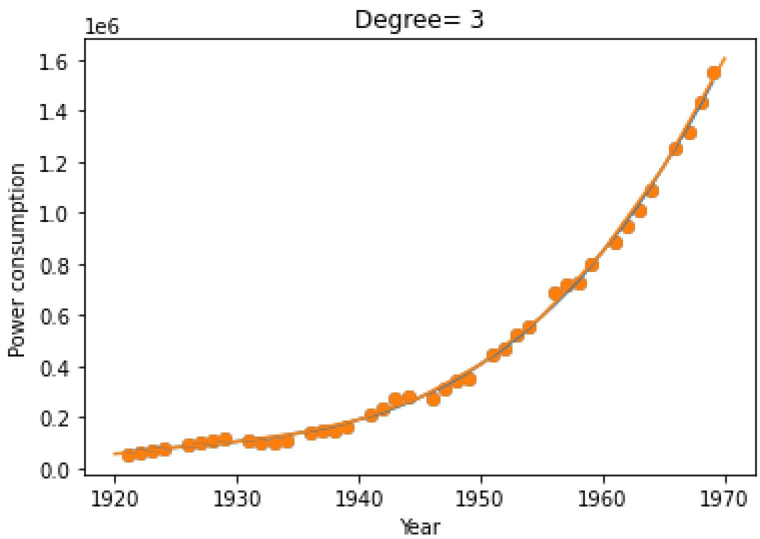
Check its library: [https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html (https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html)](https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html)

In [65]:

```
# Iterating over each degree value

for degree in degrees:
#     initialising pipeline




    #fitting pipeline with train and test




    # test performance



    #appending r2_test with r2_score




    # training performance



    #appending r2_train with r2_score



# plot predictions and actual values against year




    # train data in blue




    # test data


```







In [66]:

```
# respective test r-squared scores of predictions for each degree
```

```
[1, 2, 3]
[0.8423747402176137, 0.9908896744553596, 0.9979789881969624]
[0.816517046382681, 0.9876080502675472, 0.9984897483984051]
```

As you can see that as polynomial degree increases accuracy also increases. But degree should also be decided based on checking condition of of underfitting and overtting.

If you wanna check difference between simple, multiple and polynomial regression then watch: https://www.youtube.com/watch?v=kVVq2JDwiik (https://www.youtube.com/watch?v=kVVq2JDwiik)

Bam! Congratulations You have completed your 11th milestone challenge too!

FeedBack ¶

We hope you've enjoyed this course so far. We're committed to help you use "AI for All" course to its full potential, so that you have a great learning experience. And that's why we need your help in form of a feedback here.

Please fill this feedback form https://zfrmz.in/MtRG5oWXBdesm6rmSM7N (https://zfrmz.in/MtRG5oWXBdesm6rmSM7N)