ML-2_Assignment_Problems_July_3rd_Debajyoti_Podder_C20011

July 12, 2020

0.1 COLAB LINK : https://colab.research.google.com/drive/1mJ84YVeaP8btFC1Ec72hxS-8X-bi8_Kw?usp=sharing

0.2 QUESTION-1

0.2.1 Explain the reasons in your own words

0.2.2 *Look at the codes mentioned below and explain the output. What wrong is happenning with covariance matrix?*

0.3 SOLUTION

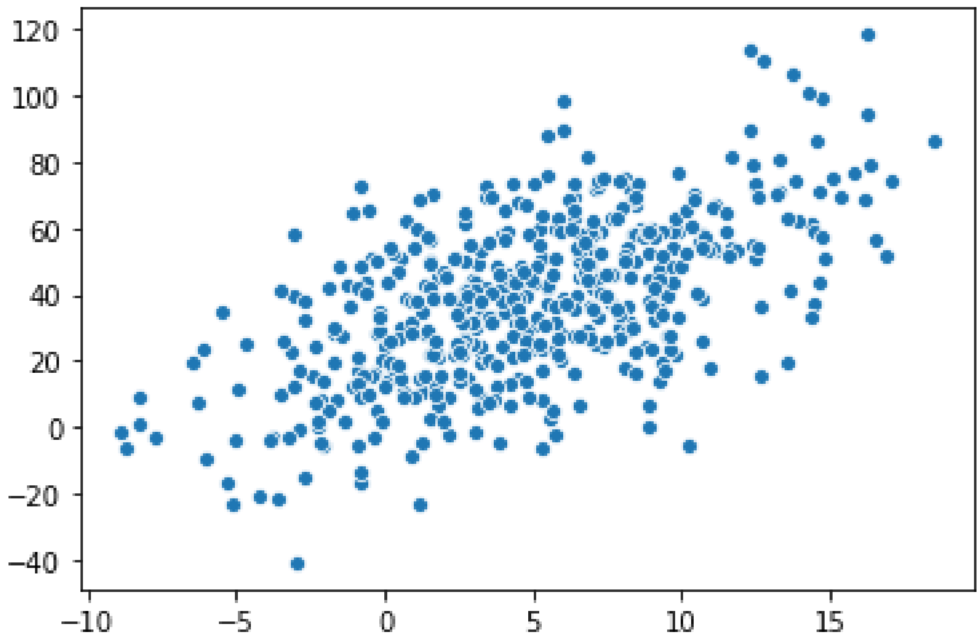
In the question the data is not a standardized data. The standard deviation of the original data for the two variables are 4.996 for x and 24.126 for y. The variable x ranges between -8.86 to 18.48 and whereas the variable y ranges between -40.72 to 118.37. And the PCA is quite sensitive regarding the variances of the initial variables. In PCA we are interested in the components that maximize the variance. That is, if there are large differences between the ranges of initial variables, those variables with larger ranges will dominate over those with small ranges in this question y will dominate over x, which will lead to biased results. If one component (x) varies less than another (y) because of their respective scales, PCA might determine that the direction of maximal variance more closely corresponds with the 'y' axis, if those features are not scaled. As a change in x of one unit can be considered much more important than the change in y of one unit, this is clearly incorrect. So, transforming the data to comparable scales can prevent this problem. Once the standardization is done, all the variables will be transformed to the same scale.

Covariance indicates the direction of the linear relationship between variables. Correlation on the other hand measures both the strength and direction of the linear relationship between two variables. Correlation is a function of the covariance. So, both of them should have same direction. But in the question given, before doing standardization the covariance matrix and correlation matrix is having different direction. But after doing the standardization the direction of the eigen vectors for both covariance matrix and correletaion matrix are in the same direction.

[1]:
```
import pandas as pd
import numpy as np
import matplotlib. pyplot as plt
import seaborn as sns
np. random. seed(0)
x = np. random. normal(5, 5, 500)
```

```
y = 3*x + 20 + np.random.normal(5, 20, 500)
sns.scatterplot(x,y)
plt.show()
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19:
FutureWarning: pandas.util.testing is deprecated. Use the functions in the
public API at pandas.testing instead.
  import pandas.util.testing as tm
```



[2]:
```
arr = np.vstack([x,y]).T
```

[3]:
```
pd.DataFrame(arr)
```

[3]:
```
             0          1
0    13.820262  74.115434
1     7.000786  45.317513
2     9.893690  76.608007
3    16.204466  68.929082
4    14.337790  61.064357
..         ...        ...
495   4.630377  47.148546
496   1.707235  26.153728
497   2.428830  34.170337
498  -0.090209   1.777153
499   4.610726  31.669897

[500 rows x 2 columns]
```

To check the descriptive statistics of the data.

[4]:
```
pd.DataFrame(arr).describe()
```

[4]:

	0	1
count	500.000000	500.000000
mean	4.873228	38.316504
std	4.995782	24.125674
min	-8.862964	-40.721955
25%	1.549126	22.615560
50%	4.759231	39.058298
75%	8.339305	54.415857
max	18.481120	118.369739

[5]:
```
covar_mat = pd. DataFrame(arr) . cov()
correl_mat = pd. DataFrame(arr) . corr()
```

[6]:
```
print(covar_mat)
```

```
           0           1
0  24.957839   70.893700
1  70.893700  582.048165
```

[7]:
```
print(correl_mat)
```

```
        0       1
0  1.0000  0.5882
1  0.5882  1.0000
```

[8]:
```
eigen_val, eigen_vec = np. linalg. eig(covar_mat)
print(eigen_val)
```

```
[ 16.07766503 590.92833906]
```

[9]:
```
eigen_val, eigen_vec = np. linalg. eig(correl_mat)
print(eigen_val)
```

```
[1.58819955 0.41180045]
```

[9]:
```

```

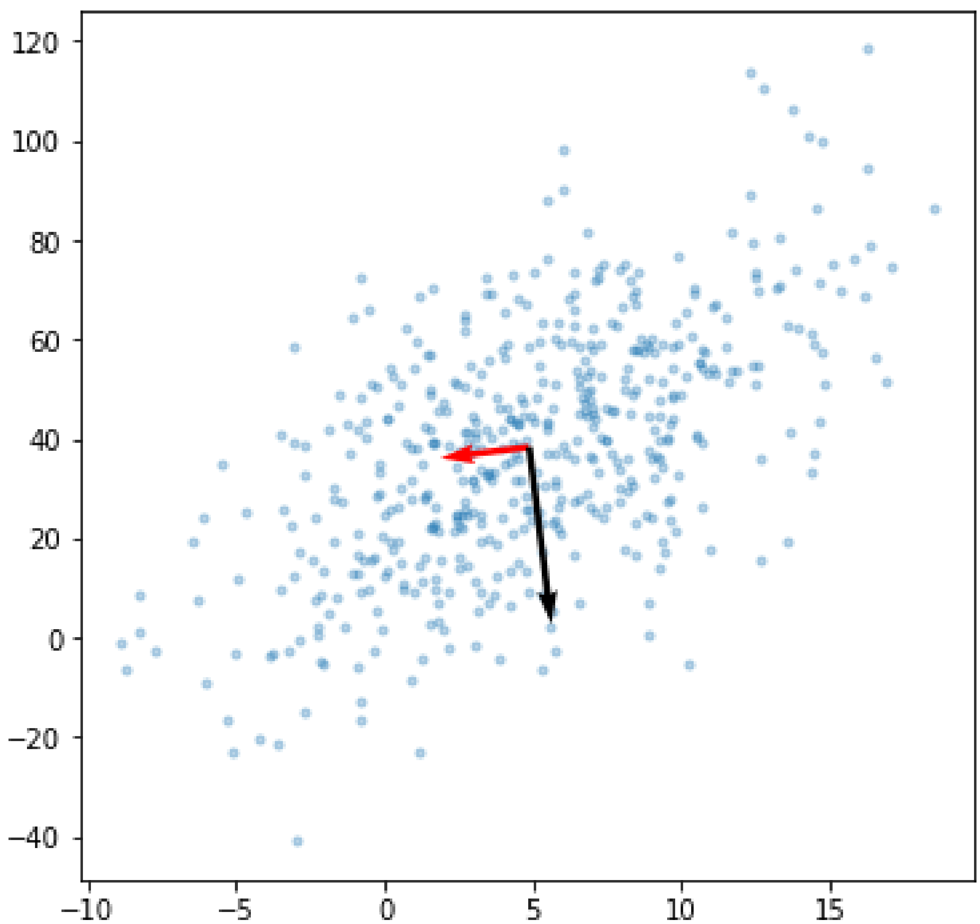
In the eigenvalues of the covariance matrix are of totally different scale. So, while plotting the eigenvectors the scale of the two axis has been changed from (eigen_val **5) to (10,5) by defining a numpy array.***

[10]:
```
def plot_pca_2D(arr, symm_mat, cov_mat=True):
    n_comp = 2
    mean_x, mean_y = (np. mean(arr[:, 0]), np. mean(arr[:, 1]))
```

```
    print(mean_x, mean_y)
    plt. figure(figsize=(6, 6))
    if cov_mat:
        eigen_val, eigen_vec = np. linalg. eig(symm_mat)
        plt. scatter(arr[:, 0],arr[:, 1], alpha=0.3, s=10)
        plt. quiver(np. array([mean_x,mean_x]), np. array([mean_y,mean_y]),␣
↪eigen_vec[:, 0], eigen_vec[:, 1], color=['r' , 'black' ], scale=(np.
↪array([10,5])))
        plt.show()
    else:
        cor = symm_mat
        eigen_val, eigen_vec = np. linalg. eig(symm_mat)
        plt. scatter(arr[:, 0],arr[:, 1], alpha=0.3, s=10)
        plt. quiver(np. array([mean_x,mean_x]),np.␣
↪array([mean_y,mean_y]),eigen_vec[:, 0], eigen_vec[:, 1], color=['r' ,␣
↪'black' ], scale=eigen_val*5)
        plt. show()
```

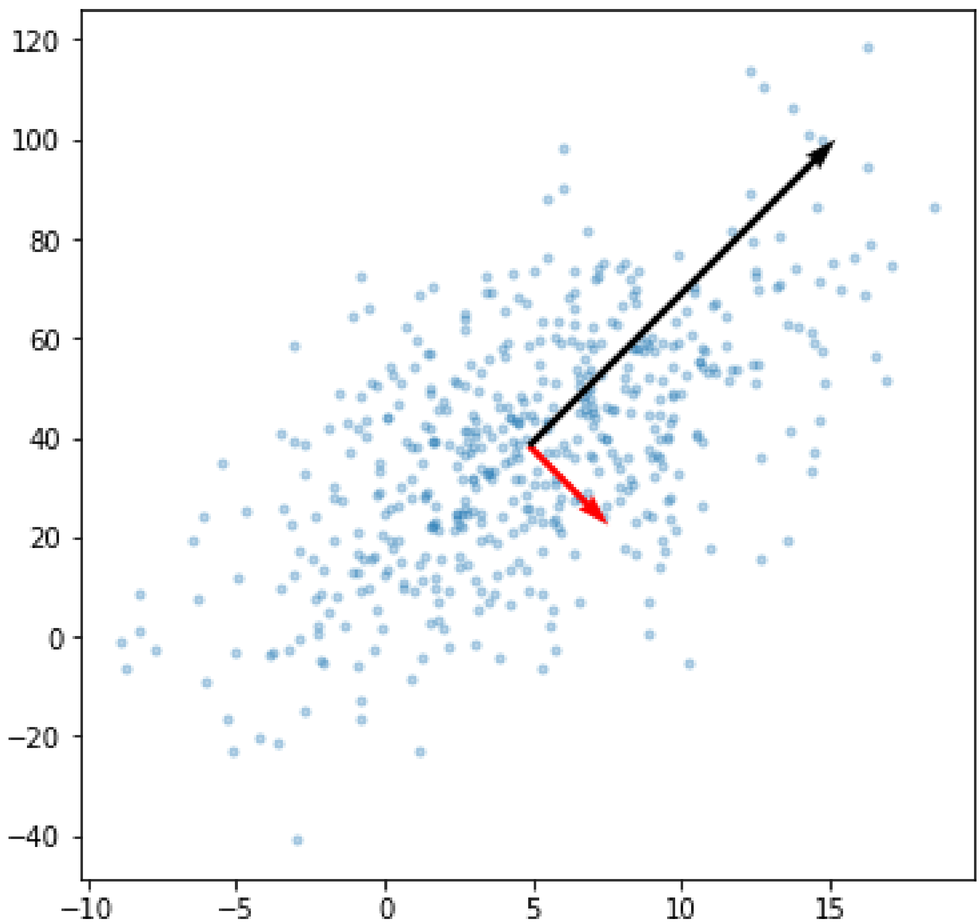
[11]:
```
plot_pca_2D(arr, symm_mat=covar_mat)
```

```
4.873227803337831 38.31650389705435
```



[12]:
```
plot_pca_2D(arr, symm_mat=correl_mat, cov_mat=False)
```

```
4.873227803337831 38.31650389705435
```



Standardization of data.

[13]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
arr2 = sc.fit_transform(arr)
```

[14]:
```
pd.DataFrame(arr2)
```

[14]:

	0	1
0	1.792711	1.485338
1	0.426297	0.290480
2	1.005947	1.588758
3	2.270433	1.270150

```
4    1.896408  0.943834
..        ...       ...
495 -0.048660  0.366451
496 -0.634368 -0.504647
497 -0.489782 -0.172029
498 -0.994521 -1.516059
499 -0.052597 -0.275775

[500 rows x 2 columns]
```

To check the descriptive statistics of the standardized data.

[15]:
```
pd.DataFrame(arr2).describe()
```

[15]:
```
                  0             1
count  5.000000e+02  5.000000e+02
mean  -1.970646e-18  9.436896e-18
std    1.001002e+00  1.001002e+00
min   -2.752311e+00 -3.279395e+00
25%   -6.660480e-01 -6.514499e-01
50%   -2.284141e-02  3.077786e-02
75%    6.944956e-01  6.679804e-01
max    2.726604e+00  3.321499e+00
```

[16]:
```
covar_mat_2 = pd. DataFrame(arr2) . cov()
correl_mat_2 = pd. DataFrame(arr2) . corr()
```

[17]:
```
print(covar_mat_2)
```

```
          0         1
0  1.002004  0.589378
1  0.589378  1.002004
```

[18]:
```
print(correl_mat_2)
```

```
        0       1
0  1.0000  0.5882
1  0.5882  1.0000
```

[19]:
```
eigen_val_2, eigen_vec_2 = np. linalg. eig(covar_mat_2)
print(eigen_val_2)
```

```
[1.59138232 0.4126257 ]
```

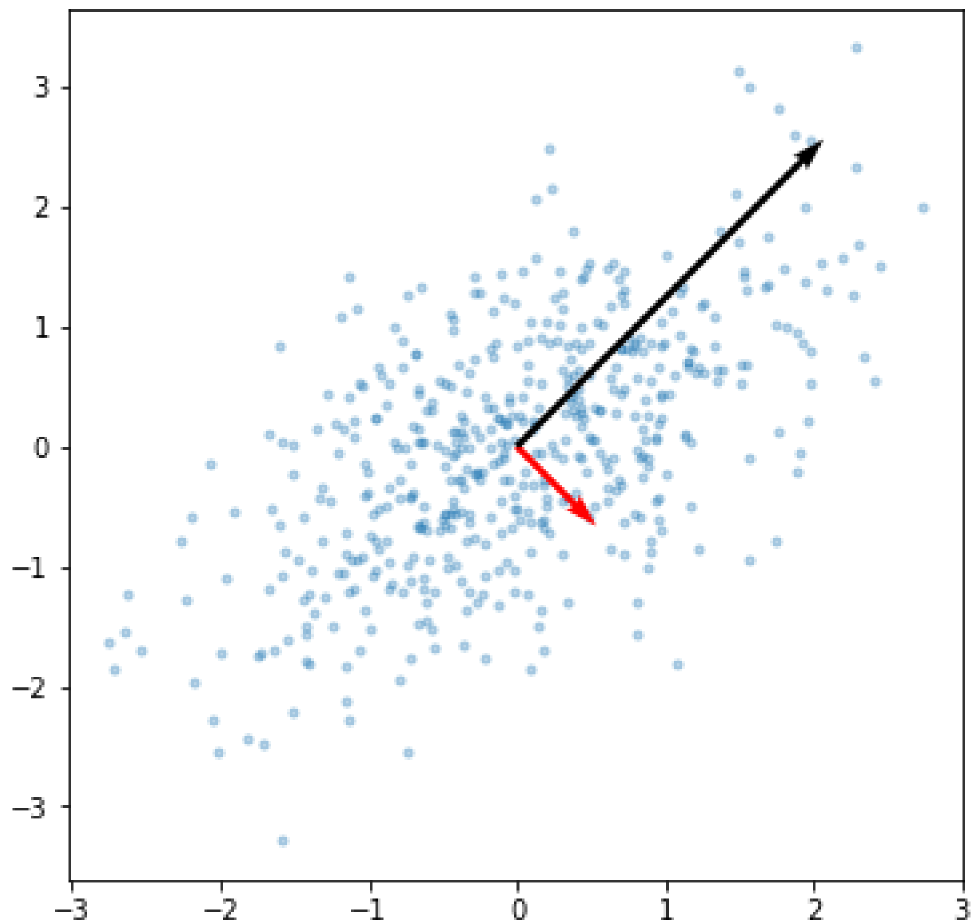
[20]:
```
eigen_val_2, eigen_vec_2 = np. linalg. eig(correl_mat_2)
print(eigen_val_2)
```

```
[1.58819955 0.41180045]
```

[21]:
```
def plot_pca_2D(arr2, symm_mat, cov_mat=True):
    n_comp = 2
    mean_x, mean_y = (np. mean(arr2[:, 0]), np. mean(arr2[:, 1]))
    print(mean_x, mean_y)
    plt. figure(figsize=(6, 6))
    if cov_mat:
        eigen_val_2, eigen_vec_2 = np. linalg. eig(symm_mat)
        plt. scatter(arr2[:, 0],arr2[:, 1], alpha=0.3, s=10)
        plt. quiver(np. array([mean_x,mean_x]), np. array([mean_y,mean_y]),␣
 ↪eigen_vec_2[:, 0], eigen_vec_2[:, 1], color=['r' , 'black' ],␣
 ↪scale=eigen_val*5)
        plt.show()
    else:
        cor = symm_mat
        eigen_val_2, eigen_vec_2 = np. linalg. eig(symm_mat)
        plt. scatter(arr2[:, 0],arr2[:, 1], alpha=0.3, s=10)
        plt. quiver(np. array([mean_x,mean_x]),np.␣
 ↪array([mean_y,mean_y]),eigen_vec_2[:, 0], eigen_vec_2[:, 1], color=['r' ,␣
 ↪'black' ], scale=eigen_val*5)
        plt. show()
```

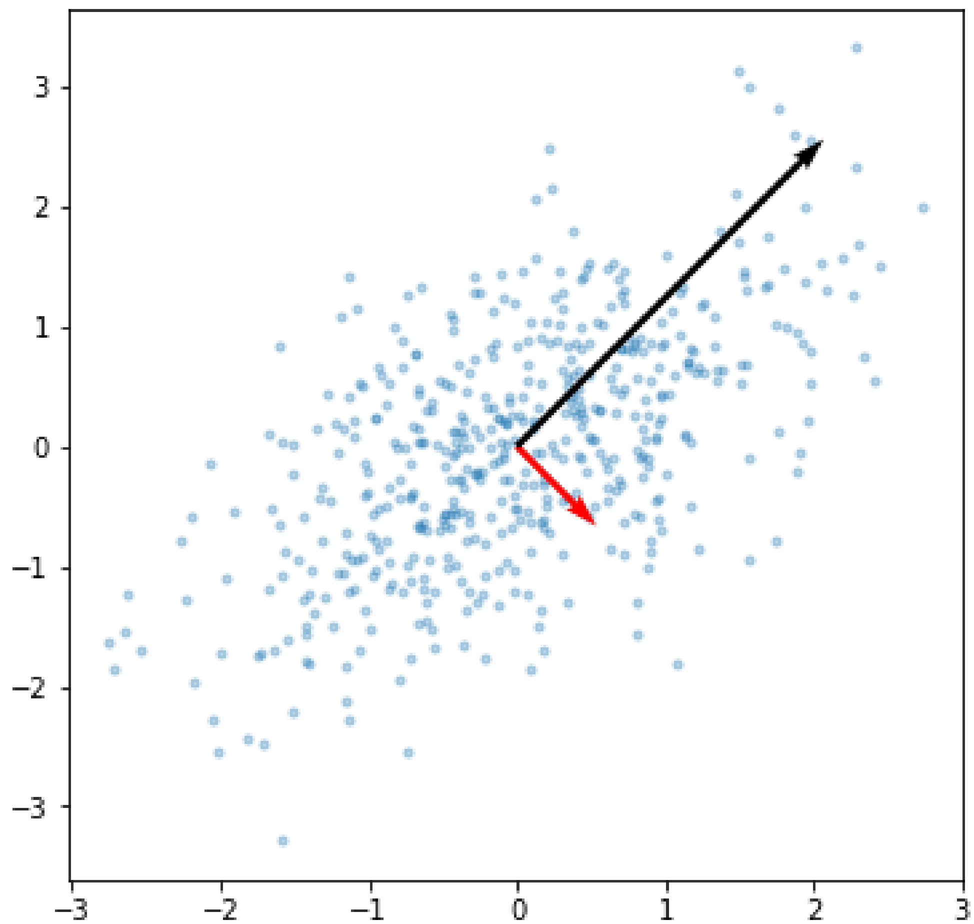
[22]:
```
plot_pca_2D(arr2, symm_mat=covar_mat_2)
```

```
-7.105427357601002e-18 7.105427357601002e-18
```



[23]:
```
plot_pca_2D(arr2, symm_mat=correl_mat_2, cov_mat=False)
```

```
-7.105427357601002e-18 7.105427357601002e-18
```



0.4 QUESTION-2

0.4.1 Solve the problem with MLE

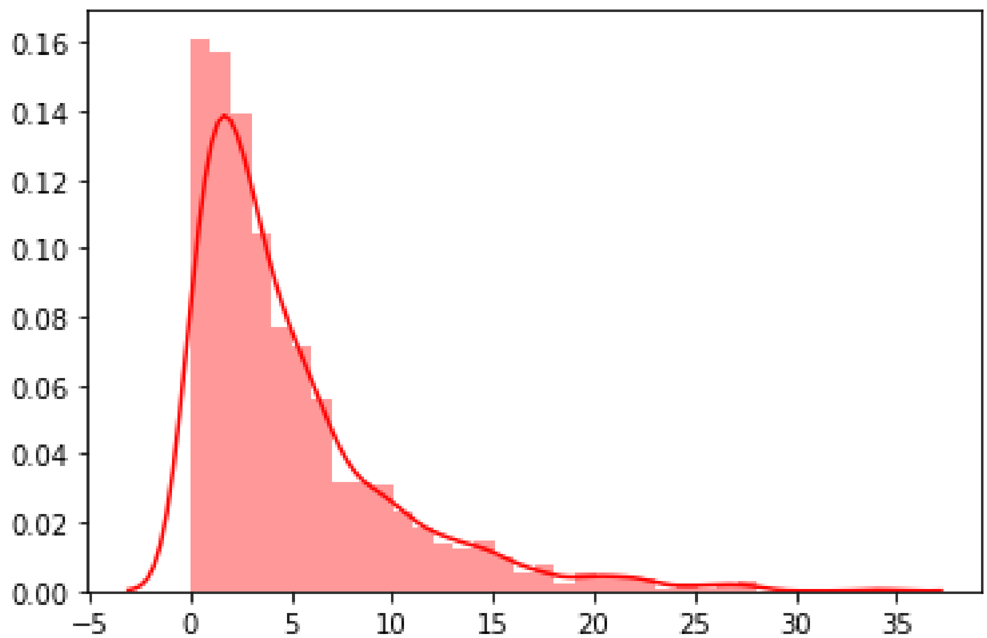
0.4.2 *You are given a dataset which was generated using exponential distribution. Find out the model parameters to extract the pattern using MLE. Write a simple python program to solve the problem*

0.5 SOLUTION

[24]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

[25]:
```
np.random.seed(123)
exp_data = np.random.exponential(5, size=1000)
sns.distplot(exp_data, color='r')
```

[25]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa90e0c6dd8>



[26]:
```
len_exp = len(exp_data)
print("The length n is :", len_exp)
```

```
The length n is : 1000
```

[27]:
```
sum_exp = np.sum(exp_data)
print("The sum of the data points :", sum_exp)
```

```
The sum of the data points : 4991.529882068665
```

[28]:
```
lambda_MLE = len_exp/sum_exp
print("The Maximum Likelihood Estimator of lambda is :", lambda_MLE)
```

```
The Maximum Likelihood Estimator of lambda is : 0.20033937963435872
```

The same can be done using the mean function.

[29]:
```
mean_exp = np.mean(exp_data)
print("The mean of the data points :", mean_exp)
```

```
The mean of the data points : 4.991529882068665
```

[30]:
```
lambda_MLE_mean = 1/mean_exp
print("The Maximum Likelihood Estimator of lambda computed from mean is :",␣
 ↪lambda_MLE_mean)
```

```
The Maximum Likelihood Estimator of lambda computed from mean is :
0.2003393796343587
```

0.6 QUESTION-3

0.6.1 Solve using EM algorithm

0.6.2 *There are two biased coins available. You can choose any one of the coins and toss it for 20 times. The outcomes are to be recorded. The problem is, the coins cannot be diffrentiated from each other and hence you cannot know which coin was chosen and tossed for 20 times in a particular experiment. Assume that the experiment was run 50 times and you have got the full information about the outcome. Try to estimate the probability of success of each coin (head = success). The outcomes are generated using the codes below. Use your own python codes from scratch.*

0.7 SOLUTION

[31]:
```
import pandas as pd
import numpy as np
import matplotlib. pyplot as plt
import seaborn as sns
```

[32]:
```
np. random. seed(123)
r1 = np. random. binomial(1, 0.7,(25, 20)) # p for one of the coins is 0.7
r2 = np. random. binomial(1, 0.4,(25, 20)) # p for the other coins is 0.4
outcome = np. vstack([r1,r2])
np. random. shuffle(outcome)
outcome[: 5,:]
```

[32]:
```
array([[1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0],
       [1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1],
       [1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1],
       [1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1]])
```

Converting the 1 to 'H' for Heads outcom and 0 to 'T' for Tails outcome.

[33]:
```
outcome_H_T = np.where(outcome == 1,'H', 'T')
outcome_H_T[: 5,:]
```

[33]:
```
array([['H', 'T', 'H', 'T', 'H', 'H', 'H', 'T', 'H', 'H', 'H', 'H', 'T',
        'H', 'H', 'H', 'H', 'H', 'H', 'T'],
       ['H', 'T', 'H', 'T', 'H', 'T', 'T', 'T', 'H', 'T', 'T', 'T', 'T',
        'H', 'T', 'H', 'H', 'H', 'H', 'H'],
       ['H', 'H', 'H', 'T', 'T', 'H', 'H', 'H', 'H', 'H', 'T', 'H', 'H',
        'T', 'H', 'H', 'H', 'H', 'T', 'H'],
```

```
['H', 'T', 'H', 'H', 'H', 'T', 'H', 'T', 'T', 'H', 'T', 'H', 'T',
 'T', 'T', 'H', 'H', 'H', 'H', 'H'],
['H', 'H', 'H', 'H', 'T', 'T', 'H', 'H', 'H', 'H', 'T', 'T', 'H',
 'T', 'H', 'H', 'H', 'H', 'H', 'H']], dtype='<U1')
```

EM Algorithm Implementation

[34]:
```
# Initial assumption for theta value of Coin-A and Coin-B
theta_coinA = 0.7
theta_coinB = 0.6
theta_coin_A_B = {'A': theta_coinA, 'B': theta_coinB}


# User defined function to simulate the EM Algorithm
def EM_cal(theta_prev, outcome_H_T):
  prob_per_set = []

  # Calculation of Expectation_E-Step
  for toss_outcome in outcome_H_T:
    F = np.math.factorial
    P = np.power
    heads_count = np.count_nonzero(toss_outcome=='H')
    prob_coinA = (F(len(toss_outcome)) / (F(heads_count) * (F(len(toss_outcome)␣
↪- heads_count)))) * (P(theta_prev['A'], heads_count))*(P((1 -␣
↪theta_prev['A']), (len(toss_outcome) - heads_count)))
    prob_coinB = (F(len(toss_outcome)) / (F(heads_count) * (F(len(toss_outcome)␣
↪- heads_count)))) * (P(theta_prev['B'], heads_count))*(P((1 -␣
↪theta_prev['B']), (len(toss_outcome) - heads_count)))

    prob_coinA_exp = prob_coinA / (prob_coinA + prob_coinB)
    prob_coinB_exp = prob_coinB / (prob_coinA + prob_coinB)
    prob_per_set.append({'Coin_A': prob_coinA_exp, 'Coin_B': prob_coinB_exp,␣
↪'Heads_Count': heads_count, 'Toss_Count': len(toss_outcome)})


  # Calculation of Maximisation_M-Step
  coin_toss_new_set = []
  for row in prob_per_set:
    toss_count = row['Toss_Count']
    heads_count = row['Heads_Count']
    heads_coinA = row['Coin_A']*heads_count
    tails_coinA = row['Coin_A']*(toss_count-heads_count)
    heads_coinB = row['Coin_B']*heads_count
    tails_coinB = row['Coin_B']*(toss_count-heads_count)
    coin_toss_new_set.append([heads_coinA, tails_coinA, heads_coinB,␣
↪tails_coinB])
```

```
  # DataFrame to view Head and Tail combination of each set of experiment
  head_tail_combi = pd.DataFrame(coin_toss_new_set, columns=['CoinA_Heads',␣
↪'CoinA_Tails', 'CoinB_Heads', 'CoinB_Tails'])

  # The values for theta that maximize the expected number of heads/tails
  prob_coinA_per_set = head_tail_combi['CoinA_Heads'].sum()/
↪(head_tail_combi['CoinA_Heads'].sum()+head_tail_combi['CoinA_Tails'].sum())
  prob_coinB_per_set = head_tail_combi['CoinB_Heads'].sum()/
↪(head_tail_combi['CoinB_Heads'].sum()+head_tail_combi['CoinB_Tails'].sum())
  updated_theta = {'A': prob_coinA_per_set, 'B': prob_coinB_per_set}

  display(head_tail_combi.head())
  return updated_theta
```

Calculating the Euclidean Distance

[35]:
```
import math
iter_count = 0
euc_distance = 1
min_dist = 10**-5

while (euc_distance>min_dist) and (iter_count<10000):
  theta_next = EM_cal(theta_coin_A_B, outcome_H_T)
  l1 = list(zip(theta_next.values(), theta_coin_A_B.values()))
  euc_distance = math.sqrt((l1[0][0]-l1[0][1])**2 + (l1[1][0]-l1[1][1])**2)
  theta_coin_A_B = theta_next
  iter_count+=1
```

	CoinA_Heads	CoinA_Tails	CoinB_Heads	CoinB_Tails
0	10.583192	3.527731	4.416808	1.472269
1	2.082817	2.082817	7.917183	7.917183
2	10.583192	3.527731	4.416808	1.472269
3	4.667627	3.111751	7.332373	4.888249
4	10.583192	3.527731	4.416808	1.472269

	CoinA_Heads	CoinA_Tails	CoinB_Heads	CoinB_Tails
0	14.409555	4.803185	0.590445	0.196815
1	1.561788	1.561788	8.438212	8.438212
2	14.409555	4.803185	0.590445	0.196815
3	6.792576	4.528384	5.207424	3.471616
4	14.409555	4.803185	0.590445	0.196815

	CoinA_Heads	CoinA_Tails	CoinB_Heads	CoinB_Tails
0	14.876156	4.958719	0.123844	0.041281
1	1.415183	1.415183	8.584817	8.584817

2	14.876156	4.958719	0.123844	0.041281
3	8.365780	5.577187	3.634220	2.422813
4	14.876156	4.958719	0.123844	0.041281

	CoinA_Heads	CoinA_Tails	CoinB_Heads	CoinB_Tails
0	14.920354	4.973451	0.079646	0.026549
1	1.587509	1.587509	8.412491	8.412491
2	14.920354	4.973451	0.079646	0.026549
3	8.986503	5.991002	3.013497	2.008998
4	14.920354	4.973451	0.079646	0.026549

	CoinA_Heads	CoinA_Tails	CoinB_Heads	CoinB_Tails
0	14.929008	4.976336	0.070992	0.023664
1	1.705977	1.705977	8.294023	8.294023
2	14.929008	4.976336	0.070992	0.023664
3	9.202091	6.134727	2.797909	1.865273
4	14.929008	4.976336	0.070992	0.023664

	CoinA_Heads	CoinA_Tails	CoinB_Heads	CoinB_Tails
0	14.931793	4.977264	0.068207	0.022736
1	1.762784	1.762784	8.237216	8.237216
2	14.931793	4.977264	0.068207	0.022736
3	9.286705	6.191137	2.713295	1.808863
4	14.931793	4.977264	0.068207	0.022736

	CoinA_Heads	CoinA_Tails	CoinB_Heads	CoinB_Tails
0	14.932887	4.977629	0.067113	0.022371
1	1.788080	1.788080	8.211920	8.211920
2	14.932887	4.977629	0.067113	0.022371
3	9.322009	6.214673	2.677991	1.785327
4	14.932887	4.977629	0.067113	0.022371

	CoinA_Heads	CoinA_Tails	CoinB_Heads	CoinB_Tails
0	14.933345	4.977782	0.066655	0.022218
1	1.799121	1.799121	8.200879	8.200879
2	14.933345	4.977782	0.066655	0.022218
3	9.337067	6.224711	2.662933	1.775289
4	14.933345	4.977782	0.066655	0.022218

	CoinA_Heads	CoinA_Tails	CoinB_Heads	CoinB_Tails
0	14.933541	4.977847	0.066459	0.022153
1	1.803909	1.803909	8.196091	8.196091
2	14.933541	4.977847	0.066459	0.022153

3	9.343540	6.229027	2.656460	1.770973
4	14.933541	4.977847	0.066459	0.022153

	CoinA_Heads	CoinA_Tails	CoinB_Heads	CoinB_Tails
0	14.933625	4.977875	0.066375	0.022125
1	1.805982	1.805982	8.194018	8.194018
2	14.933625	4.977875	0.066375	0.022125
3	9.346332	6.230888	2.653668	1.769112
4	14.933625	4.977875	0.066375	0.022125

	CoinA_Heads	CoinA_Tails	CoinB_Heads	CoinB_Tails
0	14.933661	4.977887	0.066339	0.022113
1	1.806878	1.806878	8.193122	8.193122
2	14.933661	4.977887	0.066339	0.022113
3	9.347536	6.231691	2.652464	1.768309
4	14.933661	4.977887	0.066339	0.022113

	CoinA_Heads	CoinA_Tails	CoinB_Heads	CoinB_Tails
0	14.933677	4.977892	0.066323	0.022108
1	1.807265	1.807265	8.192735	8.192735
2	14.933677	4.977892	0.066323	0.022108
3	9.348057	6.232038	2.651943	1.767962
4	14.933677	4.977892	0.066323	0.022108

[36]:
```
print("Probability of success for Coin A and Coin B using the EM Algorithm :
↪",theta_coin_A_B)
```

```
Probability of success for Coin A and Coin B using the EM Algorithm : {'A':
0.7151422887712894, 'B': 0.38576227428575244}
```

0.8 QUESTION-4

0.8.1 Visualize the impact of parameter tuning

0.8.2 *Take any dataset of you choice (at least 10k data points) and take any one of the following models:*

0.8.3 *SVM*

0.8.4 *GBM*

0.8.5 *Xgboost*

0.8.6 *Random Forest*

0.8.7 *and vary one hyperparameter at a time to estimate training error and validation error. Vary 5 such hyperparameters to plot 5x2 lineplots for training error and validation error. Based on the line plots, can you infer anything?*

0.9 SOLUTION : USING RANDOM FOREST REGRESSOR

Dataset used : https://www.kaggle.com/swathiachath/kc-housesales-data

[37]:
```
import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
```

Importing the Dataset

[38]:
```
from google.colab import files
files.upload()
```

```
Output hidden; open in https://colab.research.google.com to view.
```

[39]:
```
kc_house_price = pd.read_csv("kc_house_data.csv")
kc_house_price.head()
```

[39]:
```
           id        date     price  ...     long  sqft_living15  sqft_lot15
0  7129300520  10/13/2014  221900.0  ... -122.257           1340        5650
1  6414100192   12/9/2014  538000.0  ... -122.319           1690        7639
2  5631500400   2/25/2015  180000.0  ... -122.233           2720        8062
3  2487200875   12/9/2014  604000.0  ... -122.393           1360        5000
4  1954400510   2/18/2015  510000.0  ... -122.045           1800        7503

[5 rows x 21 columns]
```

Data Wrangling

[40]:
```
kc_house_price.shape
```

[40]: (21597, 21)

[41]:
```
kc_house_price.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             21597 non-null  int64
 1   date           21597 non-null  object
 2   price          21597 non-null  float64
 3   bedrooms       21597 non-null  int64
 4   bathrooms      21597 non-null  float64
 5   sqft_living    21597 non-null  int64
 6   sqft_lot       21597 non-null  int64
 7   floors         21597 non-null  float64
 8   waterfront     21597 non-null  int64
 9   view           21597 non-null  int64
 10  condition      21597 non-null  int64
 11  grade          21597 non-null  int64
 12  sqft_above     21597 non-null  int64
 13  sqft_basement  21597 non-null  int64
 14  yr_built       21597 non-null  int64
 15  yr_renovated   21597 non-null  int64
 16  zipcode        21597 non-null  int64
 17  lat            21597 non-null  float64
 18  long           21597 non-null  float64
 19  sqft_living15  21597 non-null  int64
 20  sqft_lot15     21597 non-null  int64
dtypes: float64(5), int64(15), object(1)
memory usage: 3.5+ MB
```

[42]:
```
kc_house_price.isnull().sum()
```

[42]:
```
id                0
date              0
price             0
bedrooms          0
bathrooms         0
sqft_living       0
sqft_lot          0
floors            0
waterfront        0
view              0
condition         0
grade             0
sqft_above        0
```

```
sqft_basement    0
yr_built         0
yr_renovated     0
zipcode          0
lat              0
long             0
sqft_living15    0
sqft_lot15       0
dtype: int64
```

[43]:
```
kc_house_price.describe()
```

[43]:
```
                 id         price  ...  sqft_living15     sqft_lot15
count  2.159700e+04  2.159700e+04  ...   21597.000000   21597.000000
mean   4.580474e+09  5.402966e+05  ...    1986.620318   12758.283512
std    2.876736e+09  3.673681e+05  ...     685.230472   27274.441950
min    1.000102e+06  7.800000e+04  ...     399.000000     651.000000
25%    2.123049e+09  3.220000e+05  ...    1490.000000    5100.000000
50%    3.904930e+09  4.500000e+05  ...    1840.000000    7620.000000
75%    7.308900e+09  6.450000e+05  ...    2360.000000   10083.000000
max    9.900000e+09  7.700000e+06  ...    6210.000000  871200.000000

[8 rows x 20 columns]
```

[44]:
```
kc_house_price.dtypes
```

[44]:
```
id                 int64
date              object
price            float64
bedrooms           int64
bathrooms        float64
sqft_living        int64
sqft_lot           int64
floors           float64
waterfront         int64
view               int64
condition          int64
grade              int64
sqft_above         int64
sqft_basement      int64
yr_built           int64
yr_renovated       int64
zipcode            int64
lat              float64
long             float64
sqft_living15      int64
sqft_lot15         int64
dtype: object
```

[45]:
```
kc_house_price.nunique()
```

[45]:
```
id               21420
date               372
price             3622
bedrooms            12
bathrooms           29
sqft_living       1034
sqft_lot          9776
floors               6
waterfront           2
view                 5
condition            5
grade               11
sqft_above         942
sqft_basement      306
yr_built           116
yr_renovated        70
zipcode             70
lat               5033
long               751
sqft_living15      777
sqft_lot15        8682
dtype: int64
```

Finding the age of each house. And dropping the unnecessary columns

[46]:
```
kc_house_price['date']= pd.to_datetime(kc_house_price['date'])
kc_house_price['house_age'] = (pd.DatetimeIndex(kc_house_price['date']).year) -␣
↪kc_house_price.yr_built
kc_house_price.drop(['id', 'date', 'yr_built', 'lat',␣
↪'long'],axis=1,inplace=True)
```

Coded the building which has been renovated with 1 and the rest with 0

[47]:
```
kc_house_price['yr_renovated'] = kc_house_price['yr_renovated'].apply(lambda x :
↪ 1 if x>0 else 0)
```

[48]:
```
kc_house_price.rename(columns={'yr_renovated': 'renovation_status'},␣
↪inplace=True)
```

[49]:
```
kc_house_price.head()
```

[49]:

	price	bedrooms	bathrooms	...	sqft_living15	sqft_lot15	house_age
0	221900.0	3	1.00	...	1340	5650	59
1	538000.0	3	2.25	...	1690	7639	63
2	180000.0	2	1.00	...	2720	8062	82
3	604000.0	4	3.00	...	1360	5000	49
4	510000.0	3	2.00	...	1800	7503	28

```
[5 rows x 17 columns]
```

Dividing the dataset into dependent and independent variables.

[50]:
```
X = kc_house_price.iloc[:, 1:].values
y = kc_house_price.iloc[:, 0].values
```

[51]:
```
print(X)
```

```
[[3.000e+00 1.000e+00 1.180e+03 ... 1.340e+03 5.650e+03 5.900e+01]
 [3.000e+00 2.250e+00 2.570e+03 ... 1.690e+03 7.639e+03 6.300e+01]
 [2.000e+00 1.000e+00 7.700e+02 ... 2.720e+03 8.062e+03 8.200e+01]
 ...
 [2.000e+00 7.500e-01 1.020e+03 ... 1.020e+03 2.007e+03 5.000e+00]
 [3.000e+00 2.500e+00 1.600e+03 ... 1.410e+03 1.287e+03 1.100e+01]
 [2.000e+00 7.500e-01 1.020e+03 ... 1.020e+03 1.357e+03 6.000e+00]]
```

[52]:
```
print(y)
```

```
[221900. 538000. 180000. ... 402101. 400000. 325000.]
```

Standardizing the dataset

[53]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)
```

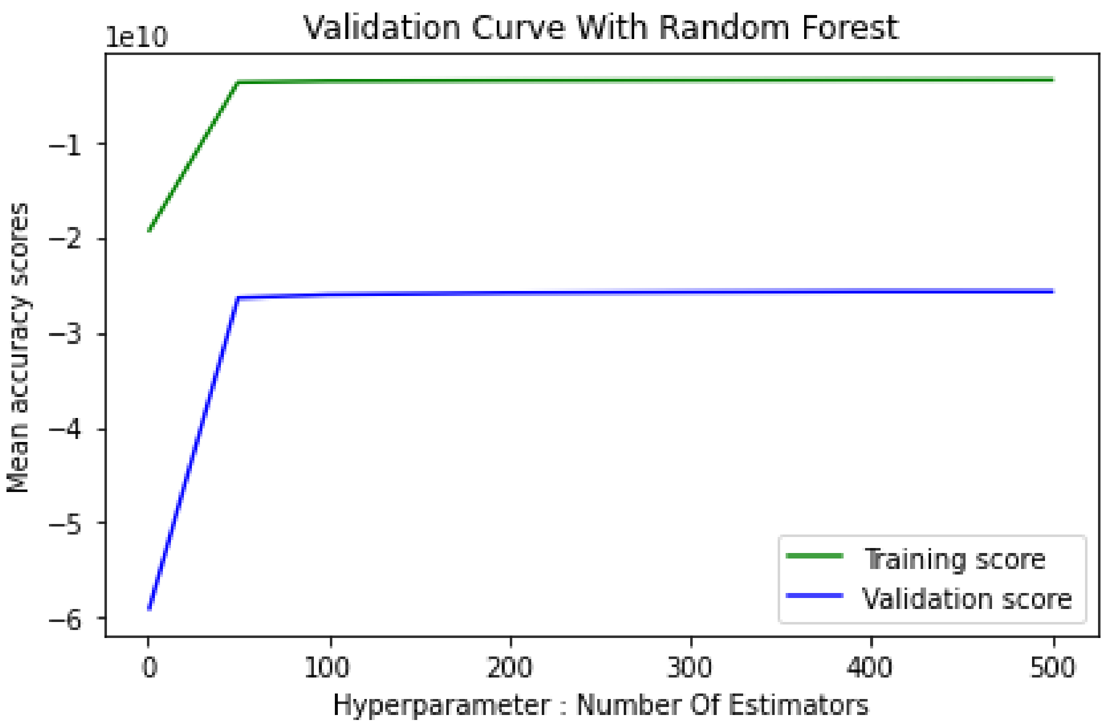
Implementing the Random Forest Regressor model on the whole dataset to estimate the training & validation score for different hyperparameters

Hyperparameter : "n_estimator"

[54]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import validation_curve


train_scores, test_scores =␣
 ↪validation_curve(RandomForestRegressor(random_state=1),
                                  X, y, param_name="n_estimators",␣
 ↪param_range=[1,50,100,200,300,400,500],
                                  cv=3, scoring="neg_mean_squared_error",␣
 ↪n_jobs=-1)

# To calculate the mean for training and test set scores
train_mean = np.mean(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
```

```
# Plotting the mean accuracy of both training and test set scores
plt.plot([1,50,100,200,300,400,500], train_mean, label="Training score",␣
 ↪color="green")
plt.plot([1,50,100,200,300,400,500], test_mean, label="Validation score",␣
 ↪color="blue")

# Plot title
plt.title("Validation Curve With Random Forest")
plt.xlabel("Hyperparameter : Number Of Estimators")
plt.ylabel("Mean accuracy scores")
plt.tight_layout()
plt.legend(loc="best")
plt.show()
```

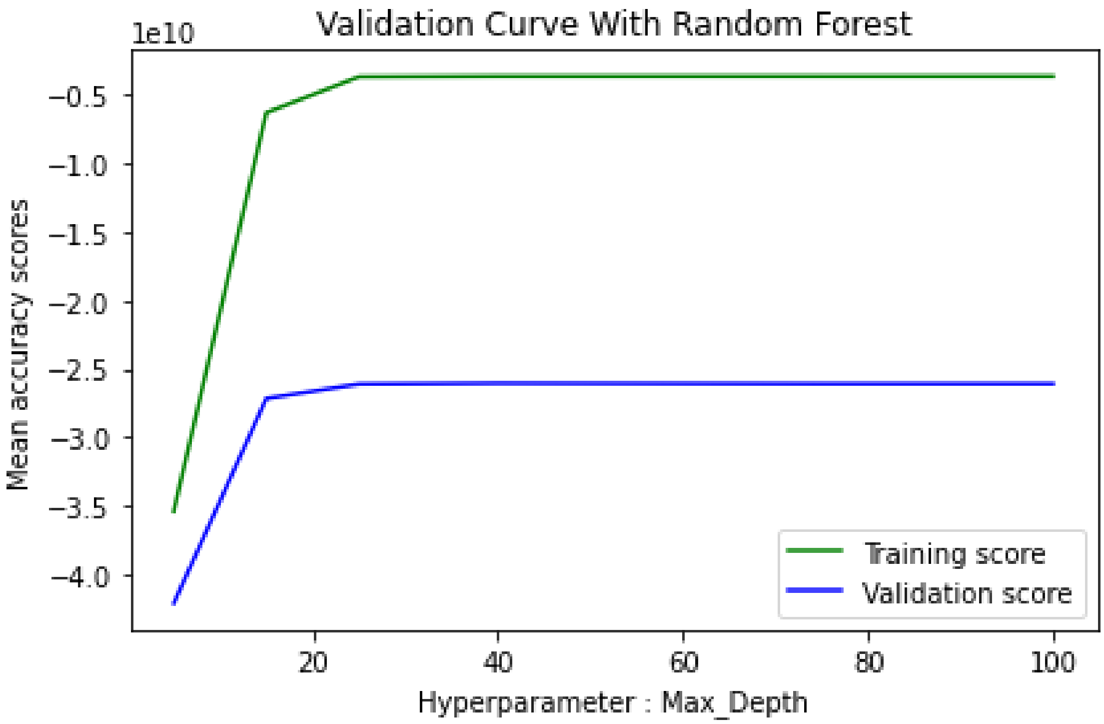
```
/usr/local/lib/python3.6/dist-
packages/joblib/externals/loky/process_executor.py:691: UserWarning: A worker
stopped while some jobs were given to the executor. This can be caused by a too
short worker timeout or by a memory leak.
  "timeout or by a memory leak.", UserWarning
```



Hyperparameter : "max_depth"

[55]:
```python
train_scores, test_scores =␣
↪validation_curve(RandomForestRegressor(random_state=1),
                                X, y, param_name="max_depth", param_range=[5,␣
↪15, 25, 40, 75, 100],
                                cv=3, scoring="neg_mean_squared_error",␣
↪n_jobs=-1)

# To calculate the mean for training and test set scores
train_mean = np.mean(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)

# Plotting the mean accuracy of both training and test set scores
plt.plot([5, 15, 25, 40, 75, 100], train_mean, label="Training score",␣
↪color="green")
plt.plot([5, 15, 25, 40, 75, 100], test_mean, label="Validation score",␣
↪color="blue")

# Plot title
plt.title("Validation Curve With Random Forest")
plt.xlabel("Hyperparameter : Max_Depth")
plt.ylabel("Mean accuracy scores")
plt.tight_layout()
plt.legend(loc="best")
plt.show()
```

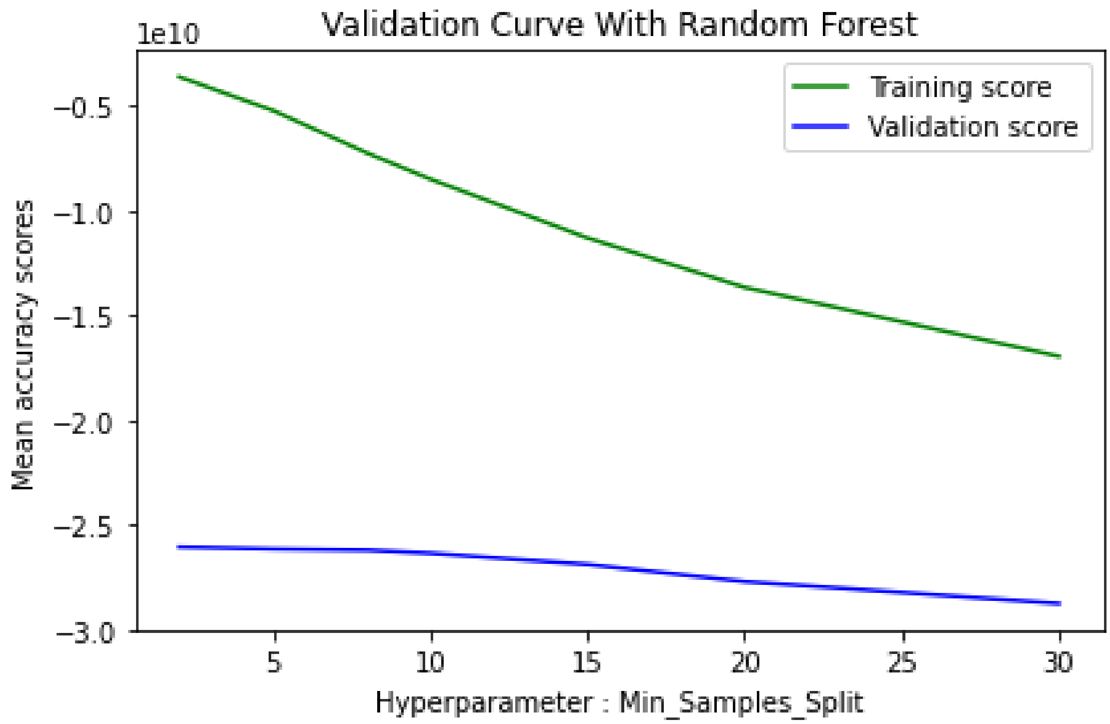
```
/usr/local/lib/python3.6/dist-
packages/joblib/externals/loky/process_executor.py:691: UserWarning: A worker
stopped while some jobs were given to the executor. This can be caused by a too
short worker timeout or by a memory leak.
  "timeout or by a memory leak.", UserWarning
```



Hyperparameter : "min_samples_split"

[56]:
```
train_scores, test_scores =␣
↪validation_curve(RandomForestRegressor(random_state=1),
                                  X, y, param_name="min_samples_split",␣
↪param_range=[2, 5, 8, 10, 15, 20, 30],
                                  cv=3, scoring="neg_mean_squared_error",␣
↪n_jobs=-1)

# To calculate the mean for training and test set scores
train_mean = np.mean(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)

# Plotting the mean accuracy of both training and test set scores
plt.plot([2, 5, 8, 10, 15, 20, 30], train_mean, label="Training score",␣
↪color="green")
plt.plot([2, 5, 8, 10, 15, 20, 30], test_mean, label="Validation score",␣
↪color="blue")

# Plot title
plt.title("Validation Curve With Random Forest")
plt.xlabel("Hyperparameter : Min_Samples_Split")
plt.ylabel("Mean accuracy scores")
```

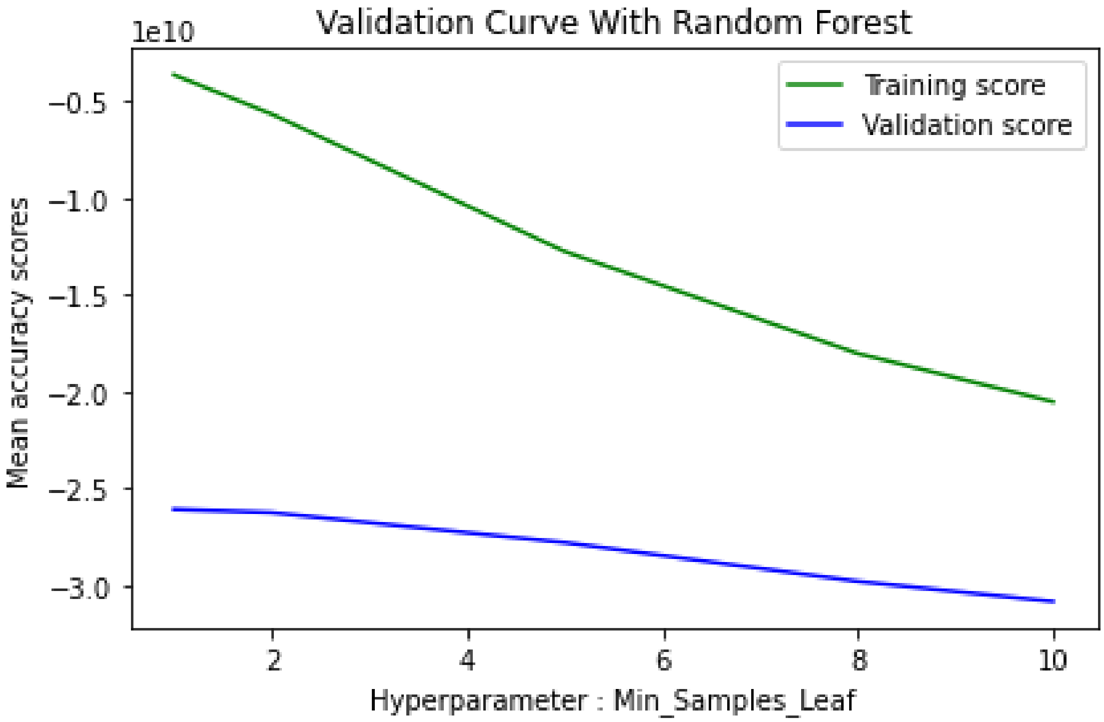
```
plt.tight_layout()
plt.legend(loc="best")
plt.show()
```



Hyperparameter : "min_samples_leaf"

[57]:
```
train_scores, test_scores =␣
 ↪validation_curve(RandomForestRegressor(random_state=1),
                                  X, y, param_name="min_samples_leaf",␣
 ↪param_range=[1, 2, 5, 8, 10],
                                  cv=3, scoring="neg_mean_squared_error",␣
 ↪n_jobs=-1)

# To calculate the mean for training and test set scores
train_mean = np.mean(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)

# Plotting the mean accuracy of both training and test set scores
plt.plot([1, 2, 5, 8, 10], train_mean, label="Training score", color="green")
plt.plot([1, 2, 5, 8, 10], test_mean, label="Validation score", color="blue")

# Plot title
plt.title("Validation Curve With Random Forest")
```

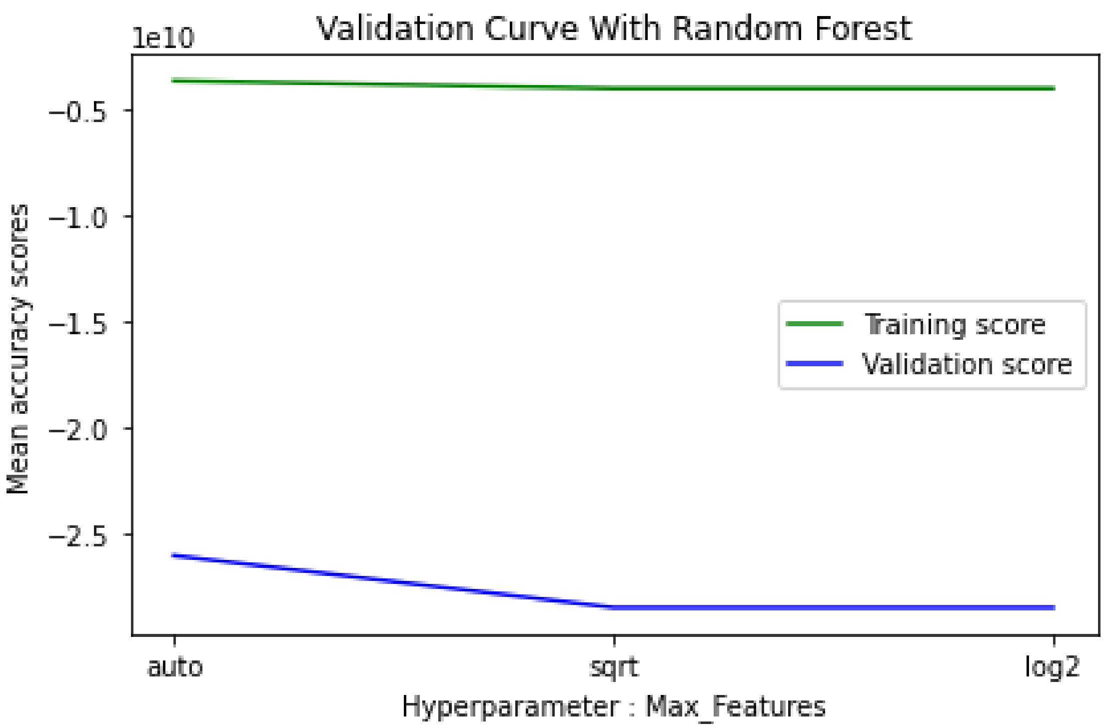
```
plt.xlabel("Hyperparameter : Min_Samples_Leaf")
plt.ylabel("Mean accuracy scores")
plt.tight_layout()
plt.legend(loc="best")
plt.show()
```



Hyperparameter : "max_features"

[58]:
```
train_scores, test_scores =␣
↪validation_curve(RandomForestRegressor(random_state=1),
                                  X, y, param_name="max_features",␣
↪param_range=['auto', 'sqrt', 'log2'],
                                  cv=3, scoring="neg_mean_squared_error",␣
↪n_jobs=-1)

# To calculate the mean for training and test set scores
train_mean = np.mean(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)

# Plotting the mean accuracy of both training and test set scores
plt.plot(['auto', 'sqrt', 'log2'], train_mean, label="Training score",␣
↪color="green")
```

```
plt.plot(['auto', 'sqrt', 'log2'], test_mean, label="Validation score",␣
↪color="blue")

# Plot title
plt.title("Validation Curve With Random Forest")
plt.xlabel("Hyperparameter : Max_Features")
plt.ylabel("Mean accuracy scores")
plt.tight_layout()
plt.legend(loc="best")
plt.show()
```



Inference

1. While we changed the hyperparameter "n_estimator", the mean accuracy score of both train and validation got increased drastically from 1 to 100. After "n_estimator"=100 there was almost no variation observed in the score, making it almost constant.

2. For variation in the "max_depth" hyperparameter, the train and validation score improved for "max_depth"=5 to 20 and from 20 onwards there was almost no further improvement in the score for both the dataset.

3. In the case of "min_sample_split", as we increased the value, the score started decreasing with a steady slope for both train and validation. But the decrease in score was more steep for train than validation.

4. Similar trend to that of "min_sample_split" can be observed for "min_samples_leaf".

5. When we changed the value of the hyperparameter 'max_features" we got best accuracy for both train and validation with "max_features" = "auto". For all other values the accuracy score got degraded for validation, but for train negligible change can be observed.

In general we can say that both "n_estimator" and "max_depth" have similar effect on the accuracy score. As we increase the value of the hyperparameters the accuracy gets improved upto a certain level and then become constant. But by increasing the value of the hyperparameters "min_sample_split" and "min_samples_leaf", we can observe the worse effect on the accuracy score.

0.9.1 FOR PDF CONVERSION

[ ]:
```
!apt-get install texlive texlive-xetex texlive-latex-extra pandoc
!pip install pypandoc
```

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

[ ]:
```
!cp drive/My\ Drive/Assignments/ML\ Assignments/
↪ML-2_Assignment_Problems_July_3rd_Debajyoti_Podder_C20011.ipynb ./
```

[ ]:
```
!jupyter nbconvert --to PDF␣
↪"ML-2_Assignment_Problems_July_3rd_Debajyoti_Podder_C20011.ipynb"
```

[ ]:
```

```