ASSIGNMENT – 2

FOUNDATIONS OF DATA SCIENCE

CS F320

MUFADDAL JIRUWALA 2020A7PS1720H
SAI HEMANTH ANANTHOJU 2020A7PS0116H

INDEX

TITLE PAGE NO.

TITLE	PAGE NO.
2-A	3
2-B	13
2-C	19

2-A:

CORRELATION COEFFICIENTS:

We are given a dataset with 26 features and one value to be predicted, ‘Appliances’.

Important Observation

We observe that the values in columns ‘rv_1’ and ‘rv_2’ are the same in the given dataset. Since they both have the same values, and hence the same correlation, we have decided to drop the column ‘rv_2’

This will not affect the models we generate in any way; hence we are sticking with this approach.

We then split the data into training and testing data (We are splitting it 80-20).

Now, we have assumed that the mean of each of the features is zero (Assumption as discussed in class). Hence, we subtract each value in each column with the mean of the entire column, so that the mean becomes zero.

We split the training data again for cross-validation (We split the data 80-20)

	Correlation Coefficients
T1	0.046837
RH_1	0.073984
T2	0.108182
RH_2	-0.066643
T3	0.076273
RH_3	0.028431
T4	0.036171
RH_4	0.009868
T5	0.013707
RH_5	0.020321
T6	0.113492
RH_6	-0.077423
T7	0.017269
RH_7	-0.059178
T8	0.036829
RH_8	-0.088884
T9	0.006491
RH_9	-0.055466
T_out	0.097117
Press_mm_hg	-0.024974
RH_out	-0.151474
Windspeed	0.081625
Visibility	0.001100
Tdewpoint	0.011789
rv1	-0.007425

As you can see, these are the correlation coefficients for each feature.

We select the variables which have the maximum correlation with the target variable, to train the model.

We then define 2 Calculate Error functions, that calculates the root mean square errors, one for training data and one for testing data.

```python
def calculateError(train_data, w_degree):
    train_pred = np.dot(train_data, w_degree)
    d = train_pred - y_data
    error = np.dot(d.T,d)

    e_rms = (error/len(train_data))**0.5

    return e_rms
```

```python
abs_corr = np.abs(np.asarray(coeff_mat))
training_errors = []
w_train = []

for i in range(1,26):
    indices = np.argpartition(abs_corr, -i)[-i:]

    x_data_by_corr = x_data[:,indices]

    w_req = np.dot(np.dot(np.linalg.inv(np.dot(x_data_by_corr.T,x_data_by_corr)),x_data_by_corr.T),y_data)
    w_train.append(w_req)
    training_errors.append(calculateError(x_data_by_corr, w_req))

# training_errors = []
# for sublist in model_error:
#     for item in sublist:
#         training_errors.append(item)

training_errors_df = pd.DataFrame(training_errors, columns=["Training Errors"])
training_errors_df.index = np.arange(1, len(training_errors_df) + 1)
training_errors_df
```

The above code shows how we are calculating out testing errors.

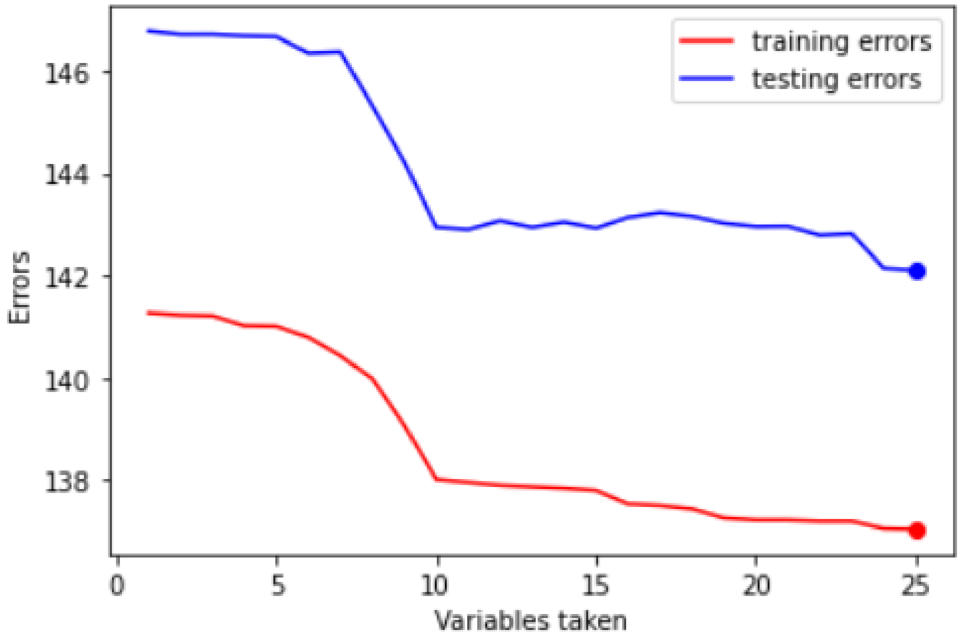
```python
def calculateErrorTest(test_data, w_train_degree):
    test_pred = np.dot(test_data, w_train_degree)
    d = test_pred - y_data_test
    error = np.dot(d.T,d)

    e_rms = (error/len(test_data))**0.5

    return e_rms
```

We then calculate the testing errors and they are as follows:

	Training Errors
1	141.267756
2	141.225722
3	141.211533
4	141.021656
5	141.010095
6	140.791863
7	140.432910
8	139.984289
9	139.078704
10	138.016240
11	137.957349
12	137.901970
13	137.871573
14	137.842931
15	137.799305
16	137.536378
17	137.503357
18	137.435891
19	137.259438
20	137.218742
21	137.217832
22	137.195900
23	137.195319
24	137.055186
25	137.039557

	Testing Errors
1	146.779739
2	146.715229
3	146.716650
4	146.687403
5	146.674157
6	146.351152
7	146.377692
8	145.319430
9	144.230708
10	142.945030
11	142.902206
12	143.074763
13	142.942980
14	143.048494
15	142.927633
16	143.138669
17	143.234385
18	143.159999
19	143.026022
20	142.959352
21	142.964310
22	142.790704
23	142.819398
24	142.145689
25	142.104912

Here, we see the training and testing errors for the respective variables taken:



PRINCIPAL COMPONENT ANALYSIS:

Firstly, we calculate the covariance matrix. (Since we have dropped column ‘rv_2’, we get a 25x25 matrix)

Covariance matrix:

	0	1	2	3	4	5	6	7	8	9	...	15	16	17	18	
0	2.652877	1.189107	3.036333	0.098906	2.959406	-0.041914	2.964032	0.849641	2.728947	0.004337	...	0.168492	2.813492	0.602615	5.999020	-1.8
1	1.189107	15.881908	2.487423	12.961859	2.134339	11.083019	1.001074	15.450509	1.578352	10.900475	...	15.563559	1.009757	12.778293	7.478736	-8.0
2	3.036333	2.487423	4.975146	-1.465119	3.343448	0.951128	3.538099	2.354824	3.043867	0.813792	...	1.002047	3.089227	1.528339	9.522340	-2.1
3	0.098906	12.961859	-1.465119	16.766850	1.178887	9.181920	-0.309382	13.007216	0.855637	9.264716	...	14.720939	0.494069	11.601102	0.772053	-7.2
4	2.959406	2.134339	3.343448	1.178887	4.115378	0.045346	3.579657	1.242411	3.399197	-0.817698	...	0.717865	3.720978	1.230596	7.620187	-2.8

We then calculate the Eigen values for each of the variables:

So, now instead of the original variables, we will now train our models using factors computed through the principal components, i.e., the corresponding eigen values and eigen vectors.

The eigen values are given in the DataFrame below:

We take the eigen vectors with the highest eigen values for the calculation of principal components for collecting as maximum variance as possible of the original variables.

	Eigenvalues
0	1181.442358
1	206.300684
2	156.928928
3	132.984437
4	99.687071
5	65.515338
6	47.774244
7	11.495474
8	7.820884
9	7.500422
10	4.001138
11	3.414821
12	2.691100
13	2.523233
14	1.289907
15	0.920207
16	0.715207
17	0.627371
18	0.486482
19	0.402613
20	0.286092
21	0.215427
22	0.133493
23	0.072326
24	0.094750

We then calculate the eigen vectors, as shown below:

Eigenvectors	0	1	2	3	4	5	6	7	8	9	...	15	16	17	
Var1	0.027945	-0.002013	0.059228	0.026600	0.011849	-0.003631	-0.002351	0.115378	0.235771	0.144302	...	-0.076335	-0.051158	-0.190591	-0.
Var2	-0.031490	-0.002367	0.248340	0.082767	-0.058234	-0.082668	0.059439	-0.247583	0.246381	-0.348806	...	0.212031	0.387019	0.042469	-0.
Var3	0.038321	-0.001414	0.083613	0.035926	-0.051495	-0.027038	0.018002	0.181760	0.134921	0.211897	...	0.057744	0.055991	-0.128076	-0.
Var4	-0.053793	-0.003848	0.211165	0.069443	0.085516	-0.039985	0.034449	-0.348574	0.315943	-0.486380	...	0.009031	-0.270509	-0.109933	0.
Var5	0.035861	-0.003388	0.085600	0.033899	0.038344	-0.014443	-0.005877	0.076257	0.248943	0.105611	...	-0.011866	-0.206108	-0.368655	-0.
Var6	-0.049236	0.000164	0.168379	0.055110	-0.096412	-0.060985	0.061175	-0.216066	0.082361	0.003252	...	-0.720625	-0.375838	0.284667	0.
Var7	0.040844	-0.002504	0.063581	0.025983	0.028870	-0.001542	0.007303	0.115096	0.289516	0.161365	...	-0.111525	-0.106944	-0.303524	0.
Var8	-0.051816	-0.002400	0.267462	0.092023	-0.098999	-0.097549	0.102239	-0.147652	-0.023758	-0.106318	...	0.268855	0.042944	-0.082313	0.
Var9	0.032988	-0.002714	0.073766	0.032335	0.033194	0.014125	-0.008101	0.083372	0.271690	0.117591	...	0.101784	-0.113646	-0.150818	0.
Var10	-0.068552	0.018671	0.289942	0.045199	-0.296297	0.900345	-0.034636	0.066200	-0.027819	-0.037653	...	0.000836	-0.000737	-0.000080	-0.
Var11	0.120834	-0.002287	0.260515	0.106108	-0.144471	-0.176820	0.075319	0.452429	-0.191205	-0.188090	...	0.342331	-0.552676	0.206372	-0.
Var12	-0.903090	0.012679	-0.106713	-0.088964	-0.301931	-0.142097	-0.007440	0.188410	0.117643	-0.011658	...	0.023813	-0.014267	0.001719	-0.
Var13	0.045337	-0.002145	0.054318	0.022192	0.043825	-0.003130	-0.009522	0.080012	0.317130	0.127494	...	0.094858	0.036171	0.381468	0.
Var14	-0.056405	-0.004591	0.324834	0.112474	-0.066947	-0.112176	0.109337	-0.138508	-0.123348	0.283255	...	0.045150	0.003764	0.036829	-0.
Var15	0.036446	-0.002184	0.041437	0.024074	0.053691	0.017467	-0.038021	0.105337	0.316871	0.113411	...	0.025126	0.178007	0.609096	-0.
Var16	-0.078261	-0.002144	0.289206	0.117418	-0.050191	-0.078481	0.115134	-0.230458	-0.234985	0.465268	...	0.087334	-0.012096	0.031743	0.
Var17	0.041412	-0.002597	0.068188	0.028944	0.058373	-0.009571	-0.014344	0.091716	0.258654	0.079585	...	0.152573	-0.063683	0.103123	0.
Var18	-0.049669	0.000709	0.232440	0.081857	-0.065306	-0.109501	0.123953	-0.234770	-0.001517	0.249983	...	0.031611	0.109921	-0.070855	0.
Var19	0.101418	-0.003359	0.238877	0.097121	-0.148672	-0.158677	0.075774	0.374405	-0.045968	-0.129856	...	-0.282023	0.261442	-0.078271	0.
Var20	0.013143	0.006462	-0.228960	-0.050519	0.026984	0.120838	0.956893	0.038693	0.042945	-0.037646	...	0.008273	-0.012772	0.022185	-0.
Var21	-0.346593	-0.002849	0.218330	0.118570	0.839910	0.160807	0.019144	0.122095	-0.163118	-0.027866	...	-0.009424	-0.037219	0.028831	0.
Var22	-0.003961	0.000573	0.027699	0.002838	-0.106059	-0.054891	-0.050704	-0.096053	-0.335685	-0.174541	...	-0.005222	-0.050052	0.043584	0.
Var23	-0.039157	-0.023200	-0.339369	0.935702	-0.063217	0.038244	-0.034007	-0.015465	0.007124	-0.010237	...	0.001310	-0.001566	-0.003432	0.
Var24	0.026560	-0.003734	0.278599	0.115952	0.032867	-0.116424	0.076544	0.335269	-0.077000	-0.180288	...	-0.300371	0.366062	-0.091120	0.
Var25	-0.011394	-0.999386	0.002179	-0.025605	-0.009679	0.017028	0.004412	0.001595	-0.003441	-0.000905	...	-0.000561	-0.000101	0.001165	0.

25 rows × 25 columns

We then calculate the training and testing errors produced principal component analysis:

:

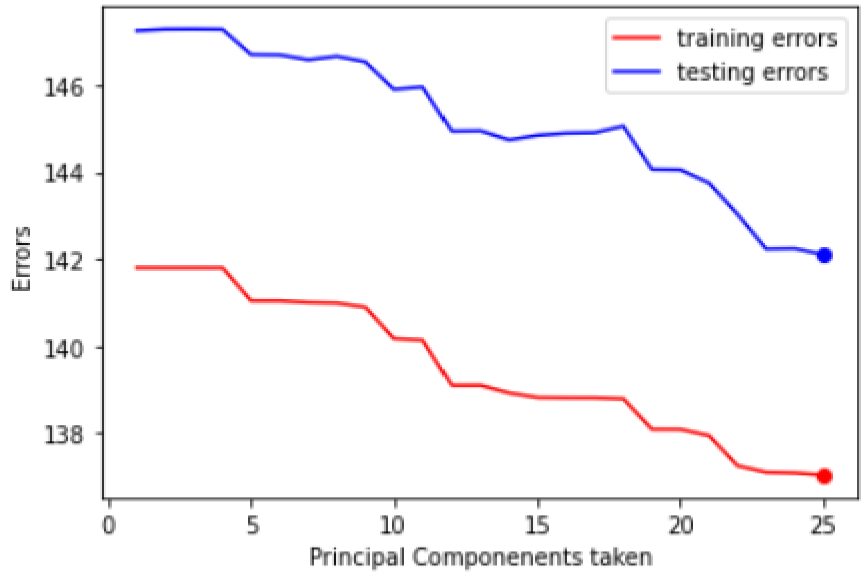
	Training Errors
1	141.806336
2	141.805121
3	141.805055
4	141.804661
5	141.046757
6	141.046028
7	141.008029
8	140.994529
9	140.897522
10	140.176164
11	140.141257
12	139.102327
13	139.102261
14	138.926607
15	138.821759
16	138.815448
17	138.814718
18	138.792200
19	138.093482
20	138.090881
21	137.947179
22	137.253434
23	137.103529
24	137.089031
25	137.039557

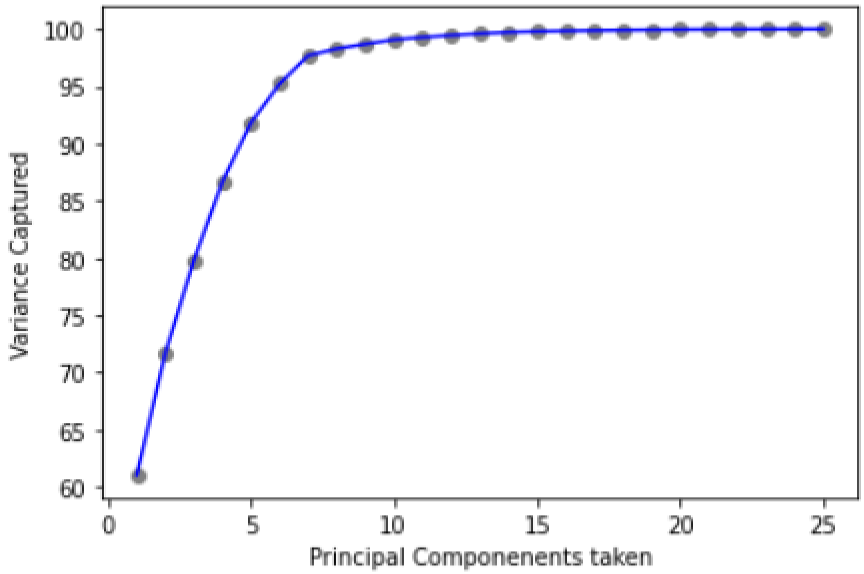
	Testing Errors
1	147.251388
2	147.285147
3	147.287591
4	147.283200
5	146.702242
6	146.694875
7	146.581247
8	146.661424
9	146.533602
10	145.904989
11	145.958475
12	144.941456
13	144.948541
14	144.740768
15	144.846122
16	144.895424
17	144.906170
18	145.057148
19	144.061867
20	144.053624
21	143.750798
22	143.032936
23	142.238118
24	142.247400
25	142.104912

	Precentage Variance Captured
1	61.046231
2	71.705980
3	79.814644
4	86.686074
5	91.836998
6	95.222237
7	97.690777
8	98.284759
9	98.688871
10	99.076425
11	99.283168
12	99.459615
13	99.598666
14	99.729044
15	99.795695
16	99.843243
17	99.880198
18	99.912615
19	99.937752
20	99.958555
21	99.973338
22	99.984469
23	99.991367
24	99.996263
25	100.000000

The above DataFrame gives us the percentage variance for the principal components taken.

```
: Text(0, 0.5, 'Errors')
```





We can observe that the variance becomes almost constant after 7 principal components.

2-B:

GREEDY FORWARD:

We implement greedy forward in the following way:

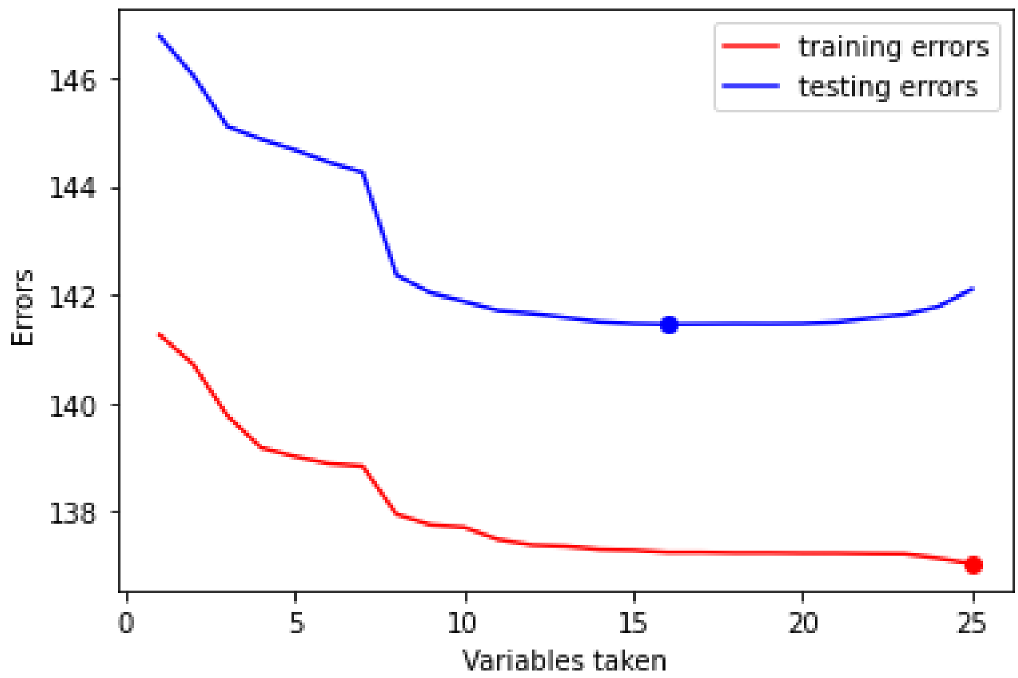
```
for i in range(1,26):

    test_error_greedy = np.inf
    x_data_greedy = []
    x_data_greedy_test = []

    temp_var = -1
    num_var = np.size(variables_taken)

    for var in variables_taken:
        x_data_greedy.append(x_data[:,var:var+1])
        x_data_greedy_test.append(x_data_test[:,var:var+1])

    for j in range(0,25):

        if(variables_taken.count(j) == 0):

            x_data_temp = x_data_greedy.copy()
            x_data_test_temp = x_data_greedy_test.copy()
            # np.concatenate([x_data_greedy,x_data[:,j:j+1].reshape(-1,1)], axis=1)
            x_data_temp.append(x_data[:,j:j+1])
            x_data_test_temp.append(x_data_test[:,j:j+1])
            x_data_req = np.asarray(x_data_temp).T.reshape(-1,num_var+1)
            w_req = np.dot(np.dot(np.linalg.inv(np.dot(x_data_req.T,x_data_req)),x_data_req.T),y_data)

            x_data_test_req = np.asarray(x_data_test_temp).T.reshape(-1,num_var+1)
            model_test_error = calculateErrorTest(x_data_test_req, w_req)

            if(model_test_error < test_error_greedy):

                test_error_greedy = model_test_error
                temp_var = j

    testing_errors_greedy_for.append(test_error_greedy)
    variables_taken.append(temp_var)
    print(variables_taken,"\n")
```

We are traversing through all the features one by one and selecting one more feature to add to our feature set in every iteration. We then iterate through all the other features that are not taken, and select the best feature, and so on.

	Variable Taken	Testing Errors
1	RH_out	146.779739
2	RH_1	146.049745
3	RH_7	145.112137
4	RH_2	144.875617
5	RH_8	144.676760
6	Windspeed	144.450142
7	T3	144.260919
8	T9	142.362126
9	T2	142.038790
10	RH_4	141.879393
11	T8	141.713060
12	T6	141.656926
13	T4	141.583115
14	Tdewpoint	141.503205
15	Visibility	141.468789
16	RH_5	141.459234
17	Press_mm_hg	141.462719
18	T_out	141.466975
19	T1	141.465218
20	T5	141.469711
21	rv1	141.497895
22	RH_9	141.574799
23	T7	141.637682
24	RH_6	141.783521
25	RH_3	142.104912

The best possible feature subset (Most significant features) is:

(Numbers can be mapped to the dataset in the same order)



We observe that through greedy forward method, we are getting minimum testing error for the 16 features taken.

The best possible feature subset (Most significant features) is:

```
[20, 1, 13, 3, 15, 21, 4, 16, 2, 7, 14, 10, 6, 23, 22, 9]
```

These are corresponding to:

```
['RH_out', 'RH_1', 'RH_7', 'RH_2', 'RH_8', 'Windspeed', 'T3', 'T9', 'T2', 'RH_4', 'T8', 'T6', 'T4', 'Tdewpoint', 'Visibility',
'RH_5']
```

GREEDY BACKWARD:

We implement greedy backward in the following way:

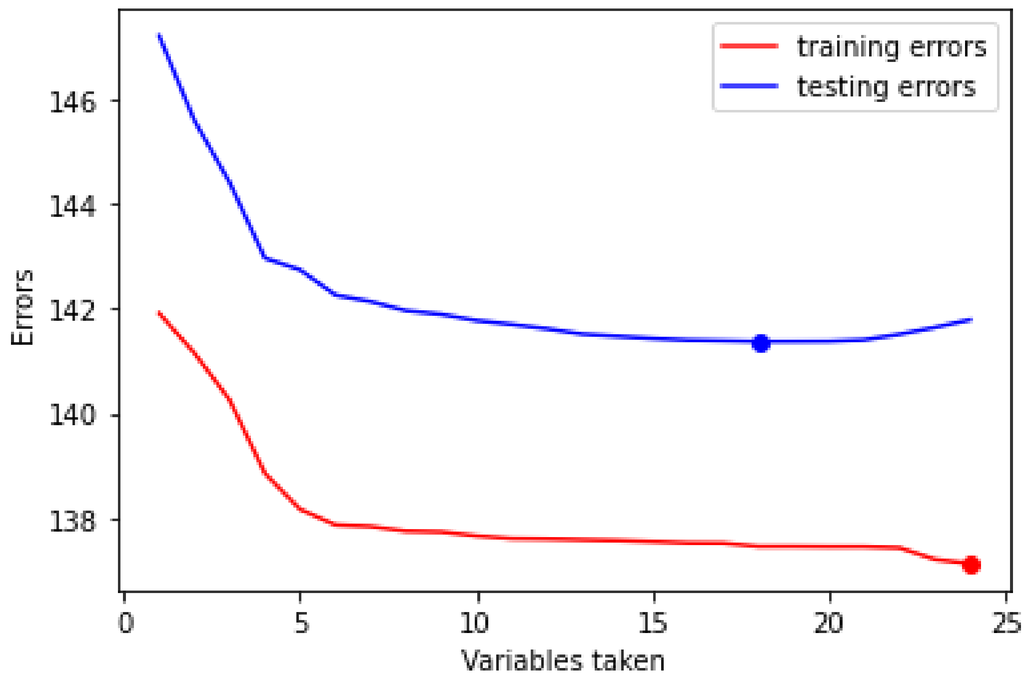
```
for i in range(1,25):

    test_error_greedy = np.inf
    temp_var = -1

    # temp_removal = variables_removed.copy()
    # x_data_greedy = np.delete(x_data,variables_removed,1)
    # x_data_greedy_test = np.delete(x_data_test,variables_removed,1)

    for j in range(0,25):

        if(variables_removed.count(j) == 0):

            temp_removal = variables_removed.copy()
            temp_removal.append(j)
            x_data_temp = np.delete(x_data,temp_removal,1)
            x_data_test_temp = np.delete(x_data_test,temp_removal,1)
            # print(x_data_test_temp.shape)

            w_req = np.dot(np.dot(np.linalg.inv(np.dot(x_data_temp.T,x_data_temp)),x_data_temp.T),y_data)
            # print(w_req)

            model_test_error = calculateErrorTest(x_data_test_temp, w_req)

            if(model_test_error < test_error_greedy):

                test_error_greedy = model_test_error
                temp_var = j

    testing_errors_greedy_back.append(test_error_greedy)
    variables_removed.append(temp_var)
    print(variables_removed,"\n")
```

We are iterating through the features, we are removing one feature each time, and the error for the remaining features is calculated. We then remove the feature where the testing error generated is the minimum for the subset of the remaining features.

	Variable Removed	Testing Errors
1	RH_3	141.783521
2	RH_6	141.637682
3	T8	141.507237
4	RH_9	141.401054
5	rv1	141.374963
6	T5	141.372463
7	Tdewpoint	141.371457
8	RH_5	141.384768
9	RH_out	141.396130
10	Windspeed	141.427886
11	Visibility	141.468469
12	T4	141.511722
13	Press_mm_hg	141.613135
14	T7	141.695706
15	T_out	141.769677
16	T6	141.888544
17	T1	141.959593
18	RH_7	142.133000
19	RH_4	142.254555
20	T2	142.735049
21	RH_2	142.958427
22	RH_1	144.401342
23	RH_8	145.583402
24	T9	147.191756

These are the testing errors for the greedy backward implementation. We are removing 7 features, and using 18 features for our best model. The features removed are:

```
['RH_3', 'RH_6', 'T8', 'RH_9', 'rv1', 'T5', 'Tdewpoint']
```



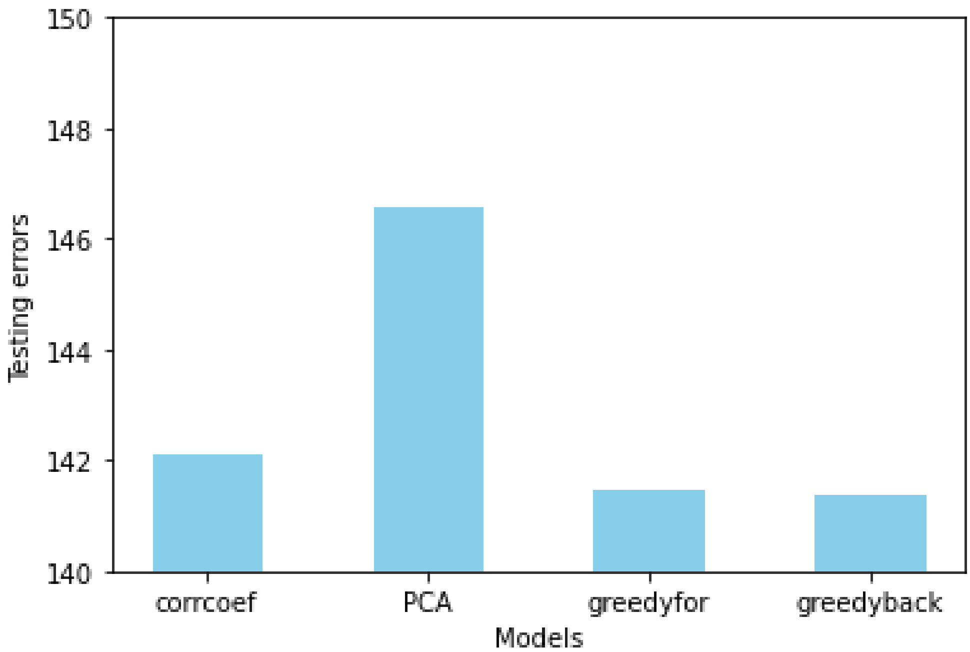
As we can observe from the above graph, using greedy backward approach, we get the minimum testing error at a subset with 18 features.

2-C:

COMPARATIVE ANALYSIS:

From the above models generated, we take the best models generated by the 4 methods, i.e., Pearson correlation coefficients, Principal Component Analysis, Greedy Forward and Greedy Backward.

Upon plotting the testing errors for the best models of each, we observe the graph to be as follows:



We are getting the least testing error for the best model generated by greedy backward approach, that is, when we have developed a model for a subset of 18 significant features, after removing 7 features.