DOCUMENTATION

2A(i)

Pearson Correlation:

A tool to measure the strength of the linear relationship between two variables, with a range of [-1,1] where -1 means complete negative correlation and 1 indicates absolute positive correlation.

$$r = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 \sum (y_i - \bar{y})^2}}$$

r = correlation coefficient
x_i = values of the x-variable in a sample
$\bar{x}$ = mean of the values of the x-variable
y_i = values of the y-variable in a sample
$\bar{y}$ = mean of the values of the y-variable

Implementation:

We start the implementation by first doing a train-test split of 80/20 form.
Since the values of columns **'rv1'** and **'rv2'** have identical values leads to multicollinearity since here the correlation between the two predictors is 1.
This may lead to incorrect predictions; hence we need to exclude one of the features.

Then we create a function to find the correlation between 2 features given by the **correlation_matrix.** Thus using this function, we see the correlation between the dependent variable(Appliances) and the independent features(T1, RH_1,..., etc.).

Following this, we sort the relationship of the corresponding independent features over the dependent variable according to its absolute value.

Then we start creating models in such a way that the first model is based on the feature with the highest correlation coefficient, then with the highest and the second highest correlation coefficient, and so on.

We have used the **create_model** function for the required job. This function takes features from the variablle_list list, which stores the features in the order of high correlation coefficient.
We have created the model using the OLS method, where the parameters(weights) are calculated using the function:

$$w = \left(X^T.X\right)^{-1}.X^T.Y$$

Along with this, we are also calculating the training and testing errors of each model using the formula:

$$\mathrm{RMSD} = \sqrt{\frac{\sum_{i=1}^{N}\left(x_i - \hat{x}_i\right)^2}{N}}$$

RMSD	= root-mean-square deviation
i	= variable i
N	= number of non-missing data points
x_i	= actual observations time series
$\hat{x}_i$	= estimated time series

Lastly, we have plotted the graph's comprehensive set of features for their training and testing errors.

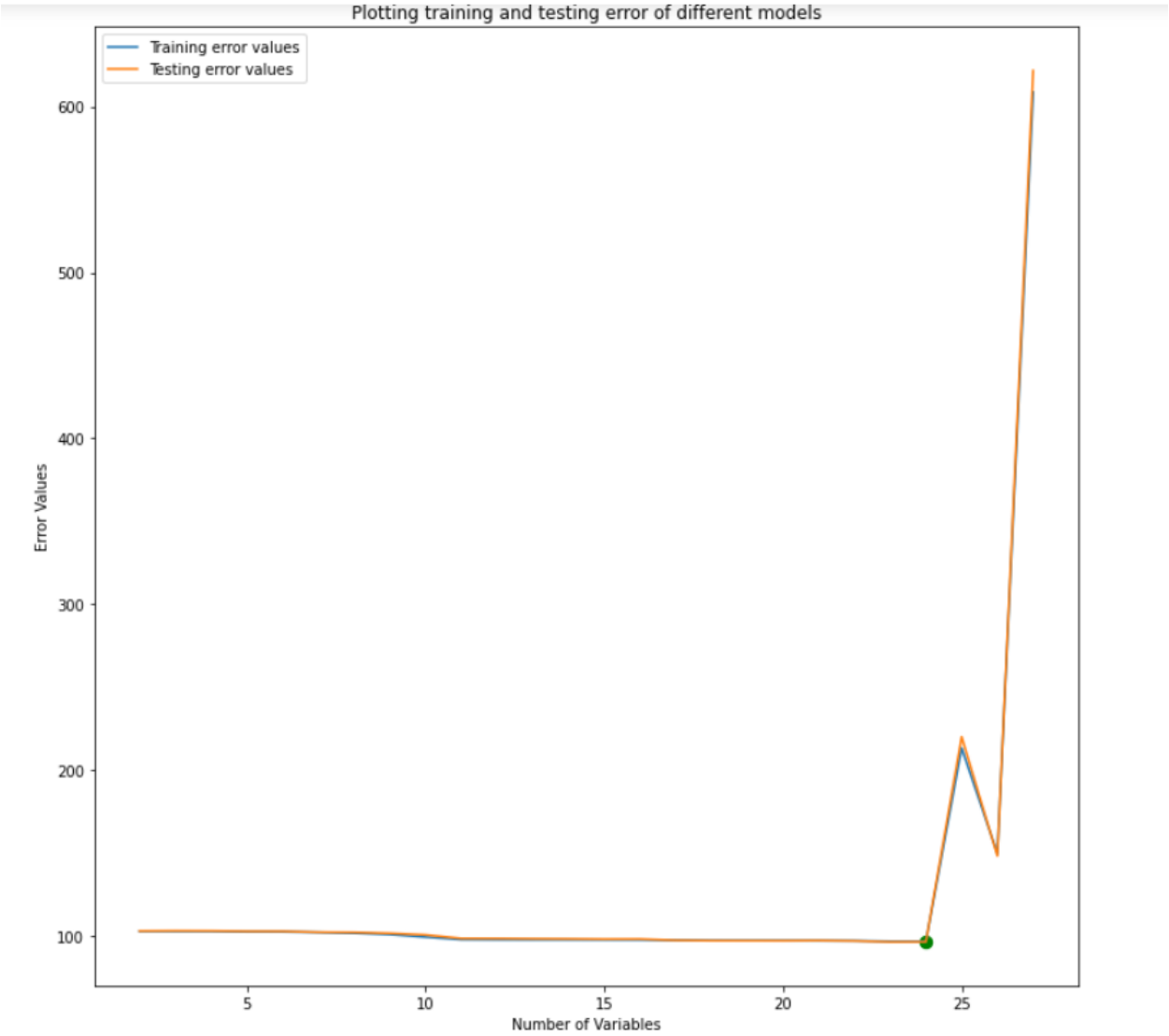
NOTE:

```
['RH_out',
 'T6',
 'T2',
 'T_out',
 'RH_8',
 'Windspeed',
 'RH_6',
 'T3',
 'RH_1',
 'RH_2',
 'RH_7',
 'T1',
 'RH_9',
 'T4',
 'T8',
 'T7',
 'RH_3',
 'Press_mm_hg',
 'Tdewpoint',
 'T5',
 'RH_4',
 'T9',
 'rv1',
 'rv2',
 'Visibility',
 'RH_5']
```

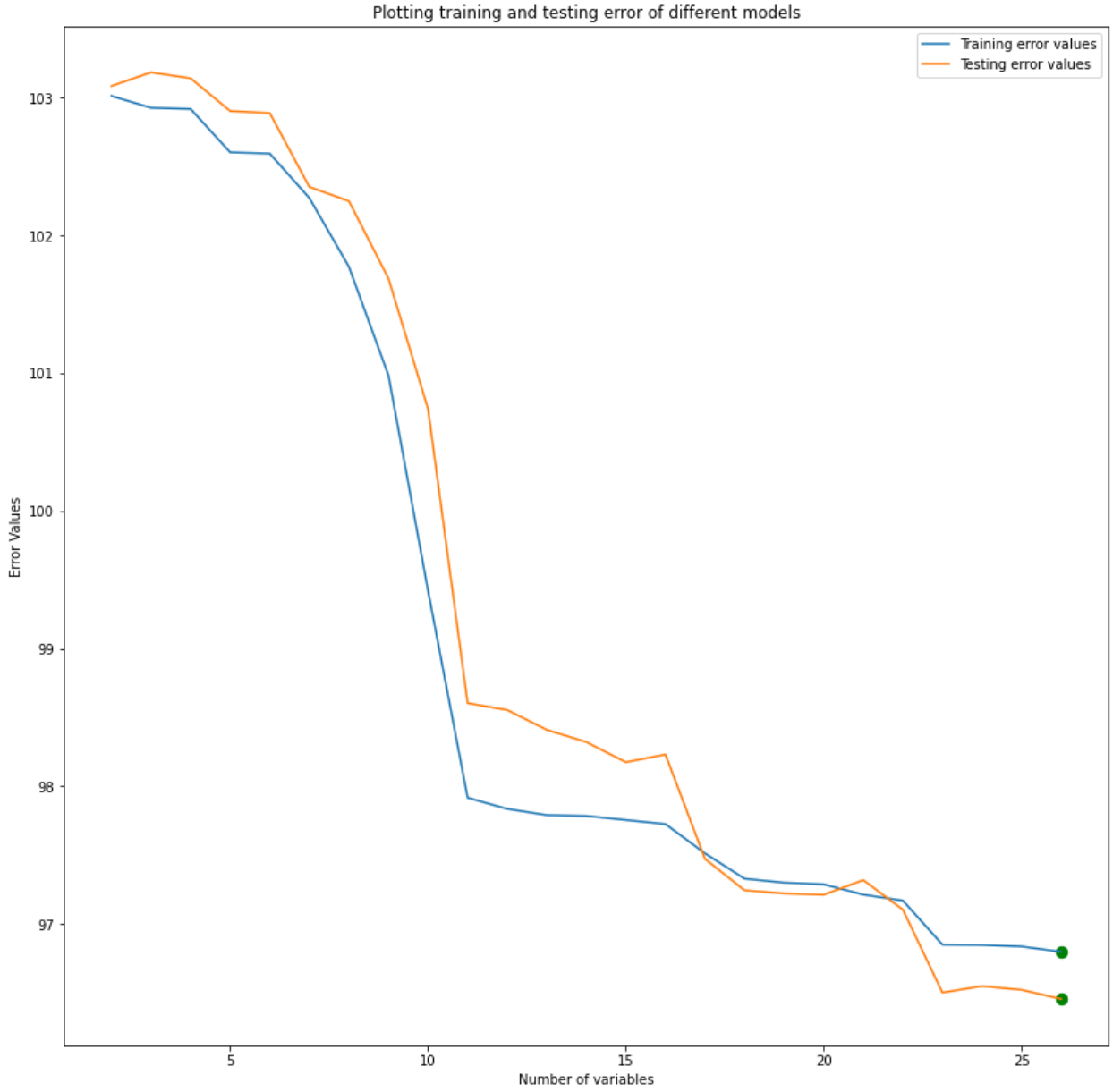
The given list of variables are in decreasing order of correlation with the dependent variable. Thus we select variables in the given descending order.

Plots:

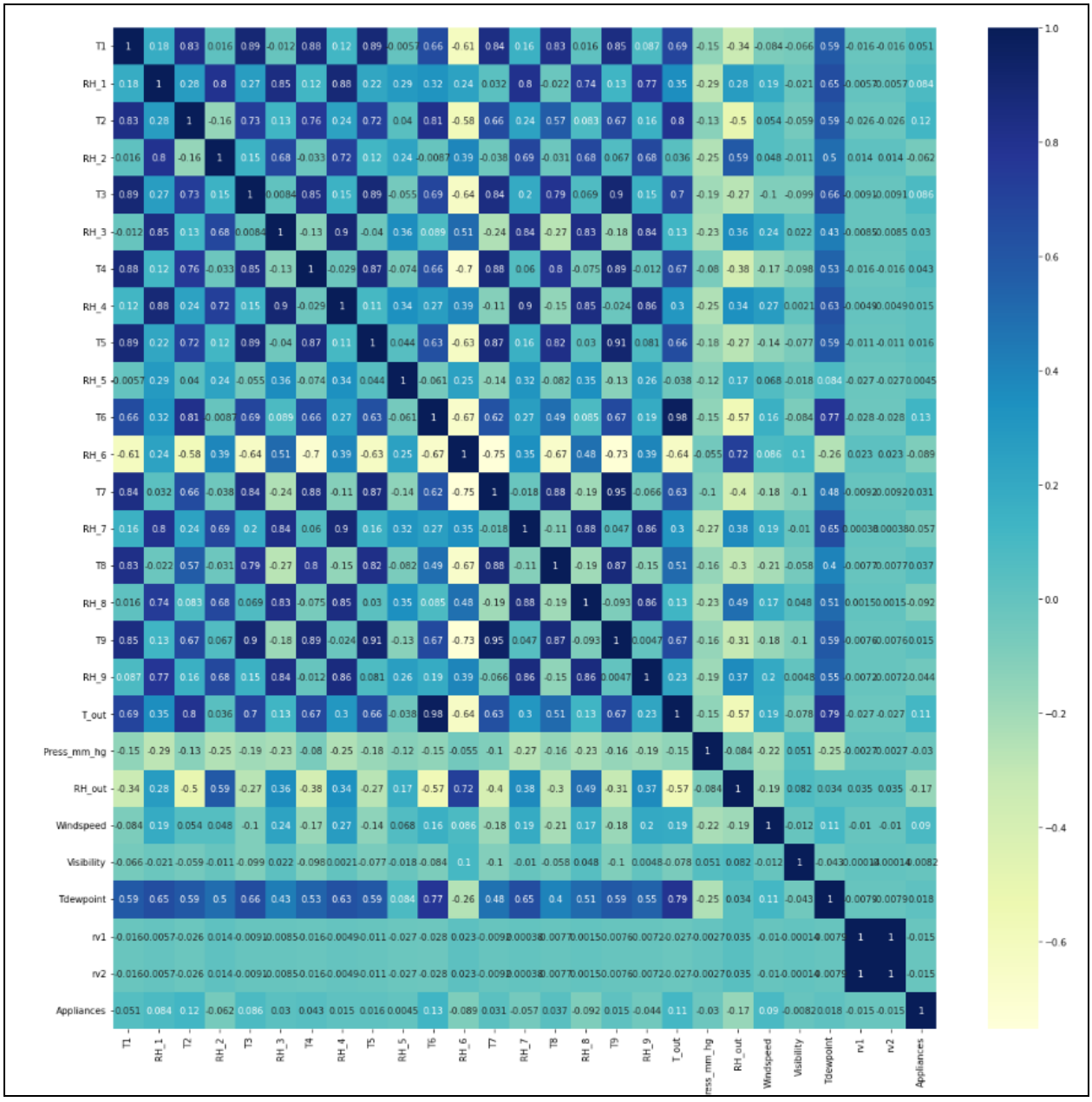
A graph without removing the perfectly correlated features: here, we can see that the error rises to a very high value upon using both the correlated features. This is because now it won't capture minimum variance, which might be possible upon dropping it.



Thus upon dropping one of the correlated features, the error value falls, and we can see the plot as follows:



The above plot shows that the best model can be found when all the features are used, with training and testing being the minimum.
The reason behind this is that now that we have dropped one of the perfectly correlated features, the model is good and captures 100% of the variance.



Here we have plotted a heatmap based on the correlation matrix. rv1 and rv2 have a correlation coefficient = 1 which indicates a perfect correlation relation between them, leading to multicollinearity. Here a color gradient shows the weight of correlation between two variables. We need the correlation coefficient of the dependent variable with all the independent variables.

Tabulating the error values:

Model	*TrainingError*	*TestingError*
1	103.01249294865849	103.0850185672770
2	102.92648600023334	103.18433080754977
3	102.91824689956155	103.1408781424227
4	102.60330040010513	102.90308600674132
5	102.59460938996271	102.88896024374168
6	102.2719676645556	102.35267071050512
7	101.77240369930104	102.24911485175242
8	100.98250478695951	107.05840119559274
9	99.41979905090615	100.73957726055038
10	97.91611031695749	98.60285353993905
11	97.8348649017202	98.553365701794
12	97.78994790549147	98.40932939249959
13	97.78388669834328	98.32088239462655
14	97.75422898415252	98.17415040192516
15	97.72504580334483	98.23010547258943
16	97.51195649157856	97.47172580166838
17	97.32813365481017	97.24378062718542
18	97.29929757160482	97.22068170809814
19	97.28733904616394	97.21149762609852
20	97.21224927238268	97.31820522826706
21	97.16927185961367	97.10323906809099
22	96.84902521456532	96.50085231448197
23	96.84679300082908	96.54771823642461
24	213.3618929046001	220.13817763054755
25	149.75204142606088	148.30450193826888
26	608.7832551513321	621.7156357793768

Again from the table, we can see that the best model is the one with the least error values, Model 23, with 23 features and 24 parameters(intercept included).

2A(ii)

Method

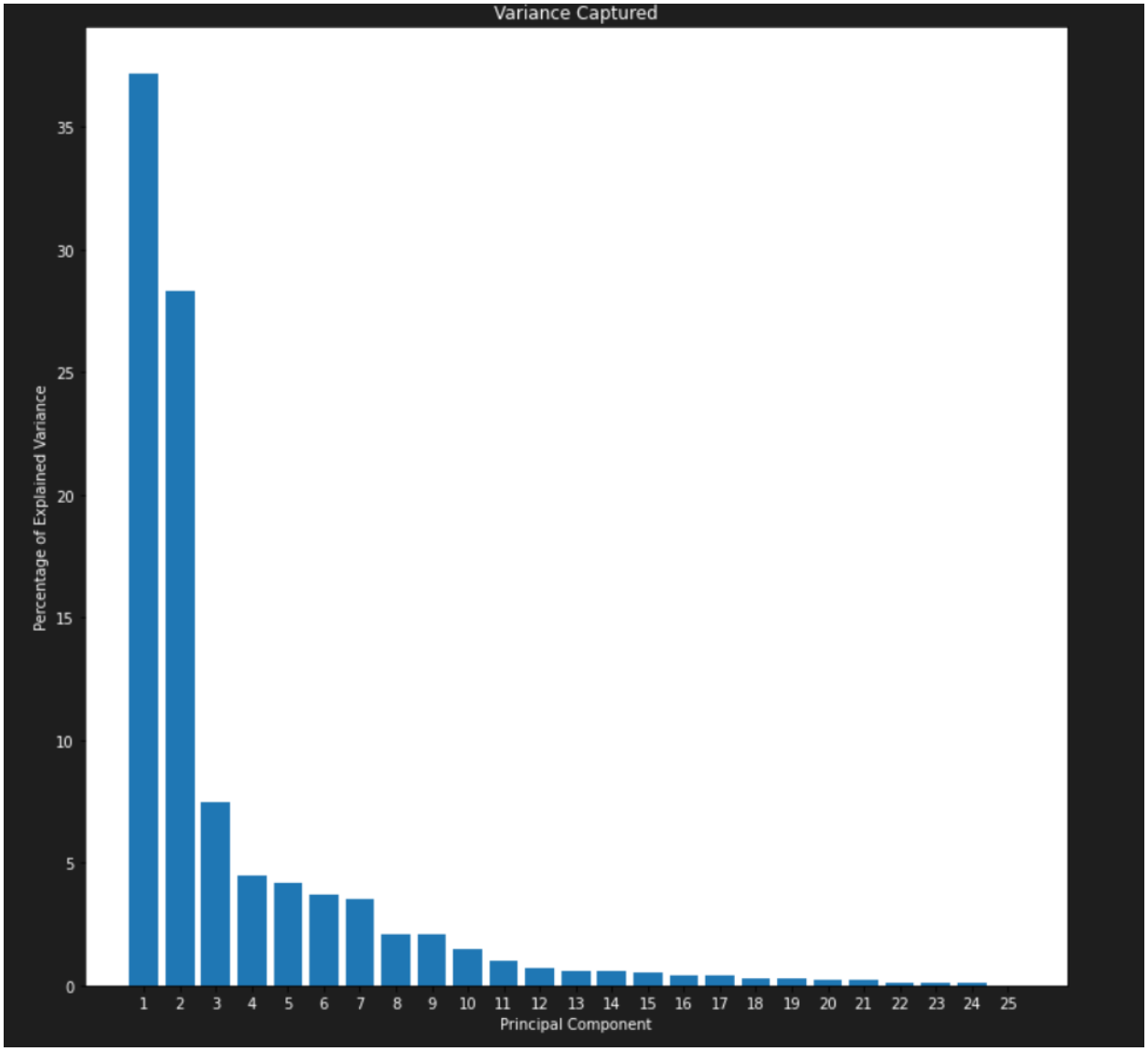
Principal component analysis (PCA) is a popular technique for analyzing large datasets containing a high number of dimensions/features per observation, increasing the interpretability of data while preserving the maximum amount of information captured in terms of variance and enabling the visualization of multidimensional data.

The **Principal Components** of a collection of points in an absolute coordinate space are a sequence of p-unit vectors, where the i-th vector is the direction of a line that best fits the data while being **orthogonal** to the first i-1 vectors.

Implementation

So we are using the sklearn library of python for implementing **PCA**. It has an inbuilt function for implementing PCA.
Firstly, to implement PCA, we need to **scale the data**. This is because data can have large values that may sway the data even with relatively little variability.

Observing the data given, we get to know that values for **rv1** and **rv2** are the same. So we need to **drop the rv2 feature** from our dataset.
We now do PCA to get principal components for the dataset.
Different Principal Components capture an additional amount of variance of the complete data.



We can see that Principal Component 1 captures the maximum variance of the data, then Principal Component 2 captures the second maximum variance of the data, and so on.

We now need to construct models for different scenarios. Like we would take multiple sets, say
Model 1 will have **Principal Component 1**,
Model 2 will have **Principal Component 1** and **Principal Component 2**,
Model 3 will have **Principal Component 1**, **Principal Component 2**, **Principal Component 3**, and so on.

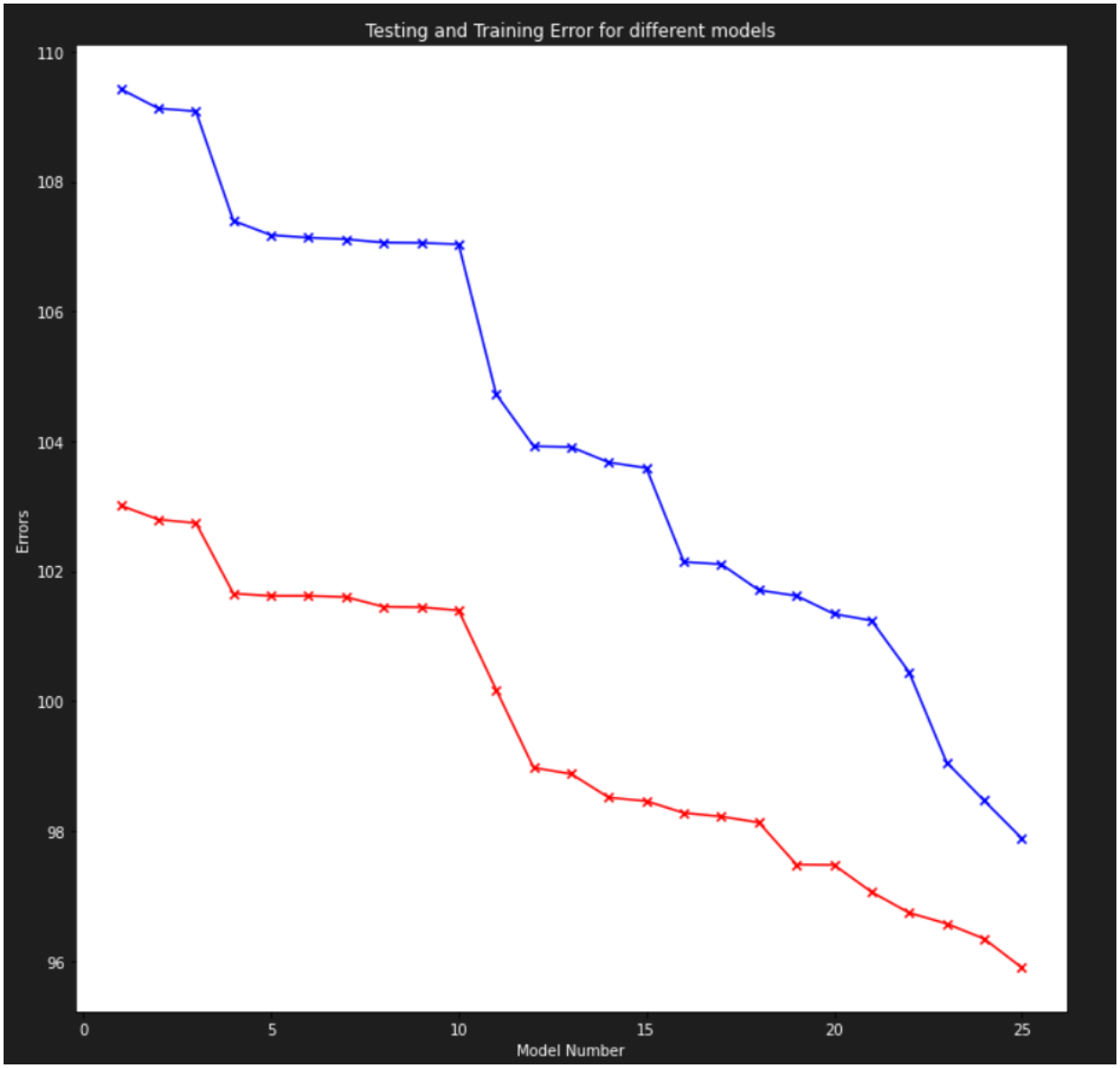
Assuming the **threshold variance** as 80% of the total data (General Practice), **Model 5** is the Best since it captures about **81.7% of the total variance**.

In the **create_model** function, we calculate the value of parameters, i.e., w's. We do this by using the vector form of **OLS(Ordinary Least Square).**

$$w = \left(X^T.X\right)^{-1}.X^T.Y$$

We now calculate the training and testing errors for all models.
Error is calculated using RMS(Root Mean Square) method.

$$RMSE = \sqrt{\frac{\left(Y_{predicted} - Y_{training}\right)^T.\left(Y_{predicted} - Y_{training}\right)}{n}}$$

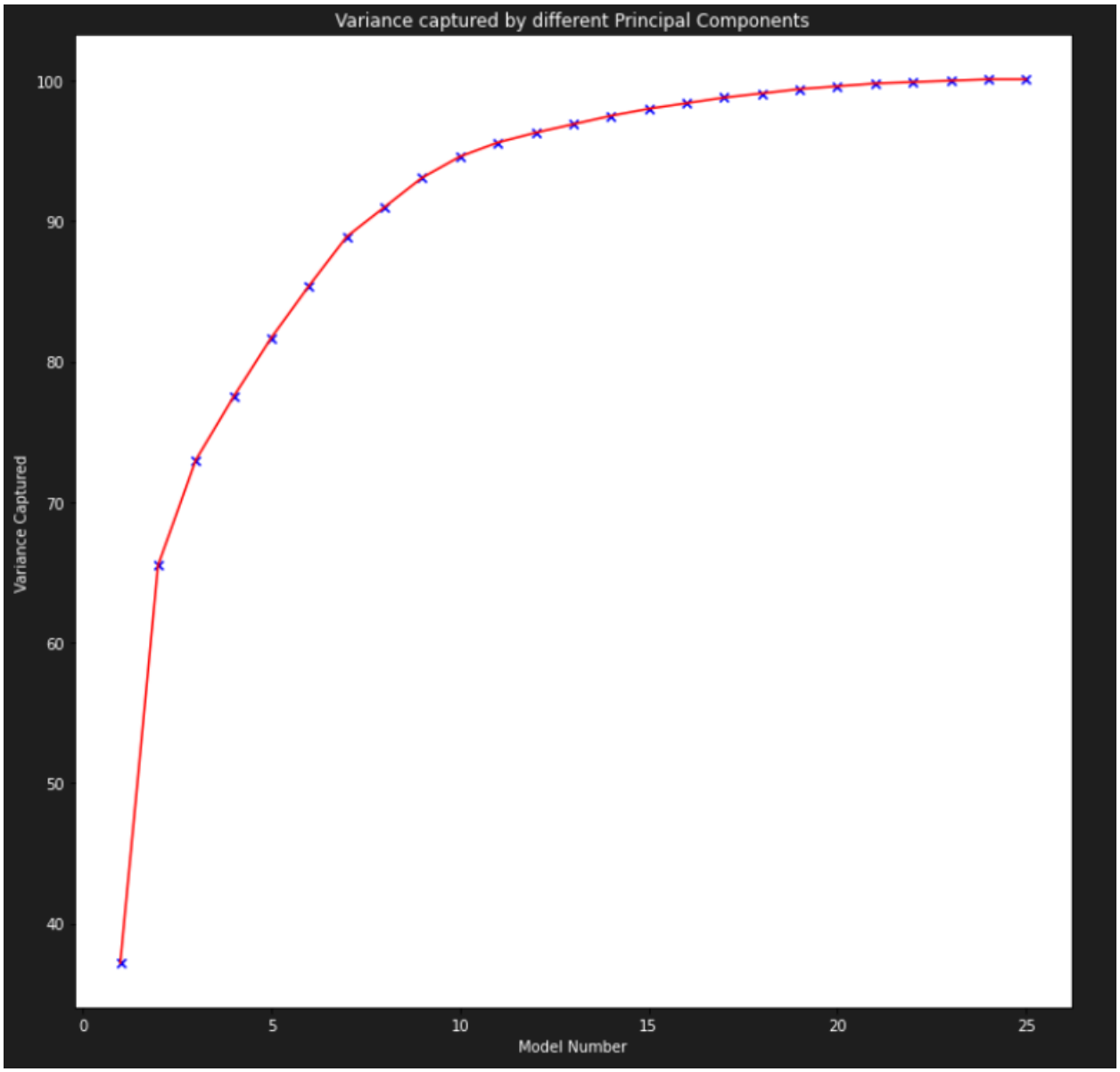


The error will reduce as we use more Principal Components in our model.

So we can clearly see that Model 25 has a minimum error and will capture 100% variance of the data. So the best model is where all the features except rv2 are involved.

If we try to plot the error for Model 26, the error value will shoot up to a considerable value, say 608.783 for the training data and 621.716 for the testing data.
The reason for this increase in standard error is that the model will not capture minimum variance. The model will overfit our data. Thus the testing error goes up.

Plot represents the percentage of variance captured by each principal component



Tabulating Error Terms

Model	*TrainingError*	*TestingError*
1	103.0140077740906	109.4271833702824
2	102.79934131813181	109.12932227883569
3	102.74539006646543	109.08496275559064
4	101.65878236732495	107.39929297084596
5	101.62354752197545	107.17846061353058
6	101.60534403067508	107.11479706804533
7	101.4561865296695	107.06404982358215
8	101.44944928958252	107.05840119559274
9	101.3994863413764	107.0381894670923
10	100.17772250507218	104.72835973134205
11	98.97725644869789	103.92725508747722
12	98.8818465256357	103.91528729957902
13	98.51775550838737	103.67923967840038
14	98.46409612975313	103.59263809837208
15	98.28151630085681	102.14803139824097
16	98.22570272245324	102.10968927755653
17	98.13350050350259	101.71103091464391
18	97.48696790446353	101.62498706287043
19	97.48115818084474	101.34568914822124
20	97.06466132061853	101.24399564484712
21	96.74435835972952	100.45127655406783
22	96.57678745736447	99.06180640119469
24	96.3436476976596	98.47130997249114
25	95.90167706184737	97.89187562414813

From the table, we can see the best model is one with 25 features included, i.e., all excluding the dropped column rv2.

2B(i)

Implementation

Greedy forward implementation is used to select features to construct a regression model giving the least error.
The final model is the one that includes only a subset of the features used out of all available in the data frame.

Method

We construct a regression model for a single feature from all the sets of 26 features given.
Thus we get a total of 26 different models of the form y = b0 + b1x, where x is the feature selected from the dataset.
Then we calculate the rmse error for each of the given models. The model with the least error is then used in the next iteration. Thus feature xi, where xi corresponds to the feature showing the least error as a linear model, is selected.
The process is repeated by selecting two features in the next iteration with the model as y = b0 + b1x1 + b2x2 where x1 is fixed (from the previous iteration) and x2 is varied again.
This process is repeated until adding a new feature adds an error to the current model rather than decreasing the error.
We say the subset of the selected features results in an optimal model for the given dataset.

Features in the best model:

```
print("Feaures used in best model of greedy forward are:")
print(df_train.columns[required_columns].values)
```

```
Feaures used in best model of greedy forward are:
['intercept' 'RH_out' 'Windspeed' 'RH_8' 'RH_1' 'RH_2' 'RH_7' 'T9' 'T3'
 'T2' 'T8' 'RH_4' 'T6' 'T4' 'T_out' 'RH_5' 'Visibility' 'Press_mm_hg' 'T1']
```

A total of 19 parameters (18 features) are used in the optimal model. We can see that after tabulating testing errors after each iteration, the testing error decreases. The error is the least when 19 parameters are used. After that, the error increases; hence we include the 19 parameters stated above.

Model	*TestingError*
1	103.08501856727703
2	102.79927594410029
3	102.65085039842643
4	101.49008546017399
5	100.27088258653588
6	99.93263640984196
7	99.75903671263819
8	97.37936665299709
9	97.09225177573028
10	96.69493566589769
11	96.49700124823705
12	96.38226280866787
13	96.26225601360346
14	96.14092884633389
15	96.05332388559341
16	96.01657686025369
17	96.00980683687807
18	96.00871445629413

Minimum Training Error: 97.03336171252752

The testing and training errors are also mentioned for the best model.

2B(ii)

Implementation

Greedy backward implementation is used to select features to construct a regression model with the least error.
The final model is the one that includes only a subset of the features used out of all available in the data frame. Here we progress in reverse order, i.e., starting with all features in the given model and then decreasing until we find the optimal model.

Method

We start by initializing a list containing all the dataset's variables. Then we iterate over all the variables in the list by popping each one at a time and generating a model. Each contains one feature less than the first model we started with. The model giving the least error is chosen. Thus, the following model is taken by removing that particular feature. So the model contains 25 features in the next iteration. We keep eliminating features until removing a feature does not decrease the error further.
This model is considered the optimal model.

Features in the best model:

```
In [30]: print("Feaures used in best model of greedy backward are:")
         print(df_train.columns[required_columns].values)

         Feaures used in best model of greedy backward are:
         ['intercept' 'RH_out' 'Windspeed' 'RH_8' 'RH_1' 'RH_2' 'RH_7' 'T9' 'T3'
          'T2' 'T8' 'RH_4' 'T6' 'T4' 'T_out' 'RH_5' 'Visibility' 'Press_mm_hg' 'T1']
```

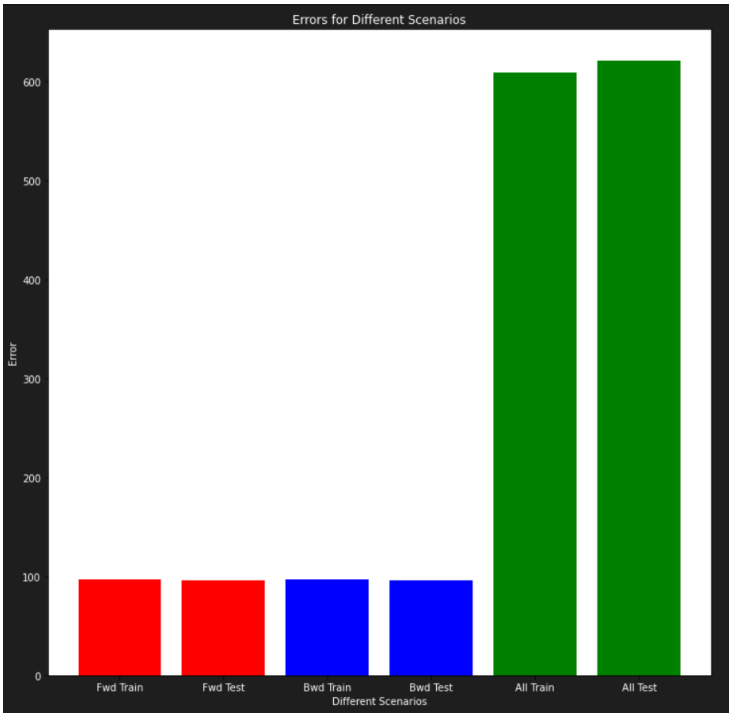
A total of 18 parameters (17 features) are used in the optimal model. We can see that the testing error is decreasing after tabulating testing errors after each iteration. The error is the least when 17 parameters are used. After that, the error increases; hence we include the 19 ones stated above, and no additional features are reduced during greedy backward feature selection.

Model	*TestingError*
25	96.30520853360514
24	96.20267262751182
23	96.15744756432093
22	96.12102761946691
21	96.10944126019082
20	96.01415601555281
19	96.00871445630679
18	95.99014483070111

Minimum Training Error: 97.11895376759993

The testing and training errors are also mentioned for the best model.

Observations:



The bar plot shown shows that the error values for the model using all its features are very high. This is because we encounter multicollinearity due to the same values in the columns "rv1" and "rv2."

The models' errors for the forward and backward greedy method are low because it generates optimal models with low errors.

Comparative Analysis between all the best models generated from different feature selection techniques:

Pearson Correlation:
The best model generated is with one having 23 features and 24 parameters(including intercept) where the training error and testing error are 96.8467 and 96.457 respectively.

Principal Component Analysis:
The best model generated is one with all features included except the one dropped feature, leading to multicollinearity with training and testing errors being 95.901 and 97.891 respectively.

Greedy Forward Approach:
The best model generated is one with 19 parameters (including intercept) having 18 features with traning and testing error being 97.033 and 96.009.

Greedy Backward Approach:
The best model generated is one with 18 parameters (including intercept) having 17 features with traning and testing error being 97.119 and 95.990.

Overall Best Model

The overall best model generated is one with the lowest testing error values.
In this case its one from the **greedy backward approach** with testing error 95.990 and 17 features selected.
Although if we increase the threshold limit of variance in PCA(which is 80% currently), it shall also give the best model with minimum testing error. Adding we can see that PCA generates minimum training error.