CS-GY 6923 Machine Learning

Professor: Dr. Raman Kannan

HW1: First Modeling Assignment

Remon Roshdy

Contents

1 Overview of the Data set

The data set is called "Smartphone-Based Recognition of Human Activities and Postural Transitions" and was taken from the UCI repository:
`http://archive.ics.uci.edu/ml/datasets/Smartphone-Based+Recognition+of+Human+Activities+and+Postural+Transitions`
It is an activity recognition data set built from the recordings of 30 subjects performing basic activities and postural transitions while carrying a waist-mounted smartphone with embedded inertial sensors. The features are composed of parameters calculated from the signals obtained through the inertial sensors, and the activity is recorded as the output variable to be predicted.

The data set consists of 561 features (independent variables), 1 dependent variable and 7767 observations. The dependent variable has categorical integer values ranging from 1 to 12; hence, this is a classification problem. And since the number of classes is larger than two, it is a multi-class classification. Before loading the data file into R, labels were assigned to the variables, and the file was saved as a csv file. The label "activity" was allocated to the class variable, and the independent variables were assigned with numerical labels ranging from 1 to 561 for better representation and easier manipulation of the large number of features.

2 Loading the Data

The data set, containing all the variables and labels, was saved as a csv file named `'train_num.csv'` and it was loaded into a variable called `data`.

```
> loadData=function(csvfile) { read.csv(csvfile,head=T,sep=',',stringsAsFactors=F ) } # Function to load the data
> data=loadData('train_num.csv') # Load the data
```

We then perform few steps to inspect the data to make sure that it was loaded properly. The dimensions of the data frame was found to be 7767×562, which corresponds to 7767 rows (number of observations) and 562 columns (number of variables). The number of columns includes the 561 features plus 1 column for the class variable. We also check that the labels are assigned correctly using the `names(data)` command. Since we labeled the independent variables with numerical values, R automatically inserts an "X" prior to the label as shown in the output. We can also see that the last column is the class variable, and its name is `"activity"`. All the features (independent variables) columns have continuous numerical values, so they are non-categorical.

```
> head(data)[,1:6] # Display the first few entries of the data
          X1           X2          X3         X4         X5         X6
1 0.04357967 -0.005970221 -0.03505434 -0.9953812 -0.9883659 -0.9373820
2 0.03948004 -0.002131276 -0.02906736 -0.9983480 -0.9829449 -0.9712729
3 0.03997778 -0.005152716 -0.02265071 -0.9954821 -0.9773138 -0.9847595
4 0.03978456 -0.011808778 -0.02891578 -0.9961941 -0.9885686 -0.9932556
5 0.03875814 -0.002288533 -0.02386289 -0.9982413 -0.9867741 -0.9931155
6 0.03898801 0.004108852 -0.01734027 -0.9974376 -0.9934854 -0.9966920
> dim(data) # Check the number of rows and columns of the data
[1] 7767 562
> names(data) # Check the names of each column (features and class variable)
  [1] "X1"      "X2"      "X3"      "X4"      "X5"      "X6"
  [7] "X7"      "X8"      "X9"      "X10"     "X11"     "X12"
 [13] "X13"     "X14"     "X15"     "X16"     "X17"     "X18"
 [19] "X19"     "X20"     "X21"     "X22"     "X23"     "X24"
 [25] "X25"     "X26"     "X27"     "X28"     "X29"     "X30"
 [31] "X31"     "X32"     "X33"     "X34"     "X35"     "X36"
 [37] "X37"     "X38"     "X39"     "X40"     "X41"     "X42"
 [43] "X43"     "X44"     "X45"     "X46"     "X47"     "X48"
 [49] "X49"     "X50"     "X51"     "X52"     "X53"     "X54"
 [55] "X55"     "X56"     "X57"     "X58"     "X59"     "X60"
 [61] "X61"     "X62"     "X63"     "X64"     "X65"     "X66"
 [67] "X67"     "X68"     "X69"     "X70"     "X71"     "X72"
 ...
[547] "X547"    "X548"    "X549"    "X550"    "X551"    "X552"
[553] "X553"    "X554"    "X555"    "X556"    "X557"    "X558"
[559] "X559"    "X560"    "X561"    "activity"
> length(names(data)) # Double check that the length of "names" matches the
         number of columns
[1] 562
> which(names(data)=='activity') # Position of the class variable column
[1] 562
```

3 Exploratory Data Analysis

Before we can train a model, we need to understand the data and perform some analyses in order to ensure that it is consistent with the classifier to be used. In Section 2, we saw that we have 7767 observations and 561 features; hence, this is considered to be a large data set as it is, which requires heavy computations and not all classifiers might be able to handle depending on the resources available. Let's first find the spectral count of the class variable `"activity"`, which will give us the number of classes, class names, and the number of observations for each class.

```
> # Find spectral count
> TBL=table(data$activity)
> TBL
   1    2    3    4    5    6    7    8    9   10   11   12 
1226 1073  987 1293 1423 1413   47   23   75   60   90   57 
> names(TBL)
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12"
> as.numeric(TBL)
 [1] 1226 1073  987 1293 1423 1413   47   23   75   60   90   57
> # Determine if binary or multi-class classification
> print(ifelse(length(TBL)==2, "Binary Classification", "MultiClass
        Classification"))

[1] "MultiClass Classification"
```

The output of the table(data$activity) command shows that we have 12 classes, each named with an integer number from ”1” to ”12”. And as mentioned earlier, since the number of classes are larger than 2, we are dealing with Multi-class classification. Moreover, just by looking at the number of observations for each class label, we can tell that there might be a class imbalance issue. Therefore, we run a check to determine if the number of observations of one class to another is less than 1:6, and in that case, the class would be imbalanced. The results show that we do have class imbalance, and the imbalanced classes are "7", "8", "9", "10", "11", and "12". Further analysis of the imbalance issue, and the process to deal with it, will be discussed at a later stage.

```
> # Determine if imbalanced. Criteria: class count 1:6
> if(any(TBL<(nrow(data)*1/12*1/6) | any(TBL>(nrow(data)*1/12*6)))) {
+   imbalanced_classes=names(TBL)[TBL<(nrow(data)*1/12*1/6) | TBL>(nrow(data)*1/12
        *6)]
+   print("Imbalanced Classes:")
+   print(imbalanced_classes)
+ } else print("Dataset is Balanced")

[1] "Imbalanced Classes:"
[1] "7"  "8"  "9"  "10" "11" "12"
```

Next, we need to check if standardization or normalization of the data is required. The standard deviation and mean of each feature column were calculated, and the results are plotted in Fig. 1. The figure shows the that the means are not equal to 0 and the standard deviations are not equal to 1, which implies that the data is not standardized. Nevertheless, the description of the data set mentioned that the data was normalized and bounded between [-1 1]; therefore, to verify this information, we calculate the ranges of each feature column and plot them in Fig. 2, which indicates that the data is normalized within [-1 1] and, thus, no scaling is needed.

```
> # Check if scaling and standardization are needed
> std_dev_data=round(apply(data[,-562],2,sd),4) # Standard deviation by column
> mean_data=round(apply(data[,-562],2,mean),4) # Mean for each column
> par(mfrow=c(2,1),mar=c(2,4,2,2)) # plot the standard deviation and mean of each feature column
> plot(std_dev_data)
> plot(mean_data)
> par(mfrow=c(1,1))
> ranges=apply(data[,-which(names(data)=='activity')],2,range) # min/max for each column
> par(mfrow=c(2,2)) # plot the ranges
> boxplot(ranges[,1:140])
> boxplot(ranges[,141:280])
> boxplot(ranges[,281:420])
> boxplot(ranges[,421:561])
> par(mfrow=c(1,1))
```

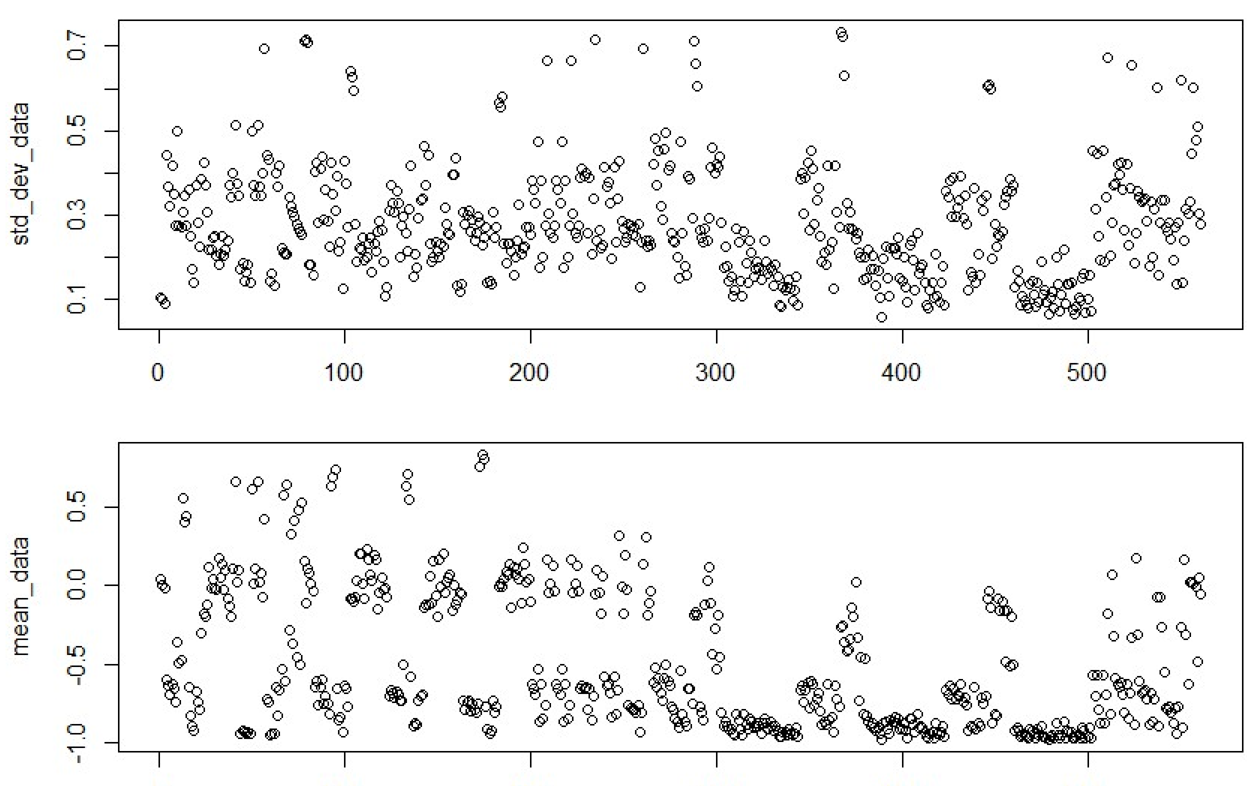

Figure 1: Calculated standard deviations (top figure) and means (bottom figures) of each feature column.

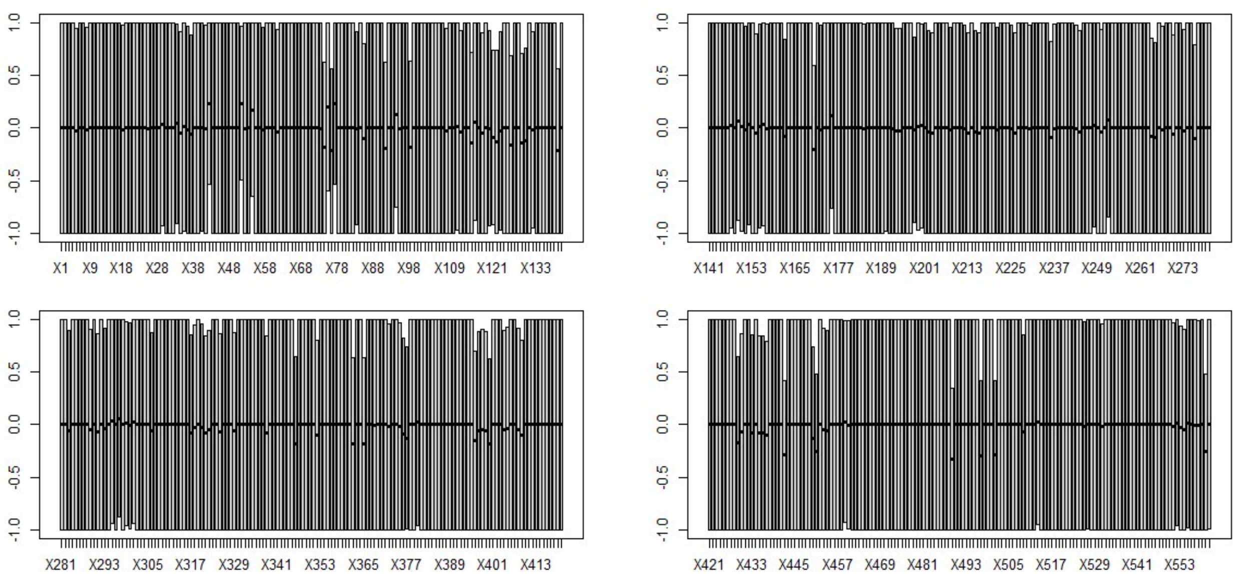

Figure 2: Boxplots showing the ranges of each feature column. Due to the large number of features, they were split into 4 figures, with each figure containing a subset of the total number of feature columns.

Thereafter, we inspect the data for constant and correlated predictors (features). Constant predictors are not required since they do not add any value in training the model, so they shall be eliminated. Similarly, if two predictors are highly correlated, then collinearity exists and, in this case, we can keep one of them and discard the other since only one of those predictors provides sufficient information to have a proper model fit. At this stage, we will only check if correlated predictors exit, whereas eliminating them will be performed at a later stage after training the first model in order to study the effect of discarding those correlated features. The output of the `pairs(data[,1:8])` command is provided in Fig. 3, which shows the scatter plots between each pair of the first 8 features. By visual inspection, we can tell that there is a strong correlation between features X4 and X7, and features X5 and X8. Using the command `cor(data[,1:561])`, we obtain the correlation matrix which includes the correlation coefficients between each pair of predictors. A correlation coefficient of 0 indicates that the predictors are uncorrelated, and a value of 1 indicates that they are perfectly correlated. Accordingly, the criteria to identify two predictors to be correlated is to have a correlation coefficient larger than 0.5 between those predictors. The results of running the R code below indicate that we have 516 correlated predictors out of the total 561 predictors in our data set.

```
> # Eliminate constant predictors
> const_pred=unlist(lapply(1:561,FUN=function(x) {
+   TBL=table(data[[x]])
+   ifelse(length(names(TBL))<2,-1*x,x)})) # Finds for each feature column if the number of distinct values are less than 2 (values of feature is constant)
> print(ifelse(any(const_pred<0),"Constant Predictors Exist","No Constant Predictors"))
> data=data[,const_pred>0]

[1] "No Constant Predictors"
```

```
> # Determine if collinearity exists (correlation between predictors). Criteria: correlation coefficient > 0.5
> pairs(data[,1:8]) # Plot the predictors one vs the other (first 8 predictors only)
> cordata=cor(data[,1:561]) # Correlation matrix (correlation coefficients)
> print(ifelse(any(abs(cordata[cordata!=1])>0.5),"Correlated Predictors Exist","No Correlated Predictors"))
> cor_index=which(abs(cordata)>0.5 & abs(cordata)!=1, arr.ind = T) # Get the indeces where the correlation coefficient > 0.5 in the correlation matrix
> cor_index=cor_index[!duplicated(cbind(pmax(cor_index[,1], cor_index[,2]), pmin(cor_index[,1], cor_index[,2]))),] # Remove duplicates, i.e. retain one of the two correlated predictors
> tbl_cor_index=table(cor_index[,1])
> cor_index_num=length(tbl_cor_index) # Number of correlated predictors
> print(paste("Number of correlated predictors =",cor_index_num))

[1] "Correlated Predictors Exist"
[1] "Number of correlated predictors = 516"
```

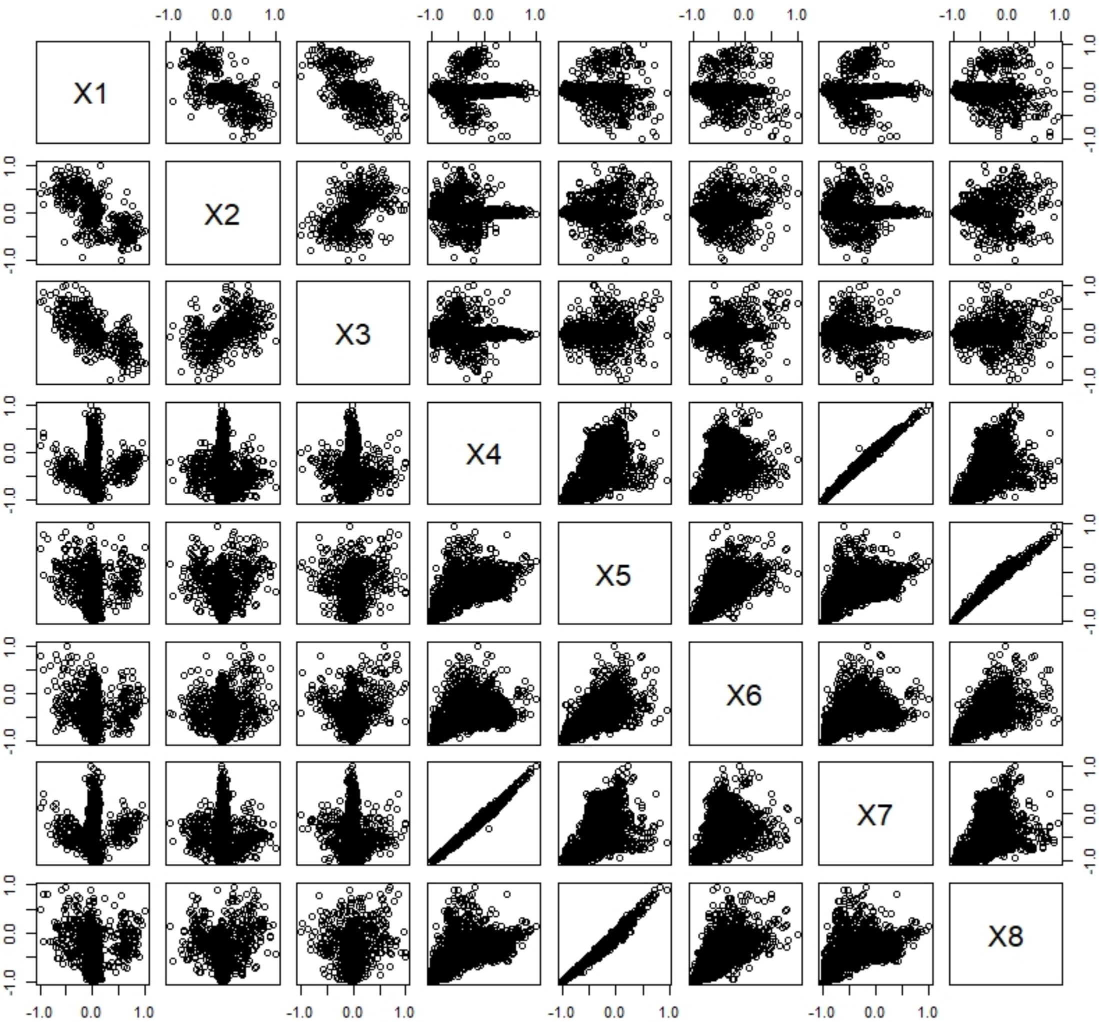

Figure 3: Scatter plots generated by the `pairs(data[,1:8])` command.

The next step is to identify the outliers in the data set. Detecting outliers was based on the 3-sigma approach, meaning that any data point or observation that falls outside three standard deviations from the mean, in each feature column, is considered to be an outlier. Histograms of the first four features are plotted and shown in Fig. 4. Here, the values that are away from the main (high frequency) observations are potential outliers. Running the R code over all feature columns, we identified 4714 out of 7767 observations as outliers. However, this identification process was based on each feature column separately; therefore, declaring an observation to be an outlier by this approach is not proper. A better way is to use a generated model and take into consideration the feature's significance and collectively consider the important features in determining if an observation is an outlier or not. Consequently, dealing with outliers will be delayed to a later stage.

```
> # Finding outliers
> par(mfrow=c(2,2)) # plot histograms of the first 4 features
> for(i in 1:4) {hist(data[,i],main=paste("Histogram of feature no.", i))}
> par(mfrow=c(1,1))
> outliers_row=c() # Loop over the feature columns
> for(i in 1:561) {
+   data_mean=mean(data[,i]) # Mean of the data in feature column i
+   data_sd=sd(data[,i]) # Standard deviation of the data in feature column i
+   low_cutoff=data_mean-3*data_sd # Lower cutoff value
+   upper_cutoff=data_mean+3*data_sd # Upper cutoff value
+   outliers_idx=which(data[,i]<low_cutoff | data[,i]>upper_cutoff) # Get the
        location of outliers in feature colum i
+   outliers_row=c(outliers_row,outliers_idx) # Append the outlier indeces to
        those of the previous feature columns
+ }
> outliers_row=unique(outliers_row) # Remove duplicated row indeces
> print(paste("Number of Outilers =",length(outliers_row)))

[1] "Number of Outilers = 4714"
```

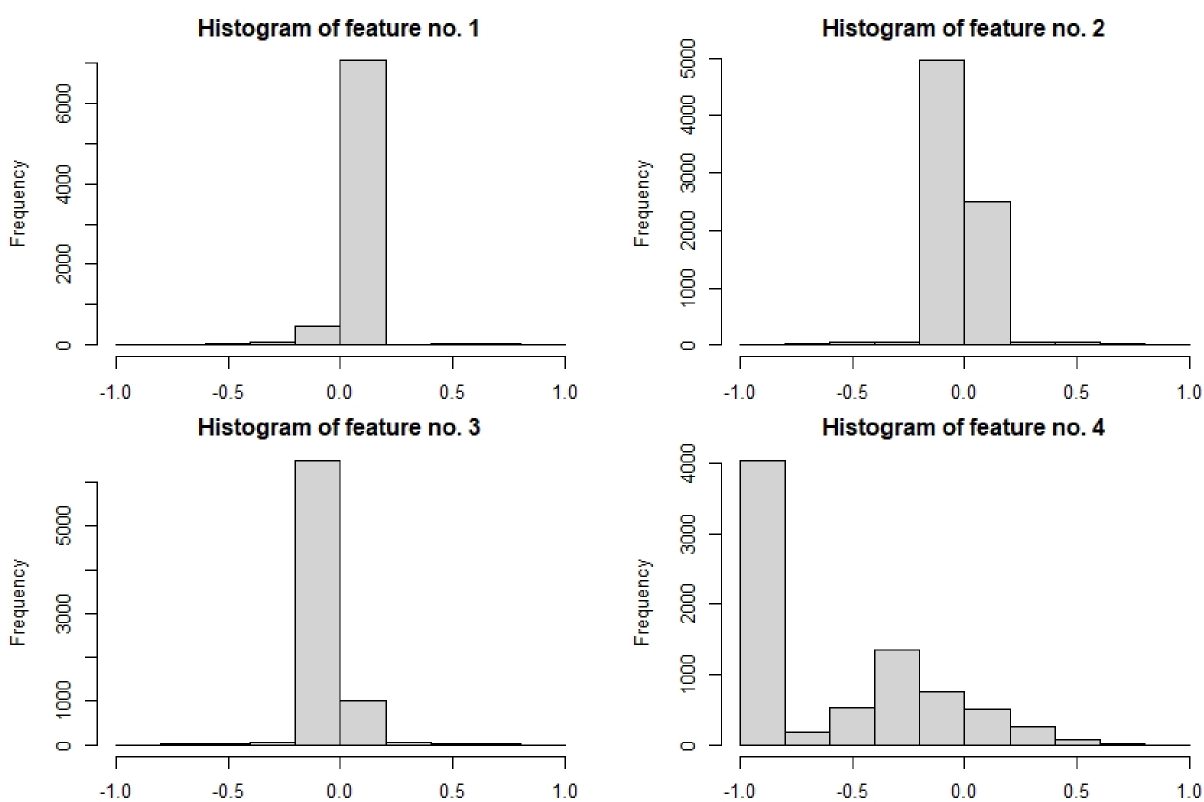

Figure 4: Histograms of the data in the first 4 feature columns.

Now, we need to prepare the data before creating the first classification model. This is done by splitting the data set into a training set and a testing set. To this end, the data was shuffled and 70% of the total number of observations were randomly sampled and stored as the training set, while the remaining 30% were stored as the testing set. After splitting, we need to ensure that the class distributions between the training set, testing set, and original full data set are all somewhat similar, so they're not overly impacted my the sampling process. This is verified by the output of the last three lines in the R code below. Also, it is worth noting that, since this process involved randomization, we should set a seed value (`set.seed(43)`) so that the results are reproducible.

```
> # Split the data set into train and test data sets
> set.seed(43)
> randomized=data[sample(1:nrow(data),nrow(data)),] # Shuffle
> tridx=sample(1:nrow(data),0.7*nrow(data),replace=F) # Get indices for 70% of the total number of samples
> trdf=randomized[tridx,] # Define training data set
> tstdf=randomized[-tridx,] # Define testing data set
> table(data$activity)/nrow(data) # Check if class distribution is similar
          1           2           3           4           5           6
0.157847303 0.138148577 0.127076091 0.166473542 0.183211021 0.181923523
          7           8           9          10          11          12
0.006051242 0.002961246 0.009656238 0.007724990 0.011587486 0.007338741

> table(trdf$activity)/nrow(trdf)
          1           2           3           4           5           6
0.153973510 0.138337013 0.125091979 0.164091244 0.185982340 0.184878587
          7           8           9          10          11          12
0.005886681 0.002575423 0.009565857 0.007542311 0.013061074 0.009013981

> table(tstdf$activity)/nrow(tstdf)
          1           2           3           4           5           6
0.166881167 0.137709138 0.131703132 0.172029172 0.176748177 0.175032175
          7           8           9          10          11          12
0.006435006 0.003861004 0.009867010 0.008151008 0.008151008 0.003432003
```

4 Random Forest Model and Variable Importance

The first objective is to obtain the ten most important features. This can be performed by training a classification model based on the random forest algorithm, using all the features in the data set, and then use that model to compute the variable importance for each feature. The `varImp()` command from `caret` package is used to obtained the variable importance from the random forest model. Fig. 5 shows the computed variable importance for the top 35 features. The features were sorted in descending order according to their variable importance value, and the top 10 features were extracted and are shown in the output of the R code below.

```
> # Create a model with random forest and compute variable importance
> require(randomForest)
> require(caret)
> formstr="activity~." # Formula argument with all features
> trdf_RF=trdf
> trdf_RF$activity=as.factor(trdf_RF$activity) # Makes it run as classification
> RF_model=randomForest(activity~.,trdf_RF) # Train the model
> RF_model_VI=varImp(RF_model) # Compute Variable Importance
> varImpPlot(RF_model,main='Variable Importance') # Plot Variable Importance
> RF_model_VI=RF_model_VI[order(RF_model_VI,decreasing=TRUE),,drop=FALSE] # Sort from highest to lowest
> top10=row.names(RF_model_VI[1:10,,drop=FALSE]) # Top ten features
> print(paste("Top ten features are =",paste(top10,collapse=", ")))

[1] "Top ten features are = X50, X42, X57, X41, X560, X53, X559, X51, X54, X58"
```

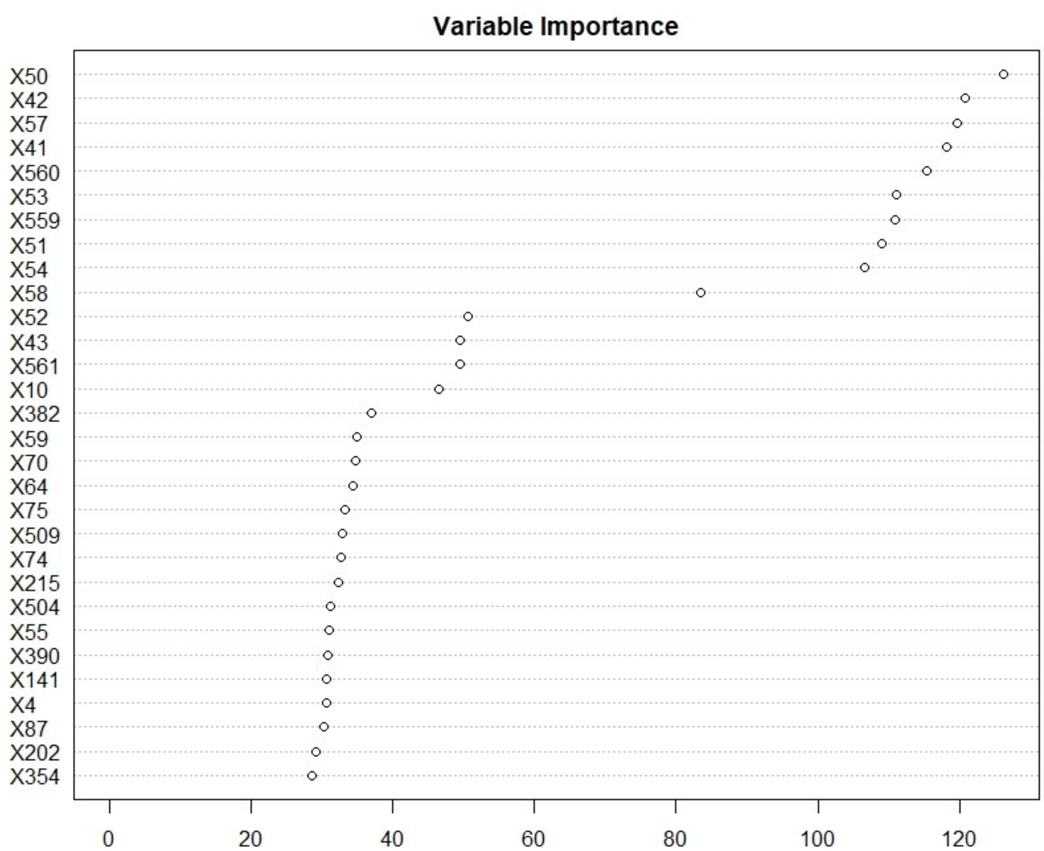

Figure 5: Variable importance for top 35 features.

5 Logistic Regression Classification Using Top 10 Features

Now, we need to generate a logistic regression classifier using the top 10 features as obtained in Section 4, and then compute its classification accuracy as a performance metric. Although the `glm()` function in R can train a logistic regression model, it is not suitable in our case since it can only handle binomial classification. Therefore, we will be using the `multinom()` function from `nnet` package, which is able to train a multinomial logistic regression classifier. The formula argument is defined such that the model will only consider the top 10 features for training. By running the `multinom()` command, we obtained the classifier and used it to perform predictions using the training set (learning phase) and testing set (generalization phase). The confusion matrix was generated and the classification accuracy was extracted in each phase, as shown in the output below. The accuracy in the learning phase was found to be 73.3%. Although the learning phase accuracy is not high enough ($< 80\%$), it does indicate that the model is capable of learning since it is better than random guessing (8.3%). The generalization phase accuracy was found to be 70.5%, which is lower by 3.8% from the learning phase accuracy. Since the drop in accuracy is not larger 25%, we can say that the model is not over-fitting.

```
> # Create a logistic regression model using the top 10 features
> require(nnet)
> formstr_top10=paste("activity~",paste(top10,collapse="+")) # Formula argument
      with top 10 features only
> mn_model1=multinom(formstr_top10,trdf,maxit=1000) # Train the model
```

```
> # Predict using the train data (Learning Phase)
> mn_pred1_tr=predict(mn_model1,trdf[,-which(names(trdf)=="activity")], type="
      class")
> mn_cfm1_tr=confusionMatrix(table(trdf[,which(names(trdf)=="activity")], mn_pred
      1_tr)) # Confusion Matrix for train data
> print("Learning Phase Confusion Matrix")
> mn_cfm1_tr
> mn_acc1_tr=round(mn_cfm1_tr$overall[['Accuracy']],4) # Accuracy of predictions
      with train data
> print(paste("Accuracy of train data predictions with top 10 features =",mn_acc1
      _tr))

```

```
[1] "Learning Phase Confusion Matrix"
Confusion Matrix and Statistics

    mn_pred1_tr
       1   2   3   4   5   6   7   8   9  10  11  12
  1  399  80  61  31 266   0   0   0   0   0   0   0
  2   73 487 132  23  35   0   0   0   0   0   2   0
  3  134  66 435  33  12   0   0   0   0   0   0   0
  4    5   3  12 722 136   0   0   4   8   0   2   0
  5   42  41  18  86 820   0   1   1   1   0   1   0
  6    0   0   0   0   0 994   0   0   0   6   0   5
  7    0  12   8   1   0   0   8   0   0   0   3   0
  8    0   0   0   2   0   0   2   9   0   0   1   0
  9    0   0   1   7   0   0   1   0  29   0   7   7
  10   0   0   0   0   0   4   0   0   4  19   2  12
  11   0   7   1   4   0   1   2   3   6   2  43   2
  12   0   0   0   0   0  11   0   0   4  11   2  21

Overall Statistics

               Accuracy : 0.7333
                 95% CI : (0.7213, 0.745)
    No Information Rate : 0.2334
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.6835

[1] "Accuracy of train data predictions with top 10 features = 0.7333"
```

```
> # Predict using the test data (Generalization Phase)
> mn_pred1_tst=predict(mn_model1,tstdf[,-which(names(tstdf)=="activity")], type="
        class")
> mn_cfm1_tst=confusionMatrix(table(tstdf[,which(names(tstdf)=="activity")],
        mn_pred1_tst)) # Confusion Matrix for test data
> print("Generalization Phase Confusion Matrix")
> mn_cfm1_tst
> mn_acc1_tst=round(mn_cfm1_tst$overall[['Accuracy']],4) # Accuracy of
        predictions with test data
> print(paste("Accuracy of test data predictions with top 10 features =",mn_acc1
        _tst))

```

```
[1] "Generalization Phase Confusion Matrix"
Confusion Matrix and Statistics

   mn_pred1_tst
      1   2   3   4   5   6   7   8   9  10  11  12
  1 187  42  26  16 118   0   0   0   0   0   0   0
  2  30 200  72   8  10   0   0   0   0   0   1   0
  3  64  41 177  19   6   0   0   0   0   0   0   0
  4   3   1   7 300  84   0   0   0   4   0   2   0
  5  20  18   6  36 330   0   1   0   0   0   1   0
  6   0   0   0   0   0 405   0   0   0   1   1   1
  7   0   2   6   1   0   0   6   0   0   0   0   0
  8   0   0   0   2   0   0   1   4   0   0   2   0
  9   0   0   2   1   0   0   0   0  14   1   4   1
  10  0   0   0   0   0   6   0   0   0   5   2   6
  11  0   2   1   0   0   0   0   1   3   0  11   1
  12  0   0   0   0   0   1   0   0   1   1   0   5

Overall Statistics

               Accuracy : 0.7053
                 95% CI : (0.6863, 0.7237)
    No Information Rate : 0.2351
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.6501

[1] "Accuracy of test data predictions with top 10 features= 0.7053"
```

```
> # Check for over-fitting. Criteria: Accuracy change from train to test > 25%
> mn_model1_isOF=abs((mn_acc1_tr-mn_acc1_tst)/mn_acc1_tr)
> mn_model1_isOF=round(mn_model1_isOF,4)
> print(paste("Accuracy drop from training data to test data is",mn_model1_isOF*100,"%"))
> if(mn_model1_isOF>0.25) print("Model is over-fitting") else print("Model is not over-fitting")

[1] "Accuracy drop from training data to test data is 3.82 %"
[1] "Model is not over-fitting"
```

6 Null and Residual Deviance

Deviance is a measure of goodness of fit of a model. A high value indicates a bad fit. Before extracting the residual deviance of the previously generated model, we need to obtain the null deviance since it is not returned by the `multinom` model summary. This is done by creating a null model with no features. The deviance of the null model would be the null deviance. The output of the below R code shows the null and residual deviance for the logistic regression classifier with top 10 features.

```
> # Obtaining the null/residual deviance
> mn_model_null=multinom(activity~1,trdf,maxit=1000) # Create the null model
> mn_summary_null=summary(mn_model_null) # Summary of null model
> mn_nulldeviance=round(mn_summary_null$deviance,2) # Obtain the null deviance
> print(paste("Null Deviance =",mn_nulldeviance))
> mn_summary1=summary(mn_model1) # Get the summary of the top 10 features model
> mn_resdeviance1=round(mn_summary1$deviance,2) # Obtain the residual deviance
> print(paste("Residual Deviance with top 10 Features =",mn_resdeviance1))

[1] "Null Deviance = 21409.92"
[1] "Residual Deviance with top 10 Features = 8344.26"
```

7 Eliminating Correlated Features

In Section 3, we identified 516 correlated features in our data set. Eliminating those features would greatly reduce the number of variables and model complexity without much compromise on the goodness of fit. All feature pairs with a correlation coefficient larger than 0.5 were considered to be correlated, and one of the features in that pair was eliminated. This process left us with 46 features out of the total 561 features in the data set. Then, we again checked the correlation matrix to ensure that the remaining 46 features are uncorrelated.

```
> # Eliminate correlated predictors
> cor_index=which(abs(cordata)>0.5 & abs(cordata)!=1, arr.ind = T) # Get the
        indeces where the correlation coefficient >0.5 in the correlation matrix
> cor_index=cor_index[!duplicated(cbind(pmax(cor_index[,1], cor_index[,2]), pmin(
        cor_index[,1], cor_index[,2]))),] # Retain one of the two predictors
> tbl_cor_index=table(cor_index[,1])
> cor_attributes=as.numeric(names(tbl_cor_index)) # Correlated attributes
> trdf_uncor=trdf[,-cor_attributes] # Remove correlated predictors from train set
> tstdf_uncor=tstdf[,-cor_attributes]# Remove correlated predictors from test set
> cordata2=cor(trdf_uncor[,-which(names(trdf_uncor)=="activity")])
> print(ifelse(any(abs(cordata2[cordata2!=1])>0.5 ), "Correlated Predictors Exist
        ", "No Correlated Predictors"))
[1] "No Correlated Predictors"
```

After eliminating correlated features, we examine if multicollinearily exists between the remaining features by computing their Variance Inflation Factor (VIF). This can be performed using the `vif()` function from the `car` package in R. However, this function requires a logsitc regressing model generated by `glm()`. As stated previously, using `glm()` directly is not suitable for our data since it is not binomial; therefore, we employ a one-vs-all approach for each class. Accordingly, we will have 12 glm models, and 12 sets of VIFs, each corresponding to a specific class, as can be seen in the output below.
The glm model for classes 6, 7 and 8 did not converge; hence, their VIF values are unrealistically high. Looking at the VIF values of feature X41 in classes 1, 3 and 5, we find that they are considered to be high (larger than 5), which implies that this feature introduces collinearity for those specific classes. However, the VIF value for that same feature is low in other classes, such as class 2, 4, 9 and 11. Therefore, it is not safe to assume that feature X41 can be eliminated due to multicollinearity. The way of aggregating VIFs for multiple classes is still not clear at this point, so we are not able to determine which features exhibit multicollinearity.

```
> for(i in 1:12){ # Loop over all classes
+   class_idx=which(trdf_uncor$activity==i) # Get the indeces of containing the class i
+   vif_df=trdf_uncor # Make a copy of the training set
+   vif_df$activity[class_idx]=1 # Set class i to 1
+   vif_df$activity[-class_idx]=0 # Set other classes to 0
+   bi_model=glm(activity~., vif_df, family="binomial") # Train a binomial glm model
+   vifs=vif(bi_model) # Compute VIFs for class i
+   print(paste("VIFs for class", i))
+   print(vifs)
+ }

[1] "VIFs for class 1"
      X1       X4      X38      X39      X40      X41      X56      X70
3.098208 2.851807 3.400710 2.498406 2.530201 6.312181 6.383775 2.251108
     X78      X79      X81      X82      X83     X109     X113     X117
1.251561 1.298808 1.158059 1.166577 1.137122 1.548071 1.431990 1.389312
    X118     X119     X120     X121     X123     X158     X159     X160
2.585682 2.302533 2.960153 1.410192 1.368086 2.242708 1.949679 2.578537
    X161     X162     X163     X189     X193     X197     X198     X199
1.130071 1.125181 1.101199 1.288766 1.371259 1.375306 2.193864 1.713175
    X200     X238     X264     X293     X371     X372     X449     X455
2.089800 1.154919 1.384957 1.277993 1.412890 1.381042 1.365529 1.616249
    X459     X525     X538     X551     X555
1.476239 1.133926 1.466514 1.098580 3.088010
```

```
[1] "VIFs for class 2"
      X1       X4      X38      X39      X40      X41      X56      X70 
1.508995 2.775926 2.311853 2.228741 2.228090 3.526166 3.873505 2.902383 
     X78      X79      X81      X82      X83     X109     X113     X117 
1.440569 1.274553 1.265775 1.264933 1.162891 1.315260 1.474532 1.300563 
    X118     X119     X120     X121     X123     X158     X159     X160 
2.246913 1.899664 2.077941 1.340376 1.589314 1.831190 1.568546 1.942967 
    X161     X162     X163     X189     X193     X197     X198     X199 
1.167426 1.127250 1.068052 1.325442 1.382098 1.389145 1.780115 1.709959 
    X200     X238     X264     X293     X371     X372     X449     X455 
1.714651 1.239904 1.157375 1.802002 1.289019 1.354888 1.316953 1.507006 
    X459     X525     X538     X551     X555 
1.893769 1.121596 1.282707 1.125824 1.271238 

[1] "VIFs for class 3"
      X1       X4      X38      X39      X40      X41      X56      X70 
3.197803 3.163519 5.892273 4.074426 6.033498 5.243117 2.624277 2.494871 
     X78      X79      X81      X82      X83     X109     X113     X117 
1.528612 1.384229 1.426188 1.419619 1.187775 1.403864 1.326794 1.511904 
    X118     X119     X120     X121     X123     X158     X159     X160 
3.843335 3.107188 3.857123 1.309814 1.404070 2.199160 1.846400 1.923367 
    X161     X162     X163     X189     X193     X197     X198     X199 
1.238444 1.281203 1.139084 1.326498 1.600055 1.622519 2.429850 2.954249 
    X200     X238     X264     X293     X371     X372     X449     X455 
2.376270 1.168124 1.658927 1.473465 1.588861 1.778404 1.437463 1.800269 
    X459     X525     X538     X551     X555 
2.289268 1.738784 1.565326 1.282566 1.518434 

[1] "VIFs for class 4"
      X1       X4      X38      X39      X40      X41      X56      X70 
1.242305 2.060340 2.338110 2.423989 1.737524 1.636163 1.679673 2.115464 
     X78      X79      X81      X82      X83     X109     X113     X117 
1.852146 1.970812 1.449579 1.688900 1.437285 1.215422 1.367005 1.331116 
    X118     X119     X120     X121     X123     X158     X159     X160 
1.657477 1.515006 1.692224 1.866466 2.006416 1.751115 1.813838 2.094575 
    X161     X162     X163     X189     X193     X197     X198     X199 
1.239992 1.231104 1.347175 1.219874 1.299089 1.333628 1.697989 1.654711 
    X200     X238     X264     X293     X371     X372     X449     X455 
1.986268 1.073119 1.082173 1.151606 1.258193 1.281281 1.329734 1.662460 
    X459     X525     X538     X551     X555 
1.589748 1.096493 1.099045 1.095673 1.046986 
```

```
[1] "VIFs for class 5"
      X1       X4      X38      X39      X40      X41      X56      X70 
1.348955 2.136724 2.470447 2.614905 1.844320 8.776145 8.791091 2.047321 
     X78      X79      X81      X82      X83     X109     X113     X117 
2.001416 2.075944 1.377120 1.476964 1.342536 1.201662 1.291318 1.281900 
    X118     X119     X120     X121     X123     X158     X159     X160 
1.698694 1.587817 1.841608 1.631727 1.615167 1.805812 2.100549 2.308834 
    X161     X162     X163     X189     X193     X197     X198     X199 
1.135650 1.354483 1.313808 1.199992 1.286884 1.315696 1.770069 1.834057 
    X200     X238     X264     X293     X371     X372     X449     X455 
2.125129 1.091365 1.087756 1.156486 1.293846 1.281544 1.307419 1.680455 
    X459     X525     X538     X551     X555 
1.595824 1.091804 1.116073 1.109597 1.093683 

[1] "VIFs for class 6"
        X1         X4        X38        X39        X40        X41        X56 
 42.855524  63.364793 125.169583  45.882620  50.781249  30.492810  22.450327 
       X70        X78        X79        X81        X82        X83       X109 
 46.825818  33.324388  28.586739  46.463162  31.653346  22.274391  34.813819 
      X113       X117       X118       X119       X120       X121       X123 
 19.535602  28.472943  39.410330  18.021598  87.117388  27.004923  45.886503 
      X158       X159       X160       X161       X162       X163       X189 
 39.991379  13.486358  46.254008  14.506555  44.412735  50.092684  29.738182 
      X193       X197       X198       X199       X200       X238       X264 
 18.123646  28.929756  14.077222  49.631339 100.748304  14.893845  18.278545 
      X293       X371       X372       X449       X455       X459       X525 
  7.112223  59.680747  13.854508  59.449782  45.064500  34.073637  12.019765 
      X538       X551       X555 
 25.336056  14.602903  17.615826 

[1] "VIFs for class 7"
        X1         X4        X38        X39        X40        X41        X56 
 31.868177 177.166625  11.200648  24.476180  66.891437  75.348373  91.972037 
       X70        X78        X79        X81        X82        X83       X109 
 37.434598  16.520406  34.939562  59.713273 124.091446 157.627019  13.879702 
      X113       X117       X118       X119       X120       X121       X123 
  9.839189  10.126926  18.686022  11.146988   7.841169  51.729168  48.735080 
      X158       X159       X160       X161       X162       X163       X189 
 85.843175  23.136353  28.950654  89.197371   6.862271  14.192065  21.053913 
      X193       X197       X198       X199       X200       X238       X264 
  8.128018  21.513301   6.915824  47.481798  12.653119  20.277080  10.861959 
      X293       X371       X372       X449       X455       X459       X525 
 78.881530   4.580368  53.220159   8.676791   4.589110  35.841888  50.066589 
      X538       X551       X555 
 14.422434   2.887353   6.945561 
```

```
[1] "VIFs for class 8"
        X1         X4        X38        X39        X40        X41        X56 
 27.014409  98.810205  56.819967  97.742047  80.593316  77.864826  17.942124 
       X70        X78        X79        X81        X82        X83       X109 
106.048513  17.882086  36.785059  41.419699 109.140793  46.194450  23.017802 
      X113       X117       X118       X119       X120       X121       X123 
 55.582809 109.438277  80.010039  49.557020  85.066878  69.021493  71.171350 
      X158       X159       X160       X161       X162       X163       X189 
 39.914155  49.006268 103.797623  20.654276  70.514073  27.002994  40.543732 
      X193       X197       X198       X199       X200       X238       X264 
 56.096445  23.143960  58.168688  67.916469  21.668761  35.160758  26.544367 
      X293       X371       X372       X449       X455       X459       X525 
 57.375771  44.072659  35.513751  38.275885  27.207302  37.743787   8.584713 
      X538       X551       X555 
 53.587029   6.724509  18.030759 

[1] "VIFs for class 9"
      X1       X4      X38      X39      X40      X41      X56      X70 
6.948660 3.537402 2.900029 3.420179 4.023964 2.290510 2.370297 3.017226 
     X78      X79      X81      X82      X83     X109     X113     X117 
2.358295 2.266026 2.667164 4.143646 1.829464 1.693695 2.368648 2.031817 
    X118     X119     X120     X121     X123     X158     X159     X160 
1.857989 2.198627 1.781480 2.294575 4.852022 2.145215 2.154618 2.202328 
    X161     X162     X163     X189     X193     X197     X198     X199 
1.910851 1.810909 2.381149 2.049078 1.929765 1.772766 1.971122 1.659192 
    X200     X238     X264     X293     X371     X372     X449     X455 
1.816320 1.851859 1.829741 2.130826 1.674056 1.686078 1.534783 1.988021 
    X459     X525     X538     X551     X555 
1.923318 1.546294 2.148793 1.446126 2.439367 

[1] "VIFs for class 10"
       X1        X4       X38       X39       X40       X41       X56 
41.119522  5.047992  5.441009  8.698068 10.897700  8.434156  8.658131 
      X70       X78       X79       X81       X82       X83      X109 
 5.298329  3.656197  5.441499  8.801977 14.764426  5.966306  7.448855 
     X113      X117      X118      X119      X120      X121      X123 
 5.919016  4.123717  5.709605  4.159300  3.610440  8.415053 14.596603 
     X158      X159      X160      X161      X162      X163      X189 
 3.171314  3.190870  5.292425  3.758870  3.434037  4.750743  4.135214 
     X193      X197      X198      X199      X200      X238      X264 
 5.025502  3.607971  5.664477  3.319357  4.562960  2.690820  2.856199 
     X293      X371      X372      X449      X455      X459      X525 
 2.976780  4.801040  3.994265  3.342047  3.883703  3.859752  3.848630 
     X538      X551      X555 
 3.243979  2.786207  4.630591 
```

```

[1] "VIFs for class 11"
      X1       X4      X38      X39      X40      X41      X56      X70 
4.439258 3.868403 3.562122 2.507171 2.224752 2.660717 2.741601 2.777117 
     X78      X79      X81      X82      X83     X109     X113     X117 
4.261607 2.406433 2.044543 3.795130 1.848667 1.775108 2.117882 1.959908 
    X118     X119     X120     X121     X123     X158     X159     X160 
1.815609 1.655642 1.609147 2.056802 3.938536 1.694040 1.734767 2.271342 
    X161     X162     X163     X189     X193     X197     X198     X199 
2.046318 1.502938 1.758021 1.680993 1.814053 1.828079 1.773085 1.551850 
    X200     X238     X264     X293     X371     X372     X449     X455 
1.841509 1.489378 1.428399 1.606238 1.422491 1.468682 1.308116 1.779544 
    X459     X525     X538     X551     X555 
1.855883 1.276256 1.436676 1.554791 1.800413 

[1] "VIFs for class 12"
       X1        X4       X38       X39       X40       X41       X56 
 9.424558  4.998048  6.333884  6.988327 10.844274  4.296336  2.693521 
      X70       X78       X79       X81       X82       X83      X109 
 2.670771  2.740794  3.676265  7.841237  4.741249  7.139081  2.978857 
     X113      X117      X118      X119      X120      X121      X123 
 2.213461  3.388556  2.173051  2.081529  2.926680  3.001478  5.728117 
     X158      X159      X160      X161      X162      X163      X189 
 2.702013  2.177165  3.279492  2.536258  1.856380  3.151271  2.122427 
     X193      X197      X198      X199      X200      X238      X264 
 1.988733  2.259901  2.043928  3.062653  4.949597  2.464531  1.705153 
     X293      X371      X372      X449      X455      X459      X525 
 3.210605  2.653512  2.240503  2.053220  2.091338  2.791467  1.664558 
     X538      X551      X555 
 1.591308  1.476650  2.402801 
```

8 Logistic Regression Model Using Uncorrelated Features

Using the uncorrelated features set, a logistic regression classifier was trained, and the classification accuracy was obtained for the learning and generalization phases, which were found to be 93.5% and 89%, respectively. The drop in accuracy from learning phase to generalization phase was 4.9%; hence, the model is not over-fitting. The residual deviance for this model is 1625.5, which is considerably lower than the one evaluated from the model with top 10 features in section 5, which was 8344.3. Moreover, comparing these two models, there is a significant improvement in the classification accuracy in this model that is trained with uncorrelated features.

```
> # Create a logistic regression model using uncorrelated features
> mn_model2=multinom(formstr,trdf_uncor,maxit=1000) # Train the model
> # Obtaining the residual deviance
> mn_summary2=summary(mn_model2) # Get the summary of model parameters
> mn_resdeviance2=round(mn_summary2$deviance,2) # Obtain the residual deviance
> print(paste("Residual Deviance with Uncorrelated Features =",mn_resdeviance2))

[1] "Residual Deviance with Uncorrelated Features = 1625.51"
```

```
> # Predict using the train data (Learning Phase)
> mn_pred2_tr=predict(mn_model2,trdf_uncor[, -which(names(trdf_uncor)=="activity"
        )], type="class")
> mn_cfm2_tr=confusionMatrix(table(trdf_uncor[, which(names(trdf_uncor)=="
        activity")], mn_pred2_tr)) # Confusion Matrix for train data
> print("Learning Phase Confusion Matrix")
> mn_cfm2_tr
> mn_acc2_tr=round(mn_cfm2_tr$overall[['Accuracy']],4) # Accuracy of predictions
        with train data
> print(paste("Classification accuracy of learning phase using uncorrelated
        features =",mn_acc2_tr))

[1] "Learning Phase Confusion Matrix"
Confusion Matrix and Statistics

   mn_pred2_tr
       1    2    3    4    5    6    7    8    9   10   11   12
  1  824    6    7    0    0    0    0    0    0    0    0    0
  2    8  730   14    0    0    0    0    0    0    0    0    0
  3    4   13  663    0    0    0    0    0    0    0    0    0
  4    0    0    0  725  167    0    0    0    0    0    0    0
  5    0    0    0  133  878    0    0    0    0    0    0    0
  6    0    0    0    0    0 1005    0    0    0    0    0    0
  7    0    0    0    0    0    0   32    0    0    0    0    0
  8    0    0    0    0    0    0    0   14    0    0    0    0
  9    0    0    0    0    0    0    0    0   52    0    0    0
  10   0    0    0    0    0    0    0    0    0   41    0    0
  11   0    0    0    0    0    0    0    0    0    0   71    0
  12   0    0    0    0    0    0    0    0    0    0    0   49

Overall Statistics

               Accuracy : 0.9352
                 95% CI : (0.9284, 0.9416)
    No Information Rate : 0.1922
```

```
   P-Value [Acc > NIR] : < 2.2e-16

                 Kappa : 0.9234

[1] "Classification accuracy of learning phase using uncorrelated features = 0.9352"
```

```
> # Predict using the test data (Generalization Phase)
> mn_pred2_tst=predict(mn_model2,tstdf_uncor[, -which(names(tstdf_uncor)=="activity")], type="class")
> mn_cfm2_tst=confusionMatrix(table(tstdf_uncor[, which(names(tstdf_uncor)=="activity")], mn_pred2_tst)) # Confusion Matrix for test data
> print("Generalization Phase Confusion Matrix")
> mn_cfm2_tst
> mn_acc2_tst=round(mn_cfm2_tst$overall[['Accuracy']],4) # Accuracy of predictions with test data
> print(paste("Classification accuracy of generalization phase using uncorrelated features =",mn_acc2_tst))

[1] "Generalization Phase Confusion Matrix"
Confusion Matrix and Statistics

    mn_pred2_tst
       1   2   3   4   5   6   7   8   9  10  11  12
  1  377   3   5   0   1   0   0   1   1   0   1   0
  2    6 299   7   0   0   0   0   0   2   0   7   0
  3    5  14 287   0   0   0   0   0   0   1   0   0
  4    0   0   0 306  92   0   2   0   1   0   0   0
  5    1   0   0  63 346   0   0   0   1   0   1   0
  6    0   0   0   0   0 401   2   0   0   1   4   0
  7    0   0   0   1   1   0  10   3   0   0   0   0
  8    0   0   0   3   0   0   0   3   0   2   0   1
  9    0   0   0   1   0   0   0   1  16   0   5   0
  10   0   0   0   0   0   0   0   0   0  14   0   5
  11   0   1   0   2   0   0   0   0   8   0   8   0
  12   0   0   0   0   0   0   0   0   0   1   0   7

Overall Statistics

               Accuracy : 0.8897
                 95% CI : (0.8763, 0.9022)
    No Information Rate : 0.1888
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8695
```

```
[1] "Classification accuracy of generalization phase using uncorrelated features
        = 0.8897"
```

```
> # Check for over-fitting. Criteria: Accuracy change from train to test > 25%
> mn_model2_isOF=abs((mn_acc2_tr-mn_acc2_tst)/mn_acc2_tr)
> mn_model2_isOF=round(mn_model2_isOF,4)
> print(paste("Accuracy drop from training data to test data is",mn_model2_isOF*1
        00,"%"))
> if(mn_model2_isOF>0.25) print("Model is over-fitting") else print("Model is not
         over-fitting")

[1] "Accuracy drop from training data to test data is 4.87 %"
[1] "Model is not over-fitting"
```

9 Logistic Regression Model Performance Metrics

As previously described, the model trained with uncorrelated features has given generally better results than the one trained with the top 10 features from `varImp()`. The performance metric used to compare between these two models was the overall classification accuracy. Here, we will extract other performance metrics, including sensitivity, precision, specificity, recall and balanced accuracy, for each class of of the logistic regression model with uncorrelated features. These are obtained from the statistics of the confusion matrices. Subsequently, macro averaging is performed to get the overall metric for all classes, which is computed by simply taking the mean over all classes, as follows:

$$P_{macro} = \frac{1}{N}\sum_{i=1}^{N} P_i$$

where, P is the performance metric, i is an index indicating class number, and N is the total number of classes, which is 12 in our case. The results for learning and generalization phases are shown in the output of the below R code.

Furthermore, another performance measurement, AUC, was obtained using the `pROC` package in R. This package contains the `multiclass.roc()` function, which computes the ROC curves and AUC for multi-class classification models as defined by *David J. Hand & Robert J. Till.* Since this method implements a One-vs-One approach in generating the ROC curves, we will have 66 curves for our 12 classes. Figures 6 and 7 show the ROC curves for learning and generalization phases, respectively. The plots present ROC curves for 7 different class combinations out of the available 66, and the overall AUCs are provided in the output of the below code.

```
> # Logistic Regression Performance Parameters
> require(pROC)
> # Learning Phase
> mn_PM2_tr=mn_cfm2_tr$byClass[, c("Balanced Accuracy", "Precision", "Sensitivity", "Specificity", "Recall")]
> print("Logistic-Regression Learning-Phase Performance Parameters:")
> mn_PM2_tr
> mn_PMavg2_tr=round(apply(mn_PM2_tr,2,mean),4)
> print("Macro Averages:")
> t(mn_PMavg2_tr)
> mn_prob2_tr=predict(mn_model2, trdf_uncor[,-which(names(trdf_uncor)=="activity")], type="prob")
> mn_AUC2_tr=multiclass.roc(trdf_uncor[,which(names(trdf_uncor)=="activity")], mn_prob2_tr)
> print(paste("Logistic-Regression Learning-Phase AUC:",round(mn_AUC2_tr$auc,4)))
> # ROC curves
> mn_ROC2_tr=mn_AUC2_tr$rocs
> ROC_num=paste("1/",as.character(2),sep="")
> plot.roc(mn_ROC2_tr[[ROC_num]][[2]], col=2, main="ROC curves of 7 One-vs-One class combinations (Learning Phase)")
> for(i in 3:8) {ROC_num=paste("1/",as.character(i),sep="")
+ lines.roc(mn_ROC2_tr[[ROC_num]][[2]],col=i)}
> legend("bottom", legend=c(’1/2’,’1/3’,’1/4’,’1/5’,’1/6’,’1/7’,’1/8’), col=2:8, lwd=2)

[1] "Logistic-Regression Learning-Phase Performance Parameters:"
          Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1         0.9914099 0.9844683 0.9856459  0.9971739 0.9856459
Class: 2         0.9849695 0.9707447 0.9746328  0.9953062 0.9746328
Class: 3         0.9828604 0.9750000 0.9692982  0.9964226 0.9692982
Class: 4         0.9042548 0.8127803 0.8449883  0.9635212 0.8449883
Class: 5         0.9049511 0.8684471 0.8401914  0.9697108 0.8401914
Class: 6         1.0000000 1.0000000 1.0000000  1.0000000 1.0000000
Class: 7         1.0000000 1.0000000 1.0000000  1.0000000 1.0000000
Class: 8         1.0000000 1.0000000 1.0000000  1.0000000 1.0000000
Class: 9         1.0000000 1.0000000 1.0000000  1.0000000 1.0000000
Class: 10        1.0000000 1.0000000 1.0000000  1.0000000 1.0000000
Class: 11        1.0000000 1.0000000 1.0000000  1.0000000 1.0000000
Class: 12        1.0000000 1.0000000 1.0000000  1.0000000 1.0000000
[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]            0.9807    0.9676      0.9679      0.9935 0.9679

[1] "Logistic-Regression Learning-Phase AUC: 0.9987"
```

```
> # Generalization Phase
> mn_PM2_tst=mn_cfm2_tst$byClass[, c("Balanced Accuracy", "Precision", "Sensitivity", "Specificity", "Recall")]
> print("Logistic-Regression Generalization-Phase Performance Parameters:")
> mn_PM2_tst
> mn_PMavg2_tst=round(apply(mn_PM2_tst,2,mean),4)
> print("Macro Averages:")
> t(mn_PMavg2_tst)
> mn_prob2_tst=predict(mn_model2, tstdf_uncor[,-which(names(tstdf_uncor)=="activity")], type="prob")
> mn_AUC2_tst=multiclass.roc(tstdf_uncor[, which(names(tstdf_uncor)=="activity")], mn_prob2_tst)
> print(paste("Logistic-Regression Generalization-Phase AUC:", round(mn_AUC2_tst$auc,4)))
> # ROC curves
> mn_ROC2_tst=mn_AUC2_tst$rocs
> ROC_num=paste("1/",as.character(2),sep="")
> plot.roc(mn_ROC2_tst[[ROC_num]][[2]], col=2, main="ROC curves of 7 One-vs-One class combinations (Generalization Phase)")
> for(i in 3:8) {ROC_num=paste("1/",as.character(i),sep="")
+ lines.roc(mn_ROC2_tst[[ROC_num]][[2]],col=i)}
> legend("bottom", legend=c('1/2','1/3','1/4','1/5','1/6','1/7','1/8'), col=2:8, lwd=2)

[1] "Logistic-Regression Generalization-Phase Performance Parameters:"
          Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1          0.9814862 0.9691517 0.9691517   0.9938208 0.9691517
Class: 2          0.9661471 0.9314642 0.9432177   0.9890765 0.9432177
Class: 3          0.9750119 0.9348534 0.9598662   0.9901575 0.9598662
Class: 4          0.8826182 0.7630923 0.8138298   0.9514066 0.8138298
Class: 5          0.8757307 0.8398058 0.7863636   0.9650978 0.7863636
Class: 6          0.9981865 0.9828431 1.0000000   0.9963731 1.0000000
Class: 7          0.8560639 0.6666667 0.7142857   0.9978420 0.7142857
Class: 8          0.6862086 0.3333333 0.3750000   0.9974171 0.3750000
Class: 9          0.7743417 0.6956522 0.5517241   0.9969592 0.5517241
Class: 10         0.8673397 0.7368421 0.7368421   0.9978374 0.7368421
Class: 11         0.6514600 0.4210526 0.3076923   0.9952278 0.3076923
Class: 12         0.7690151 0.8750000 0.5384615   0.9995686 0.5384615

[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]             0.857    0.7625      0.7247      0.9892 0.7247

[1] "Logistic-Regression Generalization-Phase AUC: 0.9748"
```

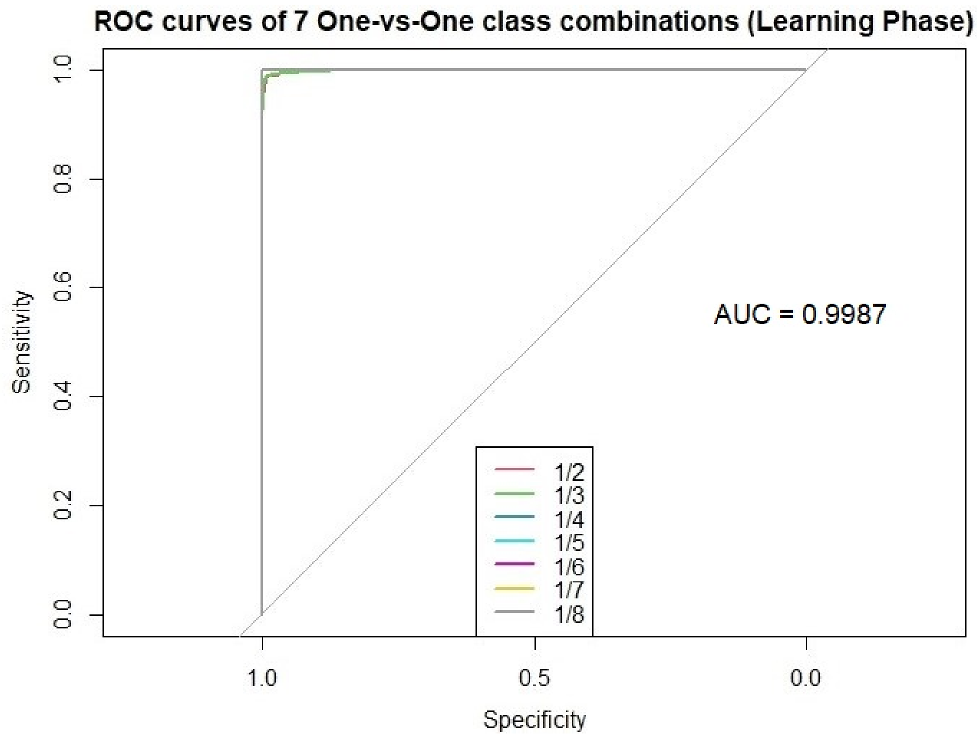

Figure 6: ROC curves for the learning phase of the logistic regression model.

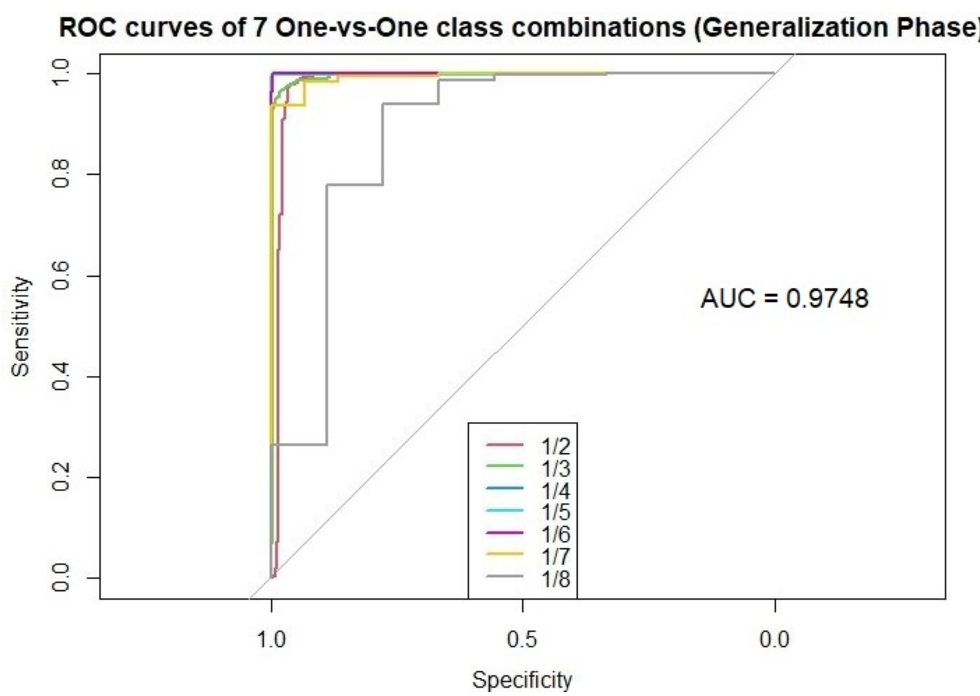

Figure 7: ROC curves for the generalization phase of logistic regression model.

A variance estimation function was created to estimate the variance in the model using 30%, 60% and 100% of the data. The process includes partitioning the data into two subsets, one includes 90% and the other includes 10% of the total number of samples. The 10% subset is reserved for testing, while the 90% subset is placed inside a loop where, in each iteration, a percentage of samples are drawn and used to train a model, and the classification accuracy is computed using the reserved 10% subset. The variance of the accuracies obtained in each iteration would represent the variance estimation for the model. Since we will be using the same function in the subsequent sections, it includes three different model types; Logistic Regression, Naive Bayes, and kNN. The results of variance estimation for the logistic regression model, using 30%, 60% and 100% of the data, are shown in the output below.

```
> # Variance Estimation
> varEst_tridx=sample(1:nrow(trdf_uncor), 0.9*nrow(trdf_uncor), replace=F) # Get indices for 90% of the total number of samples
> varEst_trdf=trdf_uncor[varEst_tridx,] # Define training data for variance estimation
> varEst_tstdf=trdf_uncor[-varEst_tridx,] # Define variance estimation partition
>
> varEst=function(trdf,tstdf,percent,type){
+   target_idx=which(names(trdf)=="activity")
+   acc_varEstp=c(); # Initialize a variable to store the accuracies computed in the loop
+   for(i in 1:100){
+     varEstp_tridx=sample(1:nrow(trdf), percent/100*nrow(trdf), replace=F) # Take samples, percent% of the data
+     varEstp_trdf=trdf[varEstp_tridx,]
+     if(type=="multinom"){
+       mn_model_varEstp=multinom(activity~., varEstp_trdf, maxit=1000, trace=F) # Train a multinomial model
+       pred_varEstp=predict(mn_model_varEstp, tstdf[,-target_idx], type="class") # Predict with variance estimation partition
+     }else if(type=="nb"){
+       varEstp_trdf$activity=as.factor(varEstp_trdf$activity)
+       nb_model_varEstp=naiveBayes(activity~.,data=varEstp_trdf) # Train a naive bayes model
+       pred_varEstp=predict(nb_model_varEstp, tstdf[,-target_idx], type="class") # Predict with variance estimation partition
+     }else if(type=="knn"){
+       trclass=factor(varEstp_trdf[,target_idx])
+       tstclass=factor(tstdf[,target_idx])
+       pred_varEstp=knn(varEstp_trdf[,-target_idx], tstdf[,-target_idx], trclass, k = 15, prob=TRUE)
+     }else{
+       print("type should be ’multinom’ ’nb’ or ’knn’")
+       return()
```

```
+     }
+     u_varEstp=union(pred_varEstp, tstdf[,target_idx]) # Avoids issues when
        number of classes are not equal
+     t_varEstp=table(factor(pred_varEstp, u_varEstp), factor(tstdf[,target_idx],
        u_varEstp))
+     mn_cfm_varEstp=confusionMatrix(t_varEstp) # Confusion Matrix
+     mn_acc_varEstp=mn_cfm_varEstp$overall[['Accuracy']] # Accuracy of
        predictions
+     acc_varEstp=c(acc_varEstp,mn_acc_varEstp) # Store
+   }
+   mean_varEstp=signif(mean(acc_varEstp),4)
+   var_varEstp=signif(var(acc_varEstp),4)
+   varEstp=data.frame(mean_varEstp,var_varEstp)
+   names(varEstp)=c("Mean of Accuracies","Variance of Accuracies")
+   return(t(varEstp))
+ }
>
> mn_varEst30=varEst(varEst_trdf, varEst_tstdf, 30, type="multinom") # Variance
        estimation using 30% of the data
> mn_varEst60=varEst(varEst_trdf, varEst_tstdf, 60, type="multinom") # Variance
        estimation using 60% of the data
> mn_varEst100=varEst(varEst_trdf, varEst_tstdf, 100, type="multinom") # Variance
         estimation using 100% of the data
>
> print("Logistic-Regression Variance Estimation using 30% of data:")
> mn_varEst30
> print("Logistic-Regression Variance Estimation using 60% of data:")
> mn_varEst60
> print("Logistic-Regression Variance Estimation using 100% of data:")
> mn_varEst100

[1] "Logistic-Regression Variance Estimation using 30% of data:"
Mean of Accuracies   0.83420
Variance of Accuracies 0.00016

[1] "Logistic-Regression Variance Estimation using 60% of data:"
Mean of Accuracies   0.8561000
Variance of Accuracies 0.0001308

[1] "Logistic-Regression Variance Estimation using 100% of data:"
Mean of Accuracies   0.8842
Variance of Accuracies 0.0000
```

10 Naive Bayes Classifier

Using again the uncorrelated features set, a model is trained based on the Naive Bayes algorithm. The `naiveBayes()` function is used from the `e1071` package, which can generate a multi-class Naive Bayes classifier. The confusion matrices are obtained from predictions in learning and generalization phases, and the performance metrics are computed through the same process that was implemented earlier for the logistic regression model in sections 8 and 9. Confusion matrices and performance metrics are shown in the output below, and the ROC curves are displayed in Fig. 8 and 9. Moreover, the variance estimation function is executed to obtain the variance in the Naive Bayes model.

```
> # Naive Bayes Performance Parameters
> require(e1071)
> # Create a Naive Bayes Model
> formstr_nb=formula(formstr)
> trdf_nb=trdf_uncor
> tstdf_nb=tstdf_uncor
> trdf_nb$activity=as.factor(trdf_nb$activity) # Makes it run as classification
> nb_model1=naiveBayes(formstr_nb,data=trdf_nb) # Train the model
```

```
> # Predict using train data (Learning Phase)
> nb_pred1_tr=predict(nb_model1, trdf_nb[, -which(names(trdf_nb)=="activity")], type="class")
> nb_cfm1_tr=confusionMatrix(table(trdf_nb[, which(names(trdf_nb)=="activity")], nb_pred1_tr)) # Confusion Matrix for train data
> print("Naive-Bayes Learning Phase Confusion Matrix")
> nb_cfm1_tr
> nb_acc1_tr=round(nb_cfm1_tr$overall[['Accuracy']],4) # Accuracy of predictions with train data
> print(paste("Naive Bayes Learning Phase Accuracy =",nb_acc1_tr))
> # Performance Parameters
> nb_PM1_tr=nb_cfm1_tr$byClass[, c("Balanced Accuracy", "Precision", "Sensitivity", "Specificity", "Recall")]
> print("Naive-Bayes Learning-Phase Performance Parameters:")
> nb_PM1_tr
> nb_PMavg1_tr=round(apply(nb_PM1_tr,2,mean),4)
> print("Macro Averages:")
> t(nb_PMavg1_tr)
> nb_prob1_tr=predict(nb_model1, trdf_nb[, -which(names(trdf_nb)=="activity")], type="raw")
> nb_AUC1_tr=multiclass.roc(trdf_nb[, which(names(trdf_nb)=="activity")], nb_prob1_tr)
> print(paste("Naive-Bayes Learning-Phase AUC:", round(nb_AUC1_tr$auc, 4)))
> # ROC curves
> nb_ROC1_tr=nb_AUC1_tr$rocs
```

```
> ROC_num=paste("1/",as.character(2),sep="")
> plot.roc(nb_ROC1_tr[[ROC_num]][[2]], col=2, main="ROC curves of 7 One-vs-One
        class combinations (Learning Phase)")
> for(i in 3:8) {ROC_num=paste("1/",as.character(i),sep="")
+ lines.roc(nb_ROC1_tr[[ROC_num]][[2]],col=i)}
> legend("bottom",legend=c('1/2','1/3', '1/4','1/5', '1/6','1/7','1/8'), col=2:8,
         lwd=2)

[1] "Naive-Bayes Learning Phase Confusion Matrix"
Confusion Matrix and Statistics

    nb_pred1_tr
       1   2   3   4   5   6   7   8   9  10  11  12
  1  766  35  36   0   0   0   0   0   0   0   0   0
  2   34 661  55   0   0   0   0   0   0   0   2   0
  3   21  34 625   0   0   0   0   0   0   0   0   0
  4    0   2   0 393 416   5  16  10  21   1  28   0
  5    1   5   0  40 934   0  17   0   1   0  13   0
  6    0   0   0   0  34 910   0   0   2   9  14  36
  7    0   0   0   0   0   0  30   0   2   0   0   0
  8    0   0   0   0   0   0   0  14   0   0   0   0
  9    0   0   0   0   0   0   1   0  40   0  11   0
  10   0   0   0   0   0   0   0   0   1  27   1  12
  11   0   0   0   0   0   0   0   0  12   0  59   0
  12   0   0   0   0   0   1   0   0   1   3   1  43

Overall Statistics

               Accuracy : 0.8282
                 95% CI : (0.8179, 0.8381)
    No Information Rate : 0.2546
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.7977

[1] "Naive Bayes Learning Phase Accuracy = 0.8282"
```

```
[1] "Naive-Bayes Learning-Phase Performance Parameters:"
          Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1          0.9582428 0.9151732 0.9318735  0.9846121 0.9318735
Class: 2          0.9387567 0.8789894 0.8968792  0.9806342 0.8968792
Class: 3          0.9306262 0.9191176 0.8729050  0.9883475 0.8729050
Class: 4          0.9039405 0.4405830 0.9076212  0.9002598 0.9076212
Class: 5          0.8279263 0.9238378 0.6748555  0.9809970 0.6748555
Class: 6          0.9862160 0.9054726 0.9934498  0.9789823 0.9934498
Class: 7          0.7341888 0.9375000 0.4687500  0.9996277 0.4687500
Class: 8          0.7916667 1.0000000 0.5833333  1.0000000 0.5833333
Class: 9          0.7488798 0.7692308 0.5000000  0.9977595 0.5000000
Class: 10         0.8362027 0.6585366 0.6750000  0.9974055 0.6750000
Class: 11         0.7275516 0.8309859 0.4573643  0.9977388 0.4573643
Class: 12         0.7357025 0.8775510 0.4725275  0.9988775 0.4725275

[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]            0.8433    0.8381      0.7029      0.9838 0.7029

[1] "Naive-Bayes Learning-Phase AUC: 0.9898"
```

```
> # Predict using test data (Generalization Phase)
> nb_pred1_tst=predict(nb_model1, tstdf_nb[, -which(names(tstdf_nb)=="activity")], type="class")
> nb_cfm1_tst=confusionMatrix(table(tstdf_nb[, which(names(tstdf_nb)=="activity")], nb_pred1_tst)) # Confusion Matrix for test data
> print("Naive-Bayes Generalization Phase Confusion Matrix")
> nb_cfm1_tst
> nb_acc1_tst=round(nb_cfm1_tst$overall[['Accuracy']],4) # Accuracy of predictions with test data
> print(paste("Naive Bayes Generalization Phase Accuracy =",nb_acc1_tst))
> # Check for over-fitting. Criteria: Accuracy change from train to test > 25%
> nb_model1_isOF=abs((nb_acc1_tr-nb_acc1_tst)/nb_acc1_tr)
> nb_model1_isOF=round(nb_model1_isOF,4)
> print(paste("Accuracy drop from training data to test data is",nb_model1_isOF*100,"%"))
> if(nb_model1_isOF>0.25) print("Model is over-fitting") else print("Model is not over-fitting")
> # Performance Parameters
> nb_PM1_tst=nb_cfm1_tst$byClass[, c("Balanced Accuracy", "Precision", "Sensitivity", "Specificity", "Recall")]
> print("Naive-Bayes Generalization-Phase Performance Parameters:")
> nb_PM1_tst
> nb_PMavg1_tst=round(apply(nb_PM1_tst,2,mean),4)
> print("Macro Averages:")
```

```
> t(nb_PMavg1_tst)
> nb_prob1_tst=predict(nb_model1, tstdf_nb[, -which(names(tstdf_nb)=="activity")
        ], type="raw")
> nb_AUC1_tst=multiclass.roc(tstdf_nb[, which(names(tstdf_nb)=="activity")],
        nb_prob1_tst)
> print(paste("Naive-Bayes Generalization-Phase AUC:", round(nb_AUC1_tst$auc, 4))
        )
> # ROC curves
> nb_ROC1_tst=nb_AUC1_tst$rocs
> ROC_num=paste("1/",as.character(2),sep="")
> plot.roc(nb_ROC1_tst[[ROC_num]][[2]], col=2, main="ROC curves of 7 One-vs-One
        class combinations (Generalization Phase)")
> for(i in 3:8) {ROC_num=paste("1/",as.character(i),sep="")
+ lines.roc(nb_ROC1_tst[[ROC_num]][[2]],col=i)}
> legend("bottom",legend=c('1/2','1/3','1/4','1/5' ,'1/6','1/7','1/8'), col=2:8,
        lwd=2)

[1] "Naive-Bayes Generalization Phase Confusion Matrix"
Confusion Matrix and Statistics

    nb_pred1_tst
       1   2   3   4   5   6   7   8   9  10  11  12
  1  354  14  21   0   0   0   0   0   0   0   0   0
  2   15 282  22   0   0   0   0   0   0   0   2   0
  3    8  14 285   0   0   0   0   0   0   0   0   0
  4    0   1   0 165 200   6   5   5   8   1  10   0
  5    0   0   0  20 373   0  11   0   0   0   8   0
  6    0   0   0   0  12 374   0   0   1   7   5   9
  7    0   2   0   0   0   0  13   0   0   0   0   0
  8    0   0   0   0   0   0   0   8   0   0   0   1
  9    0   0   0   0   0   0   0   0  19   0   4   0
  10   0   0   0   0   0   0   0   0   0   7   0  12
  11   0   0   0   0   0   0   0   0   1   0  18   0
  12   0   0   0   0   0   0   0   0   1   1   0   6

Overall Statistics

               Accuracy : 0.8168
                 95% CI : (0.8005, 0.8323)
    No Information Rate : 0.251
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.7842

[1] "Naive Bayes Generalization Phase Accuracy = 0.8168"
```

```
[1] "Accuracy drop from training data to test data is 1.38 %"

[1] "Model is not over-fitting"

[1] "Naive-Bayes Generalization-Phase Performance Parameters:"
          Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1          0.9605400 0.9100257 0.9389920  0.9820880 0.9389920
Class: 2          0.9408162 0.8785047 0.9009585  0.9806739 0.9009585
Class: 3          0.9289595 0.9283388 0.8689024  0.9890165 0.8689024
Class: 4          0.8909599 0.4114713 0.8918919  0.8900280 0.8918919
Class: 5          0.8076350 0.9053398 0.6376068  0.9776632 0.6376068
Class: 6          0.9833918 0.9166667 0.9842105  0.9825730 0.9842105
Class: 7          0.7237035 0.8666667 0.4482759  0.9991312 0.4482759
Class: 8          0.8074766 0.8888889 0.6153846  0.9995686 0.6153846
Class: 9          0.8157975 0.8260870 0.6333333  0.9982616 0.6333333
Class: 10         0.7161582 0.3684211 0.4375000  0.9948164 0.4375000
Class: 11         0.6912704 0.9473684 0.3829787  0.9995622 0.3829787
Class: 12         0.6067086 0.7500000 0.2142857  0.9991316 0.2142857

[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]            0.8228    0.7998      0.6629      0.9827 0.6629

[1] "Naive-Bayes Generalization-Phase AUC: 0.9813"
```

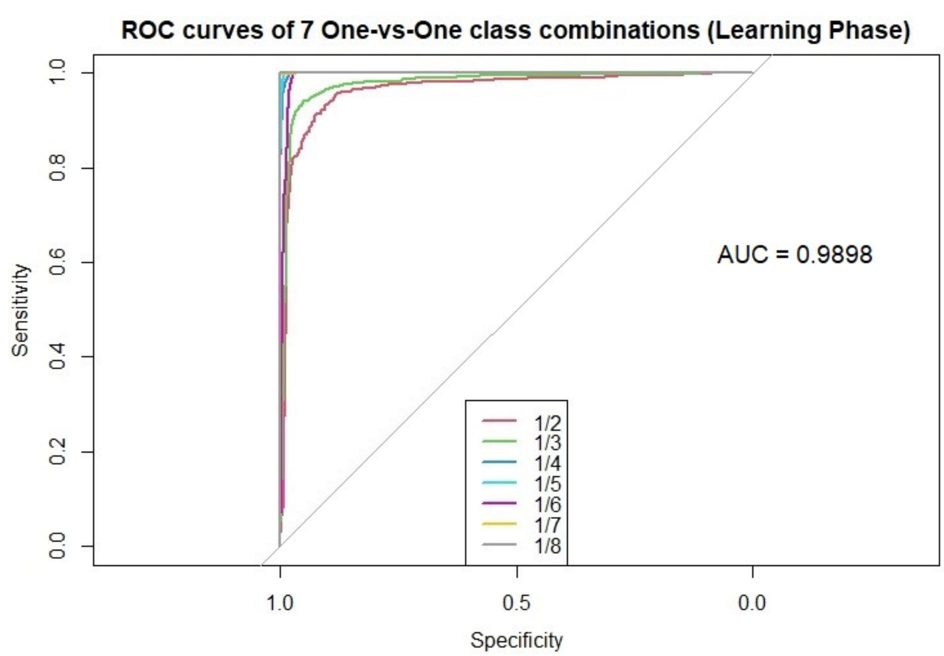

Figure 8: ROC curves for the learning phase of the Naive Bayes model.

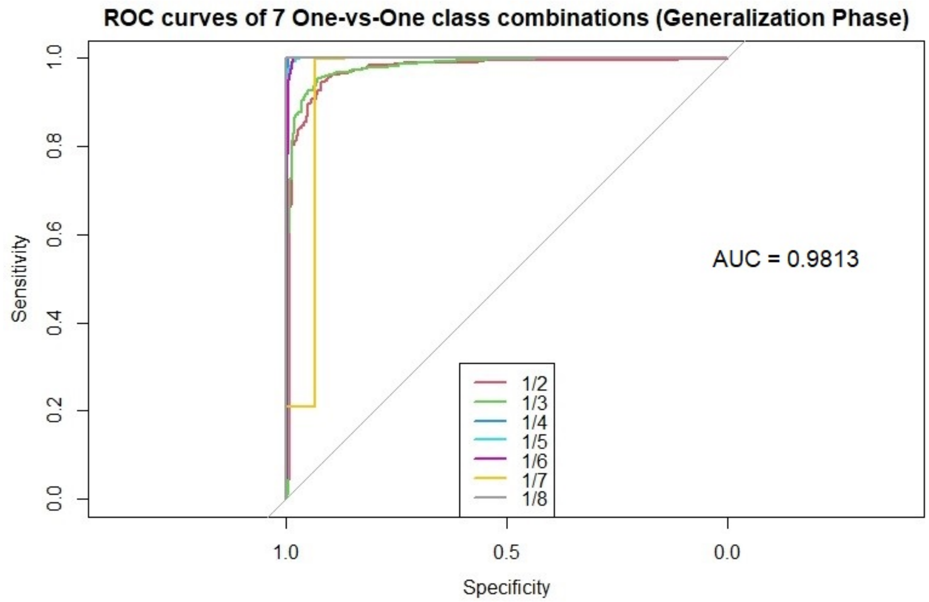

Figure 9: ROC curves for the generalization phase of Naive Bayes model.

```
> # Variance Estimation
> nb_varEst30=varEst(varEst_trdf, varEst_tstdf, 30, type="nb") # Variance
        estimation using 30% of the data
> nb_varEst60=varEst(varEst_trdf, varEst_tstdf, 60, type="nb") # Variance
        estimation using 60% of the data
> nb_varEst100=varEst(varEst_trdf, varEst_tstdf, 100, type="nb") # Variance
        estimation using 100% of the data
> print("Naive-Bayes Variance Estimation using 30% of data:")
> nb_varEst30
> print("Naive-Bayes Variance Estimation using 60% of data:")
> nb_varEst60
> print("Naive-Bayes Variance Estimation using 100% of data:")
> nb_varEst100

[1] "Naive-Bayes Variance Estimation using 30% of data:"
Mean of Accuracies   0.7876000
Variance of Accuracies 0.0001969

[1] "Naive-Bayes Variance Estimation using 60% of data:"
Mean of Accuracies   7.981e-01
Variance of Accuracies 7.983e-05

[1] "Naive-Bayes Variance Estimation using 100% of data:"
Mean of Accuracies   0.8051
Variance of Accuracies 0.0000
```

11 kNN Classifier

K-nearest neighbors (kNN) is a non-parametric method for classification, meaning that there are no model parameters to estimate. Therefore, there is no learning phase with this algorithm, since it directly predicts the class of previously unseen data based on the available labeled observations. The `knn()` function from `class` package is used, which takes the training set, its labels, and testing set as inputs, and returns the class predictions for the testing set. Since our number of classes is 12, the value of k should be an odd number larger than 12; hence; k was set as 15. The confusion matrix, performance metrics, and ROC curve (Fig. 10) are provided below.

```
> # kNN Classifier
> require(class)
> trdf_knn=trdf_uncor[, -which(names(trdf_uncor)=="activity")]
> tstdf_knn=tstdf_uncor[, -which(names(trdf_uncor)=="activity")]
> trclass_knn=factor(trdf_uncor[, which(names(trdf_uncor)=="activity")])
> tstclass_knn=factor(tstdf_uncor[, which(names(tstdf_uncor)=="activity")])
> knn_pred1=knn(trdf_knn,tstdf_knn,trclass_knn, k = 15, prob=TRUE)
```

```
> # Predict using test data (Generalization Phase)
> knn_cfm1_tst=confusionMatrix(table(tstclass_knn,knn_pred1)) # Confusion Matrix
      for test data
> knn_cfm1_tst
> knn_acc1_tst=round(knn_cfm1_tst$overall[['Accuracy']],4) # Accuracy of
      predictions with test data
> print(paste("kNN Generalization Phase Accuracy =",knn_acc1_tst))
> # Performance Parameters
> knn_PM1_tst=knn_cfm1_tst$byClass[, c("Balanced Accuracy", "Precision", "
      Sensitivity", "Specificity", "Recall")]
> print("kNN Generalization-Phase Performance Parameters:")
> knn_PM1_tst
> knn_PMavg1_tst=round(apply(knn_PM1_tst,2,mean),4)
> print("Macro Averages:")
> t(knn_PMavg1_tst)
> knn_prob1_tst=attr(knn_pred1,"prob")
> knn_AUC1_tst=multiclass.roc(tstclass_knn, as.ordered(knn_pred1))
> print(paste("kNN Generalization-Phase AUC:",round(knn_AUC1_tst$auc,4)))
> # ROC curves
> knn_ROC1_tst=knn_AUC1_tst$rocs
> plot.roc(knn_ROC1_tst[[1]], col=1, main="ROC curves of 7 One-vs-One class
      combinations")
> for(i in 2:7) {lines.roc(knn_ROC1_tst[[i]],col=i)}
> legend("bottom",legend=c('1/2','1/3','1/4', '1/5','1/6','1/7','1/8'), col=1:7,
      lwd=2)

```

```
Confusion Matrix and Statistics

           knn_pred1
tstclass_knn  1   2   3   4   5   6   7   8   9  10  11  12
          1 383   1   4   0   1   0   0   0   0   0   0   0
          2  13 305   3   0   0   0   0   0   0   0   0   0
          3  15   5 287   0   0   0   0   0   0   0   0   0
          4   0   1   0 249 147   4   0   0   0   0   0   0
          5   1   1   0  34 376   0   0   0   0   0   0   0
          6   0   0   0   2   2 404   0   0   0   0   0   0
          7   0   1   0   0   8   0   6   0   0   0   0   0
          8   0   0   0   4   4   0   0   1   0   0   0   0
          9   0   1   0   7   1   0   0   0  10   0   4   0
          10  0   0   0   2   0   2   0   0   0   4   0  11
          11  2   3   0   4   2   0   0   0   3   0   5   0
          12  0   1   0   1   0   2   0   0   0   0   0   4

Overall Statistics

               Accuracy : 0.8726
                 95% CI : (0.8584, 0.8859)
    No Information Rate : 0.2321
    P-Value [Acc > NIR] : < 2.2e-16
                  Kappa : 0.8484

[1] "kNN Generalization Phase Accuracy = 0.8726"

[1] "kNN Generalization-Phase Performance Parameters:"
          Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1          0.9609954 0.9845758 0.9251208   0.9968701 0.9251208
Class: 2          0.9740803 0.9501558 0.9561129   0.9920477 0.9561129
Class: 3          0.9831861 0.9348534 0.9761905   0.9901816 0.9761905
Class: 4          0.8734157 0.6209476 0.8217822   0.9250493 0.8217822
Class: 5          0.8374488 0.9126214 0.6950092   0.9798883 0.6950092
Class: 6          0.9892491 0.9901961 0.9805825   0.9979156 0.9805825
Class: 7          0.9980645 0.4000000 1.0000000   0.9961290 1.0000000
Class: 8          0.9982833 0.1111111 1.0000000   0.9965665 1.0000000
Class: 9          0.8818112 0.4347826 0.7692308   0.9943917 0.7692308
Class: 10         0.9967770 0.2105263 1.0000000   0.9935539 1.0000000
Class: 11         0.7747631 0.2631579 0.5555556   0.9939707 0.5555556
Class: 12         0.6324698 0.5000000 0.2666667   0.9982729 0.2666667
[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]            0.9084    0.6094      0.8289      0.9879 0.8289
[1] "kNN Generalization-Phase AUC: 0.8216"
```

```
> # Variance Estimation
> knn_varEst30=varEst(varEst_trdf, varEst_tstdf, 30, type="knn") # 30% of data
> knn_varEst60=varEst(varEst_trdf, varEst_tstdf, 60, type="knn") # 60% of data
> knn_varEst100=varEst(varEst_trdf, varEst_tstdf, 100, type="knn") # 100% of data
> print("kNN Variance Estimation using 30% of data:")
> knn_varEst30
> print("kNN Variance Estimation using 60% of data:")
> knn_varEst60
> print("kNN Variance Estimation using 100% of data:")
> knn_varEst100

[1] "kNN Variance Estimation using 30% of data:"
Mean of Accuracies   0.8128000
Variance of Accuracies 0.0000908

[1] "kNN Variance Estimation using 60% of data:"
Mean of Accuracies   8.378e-01
Variance of Accuracies 5.701e-05

[1] "kNN Variance Estimation using 100% of data:"
Mean of Accuracies   8.585e-01
Variance of Accuracies 5.077e-06
```

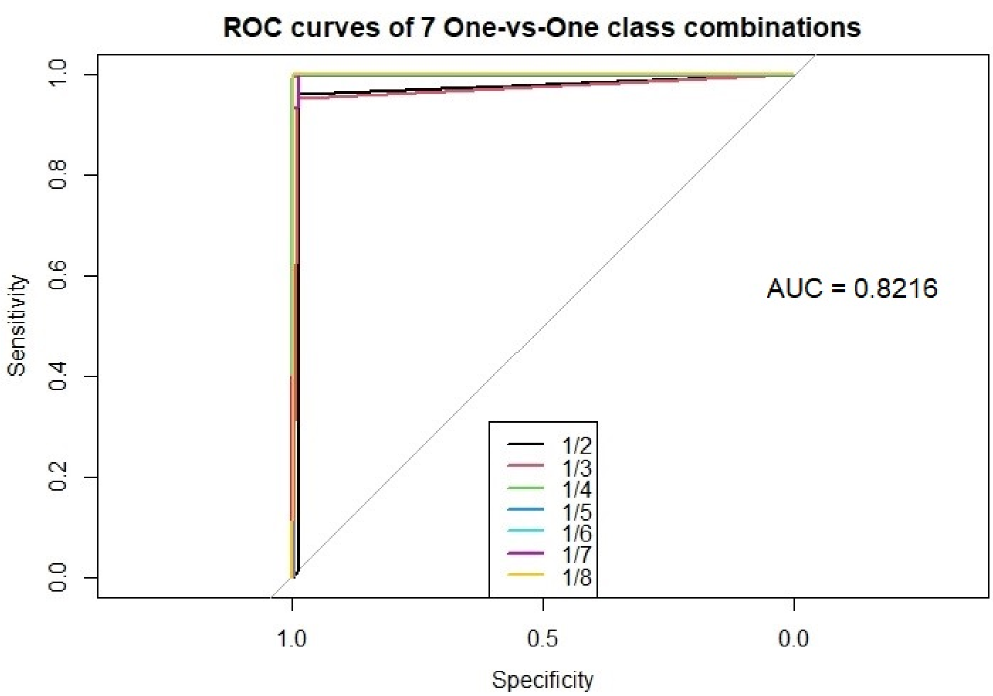

Figure 10: ROC curves for the generalization phase of kNN classifier.

12 Dealing with the Class Imbalance Issue

Up until this point, we have been working with the data set as it is without addressing the imbalance issue that was observed during EDA in section 3. Out of the three classifiers, it seems that the logistic regression model performed the best in terms of classification accuracy. However, in each of the three classifiers, the classes that suffer imbalance, which are `"7", "8", "9", "10", "11", and "12"`, have low balanced accuracies, as can be seen in the performance metrics table evaluated earlier for each respective classifier. For example, the balanced accuracy of class `"11"` was 65% in the logistic regression model, 69% in the Naive Bayes model, and 77% in the kNN classifier. Also, for class `"12"`, the accuracies were found to be 77%, 61% and 63% in logistic, Naive Bayes and kNN, respectively. Furthermore, we notice that the precision for these classes, which is measured as $\frac{TP}{TP+FP}$, are quite low. These performance metric values are considered to be poor and we need to improve them.

In order to tackle the class imbalance issue, the training data is split into two groups; group A contains the observations with classes from 1 to 6, and group B contains classes 7 to 12, and a classifier is trained for each group. Then, the classification process is divided into two steps. In the first step, we determine the group that the observation belongs to, which is performed using a trained SVM model since it is a non-probabilistic classifier; hence, it is less susceptible to class imbalance issues. The second step is to classify the observation using the model of the group that it belongs to. Moving forward, this process will be loosely referred to as Dual-group Classification.

The below R code implements the dual-group classification process, and the results are provided in the output. Here, we used logistic regression (`multinom()`) to train the two models for groups A and B. Now, we compare the results of this classifier with those obtained for the logistic regression model in section 9. Even though the learning phase accuracy is very slightly worse in the dual-group classifier, the generalization phase accuracy is slightly better. More importantly, there is a noticeable improvement in the balanced accuracies of the imbalanced classes 7 to 12. The accuracy in classes 7, 8, 9, 10, and 11 improved by 10.5%, 1.7%, 12.8%, 8%, and 34.1%, respectively. However, in class 12 it worsened by 2.6%, and in class 8 the accuracy is still undesirably below 70%. Also, the precision values did not demonstrate any improvement.

```
> ###### Dual-group Classification #######
> target = which(names(trdf_uncor)=="activity")
> # Change the labels of classes 1-6 to 0, and classes 7-12 to 1 for training
      data
> class16_idx=which(trdf_uncor$activity %in% 1:6) # Indeces of rows containing
      classes 1-6
> class712_idx=which(trdf_uncor$activity %in% 7:12) # Indeces of rows containing
      classes 7-12
> trdf_svm=trdf_uncor # Load the data to a different variable
> trdf_svm$activity[class16_idx]=rep(0,length(class16_idx)) # Change the classes
      1-6 to 0s
> trdf_svm$activity[class712_idx]=rep(1,length(class712_idx))# Change the classes
```

```
        7-12 to 1s
> table(trdf_svm$activity)

   0    1
5177  259
```

```
> # Create the model for group detection
> trdf_svm$activity=as.factor(trdf_svm$activity) # Makes it run classification
> svm_model=svm(activity~.,data=trdf_svm, probability=TRUE) # Train an SVM model
> # Partition the training data into two subsets A and B. Subset A contains
       classes 1-6, and subset B contains 7-12
> trdfA=trdf_uncor[class16_idx,] # Create subset A
> trdfB=trdf_uncor[class712_idx,] # Create subset B
> # Train a classifier on subset A
> trdfA$activity=as.factor(trdfA$activity)
> mn_modelA=multinom(activity~.,trdfA,maxit=1000)
> # Train a classifier on subset B
> trdfB$activity=as.factor(trdfB$activity)
> mn_modelB=multinom(activity~.,trdfB,maxit=1000)
```

```
> # Predict using full train data
> svm_pred_tr = predict(svm_model,trdf_uncor[, -target], type="class") # Predict
       with the group detector
> svm_pred_tr_idx0=which(svm_pred_tr==0) # Get indeces of data to be predicted by
        classifier A (classes 1-6)
> svm_pred_tr_idx1=which(svm_pred_tr==1) # Get indeces of data to be predicted by
        classifier B (classes 7-12)
>
> mn_predA_tr = predict(mn_modelA, trdf_uncor[svm_pred_tr_idx0, -target], type="
       class") # Predict the data that belong to subset A with classifier A
> mn_predB_tr = predict(mn_modelB, trdf_uncor[svm_pred_tr_idx1, -target], type="
       class") # Predict the data that belong to subset B with classifier B
>
> grp_pred_tr=rep(0,length(mn_predA_tr)+length(mn_predB_tr)) # Create a vector to
        combine the predictions
> grp_pred_tr[svm_pred_tr_idx0]=as.numeric(as.character(mn_predA_tr))
> grp_pred_tr[svm_pred_tr_idx1]=as.numeric(as.character(mn_predB_tr))
>
> grp_cfm_tr=confusionMatrix(table(trdf_uncor[,target], grp_pred_tr)) # Confusion
        Matrix for train data
> grp_cfm_tr
> print("Dual-group Method Learning-Phase Confusion Matrix")
> grp_acc_tr=round(grp_cfm_tr$overall[['Accuracy']],4) # Accuracy of predictions
       with train data
```

```
> print(paste("Dual-group Method Learning-Phase Accuracy =", grp_acc_tr))
> # Performance Parameters
> grp_PM_tr=grp_cfm_tr$byClass[, c("Balanced Accuracy", "Precision", "Sensitivity", "Specificity", "Recall")]
> print("Dual-group Method Learning-Phase Performance Parameters:")
> grp_PM_tr
> grp_PMavg_tr=round(apply(grp_PM_tr,2,mean),4)
> print("Macro Averages:")
> t(grp_PMavg_tr)

[1] "Dual-group Method Learning-Phase Confusion Matrix"
Confusion Matrix and Statistics

   grp_pred_tr
       1    2    3    4    5    6    7    8    9   10   11   12
  1  824    6    7    0    0    0    0    0    0    0    0    0
  2    8  730   14    0    0    0    0    0    0    0    0    0
  3    4   13  663    0    0    0    0    0    0    0    0    0
  4    0    0    0  724  167    0    1    0    0    0    0    0
  5    0    0    0  133  878    0    0    0    0    0    0    0
  6    0    0    0    0    0 1005    0    0    0    0    0    0
  7    0    0    0    1    1    0   30    0    0    0    0    0
  8    0    3    0    3    0    0    0    8    0    0    0    0
  9    0    0    0    1    0    0    0    0   51    0    0    0
  10   0    0    0    0    0    0    0    0    0   41    0    0
  11   0    0    0    1    0    0    0    0    0    0   70    0
  12   0    0    0    0    0    1    0    0    0    0    0   48

Overall Statistics

               Accuracy : 0.933
                 95% CI : (0.9261, 0.9395)
    No Information Rate : 0.1924
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9208

[1] "Dual-group Method Learning-Phase Accuracy = 0.933"

```

```
[1] "Dual-group Method Learning-Phase Performance Parameters:"
         Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1         0.9914099 0.9844683 0.9856459  0.9971739 0.9856459
Class: 2         0.9830239 0.9707447 0.9707447  0.9953032 0.9707447
Class: 3         0.9828604 0.9750000 0.9692982  0.9964226 0.9692982
Class: 4         0.9010983 0.8116592 0.8389340  0.9632626 0.8389340
Class: 5         0.9045460 0.8684471 0.8393881  0.9697039 0.8393881
Class: 6         0.9995030 1.0000000 0.9990060  1.0000000 0.9990060
Class: 7         0.9836860 0.9375000 0.9677419  0.9996300 0.9677419
Class: 8         0.9994473 0.5714286 1.0000000  0.9988946 1.0000000
Class: 9         0.9999071 0.9807692 1.0000000  0.9998143 1.0000000
Class: 10        1.0000000 1.0000000 1.0000000  1.0000000 1.0000000
Class: 11        0.9999068 0.9859155 1.0000000  0.9998136 1.0000000
Class: 12        0.9999072 0.9795918 1.0000000  0.9998144 1.0000000

[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]            0.9788    0.9221      0.9642      0.9933 0.9642
```

```
> # Predict using test data
> svm_pred_tst = predict(svm_model, tstdf_uncor[,-target], type="class") # Predict with the group detector
> svm_pred_tst_idx0=which(svm_pred_tst==0) # Get indeces of data to be predicted by classifier A (classes 1-6)
> svm_pred_tst_idx1=which(svm_pred_tst==1) # Get indeces of data to be predicted by classifier B (classes 7-12)
>
> mn_predA_tst = predict(mn_modelA, tstdf_uncor[svm_pred_tst_idx0,-target], type="class") # Predict the data that belong to subset A with classifier A
> mn_predB_tst = predict(mn_modelB, tstdf_uncor[svm_pred_tst_idx1,-target], type="class") # Predict the data that belong to subset B with classifier B
>
> grp_pred_tst=rep(0,length(mn_predA_tst)+length(mn_predB_tst)) # Create a vector to combine the predictions
> grp_pred_tst[svm_pred_tst_idx0]=as.numeric(as.character(mn_predA_tst))
> grp_pred_tst[svm_pred_tst_idx1]=as.numeric(as.character(mn_predB_tst))
>
> grp_cfm_tst=confusionMatrix(table(tstdf_uncor[,target],grp_pred_tst)) # Confusion Matrix for test data
> print("Dual-group Method Generalization-Phase Confusion Matrix")
> grp_cfm_tst
> grp_acc_tst=round(grp_cfm_tst$overall[['Accuracy']],4) # Accuracy of predictions with test data
> print(paste("Dual-group Method Generalization-Phase Accuracy =", grp_acc_tst))
> # Check for over-fitting. Criteria: Accuracy change from train to test > 25%
```

```
> grp_model_isOF=abs((grp_acc_tr-grp_acc_tst)/grp_acc_tr)
> grp_model_isOF=round(grp_model_isOF,4)
> print(paste("Accuracy drop from training data to test data is",grp_model_isOF*100,"%"))
> if(grp_model_isOF>0.25) print("Model is over-fitting") else print("Model is not over-fitting")
> # Performance Parameters
> grp_PM_tst=grp_cfm_tst$byClass[, c("Balanced Accuracy", "Precision", "Sensitivity", "Specificity", "Recall")]
> print("Dual-group Method Learning-Phase Performance Parameters:")
> grp_PM_tst
> grp_PMavg_tst=round(apply(grp_PM_tst,2,mean),4)
> print("Macro Averages:")
> t(grp_PMavg_tst)

[1] "Dual-group Method Generalization-Phase Confusion Matrix"
Confusion Matrix and Statistics

   grp_pred_tst
      1   2   3   4   5   6   7   8   9  10  11  12
  1 379   4   5   0   1   0   0   0   0   0   0   0
  2   6 307   8   0   0   0   0   0   0   0   0   0
  3   5  15 287   0   0   0   0   0   0   0   0   0
  4   0   0   0 306  93   2   0   0   0   0   0   0
  5   0   2   0  63 347   0   0   0   0   0   0   0
  6   0   0   0   0   0 408   0   0   0   0   0   0
  7   0   2   0   1   2   0   9   1   0   0   0   0
  8   0   1   0   4   0   1   0   2   0   1   0   0
  9   0   0   0   1   0   1   1   0  15   1   4   0
  10  0   0   0   0   0   0   0   0   0  14   0   5
  11  0   1   0   1   0   0   0   0   5   0  12   0
  12  0   0   1   0   0   0   0   2   0   0   0   5

Overall Statistics

               Accuracy : 0.897
                 95% CI : (0.884, 0.9091)
    No Information Rate : 0.19
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8779

[1] "Dual-group Method Generalization-Phase Accuracy = 0.897"

[1] "Accuracy drop from training data to test data is 3.86 %"
```

```
[1] "Model is not over-fitting"

[1] "Dual-group Method Generalization-Phase Performance Parameters:"
          Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1          0.9833214 0.9742931 0.9717949  0.9948480 0.9717949
Class: 2          0.9588476 0.9563863 0.9246988  0.9929965 0.9246988
Class: 3          0.9718181 0.9348534 0.9534884  0.9901478 0.9534884
Class: 4          0.8826182 0.7630923 0.8138298  0.9514066 0.8138298
Class: 5          0.8744339 0.8422330 0.7832957  0.9655720 0.7832957
Class: 6          0.9951456 1.0000000 0.9902913  1.0000000 0.9902913
Class: 7          0.9487075 0.6000000 0.9000000  0.9974149 0.9000000
Class: 8          0.6984953 0.2222222 0.4000000  0.9969905 0.4000000
Class: 9          0.8732691 0.6521739 0.7500000  0.9965383 0.7500000
Class: 10         0.9364201 0.7368421 0.8750000  0.9978402 0.8750000
Class: 11         0.8734881 0.6315789 0.7500000  0.9969762 0.7500000
Class: 12         0.7493537 0.6250000 0.5000000  0.9987075 0.5000000

[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]            0.8955    0.7449      0.801       0.99  0.801
```

```
[1] "Generalization-Phase Performance Parameters of Logistic-Regression Model
         without Dual-group Method, for Comparison:"
          Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1          0.9814862 0.9691517 0.9691517  0.9938208 0.9691517
Class: 2          0.9661471 0.9314642 0.9432177  0.9890765 0.9432177
Class: 3          0.9750119 0.9348534 0.9598662  0.9901575 0.9598662
Class: 4          0.8826182 0.7630923 0.8138298  0.9514066 0.8138298
Class: 5          0.8757307 0.8398058 0.7863636  0.9650978 0.7863636
Class: 6          0.9981865 0.9828431 1.0000000  0.9963731 1.0000000
Class: 7          0.8560639 0.6666667 0.7142857  0.9978420 0.7142857
Class: 8          0.6862086 0.3333333 0.3750000  0.9974171 0.3750000
Class: 9          0.7743417 0.6956522 0.5517241  0.9969592 0.5517241
Class: 10         0.8673397 0.7368421 0.7368421  0.9978374 0.7368421
Class: 11         0.6514600 0.4210526 0.3076923  0.9952278 0.3076923
Class: 12         0.7690151 0.8750000 0.5384615  0.9995686 0.5384615

[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]             0.857    0.7625     0.7247     0.9892 0.7247
```

13 Dimensionality Reduction Using PCA

Principal Component Analysis (PCA) is a technique used to reduce the dimensionality of the data set and improve model's performance. It involves decomposing the feature columns into principal components (PCs) by solving an eigenvalue problem. The resulting eigenvectors represent the PCs that will be used to train the model, and the eigenvalue of each PC provides information of its importance.

The `prcomp()` function in R is used to carry out the PCA on the entire data set containing all 561 features. A summary of the first 18 PCs is shown in the output, sorted according to their eigenvalues. Since eignevalues are calculated based on the variance of the data, the standard deviation here corresponds to the square root of the eigenvalue for each PC. Proportion of Variance indicates the amount of variance explained by that single PC, and the Cumulative Proportion is just the amount of variance captured by that PC and the ones before it accumulated. A threshold value of 1.5 was set on the standard deviation, which means that PCs with standard deviation larger than 1.5 will be used to train the model. It was found that there are 28 PCs satisfying this criteria, as shown in Fig. 11, and they account for about 81% of the variance in the data.

```r
> # Perform PCA using all features
> target=which(names(trdf)=="activity")
> pca=prcomp(trdf[,-target], center = TRUE, scale = TRUE) # Run PCA
> pca_summary=summary(pca)
> pca_summary$importance[,1:18] # Display a summary of the important principal components
> pca_std=pca_summary$sdev # Standard deviations of the principal components
> high_pca_idx=which(pca_std>=1.5) # Get the principal components with standard deviation >= 1.5
> print(paste("Number of PCs with standard deviation larger than 1.5 is", length(high_pca_idx)))
> pca_var_1.5=pca_summary$importance['Cumulative Proportion',min(which(pca_std<1.5))] # Amount of variance accounted for in PCs with sdev > 1.5
> print(paste("Compulative variance proportion for PCs with standard deviation larger than 1.5 is", pca_var_1.5*100, "%"))
> plot(x = 1:40, y = pca_std[1:40], type = "o", xlab = "Principal Component", ylab = "Standard Deviation",xaxp=c(1,40,39))
> abline(h = 1.5, col="red", lty=5)
> legend("top", legend="Standard Deviation = 1.5", col=c("red"), lty=20, bty="n")
```

```
                                PC1      PC2     PC3      PC4      PC5      PC6
Standard deviation   16.43764 6.796284 4.23171 3.678691 3.176991 3.088045
Proportion of Variance 0.48163 0.082330 0.03192 0.024120 0.017990 0.017000
Cumulative Proportion 0.48163 0.563970 0.59589 0.620010 0.638000 0.655000
                                PC7      PC8     PC9    PC10     PC11    PC12
Standard deviation   2.750629 2.676283 2.52943 2.34298 2.256404 2.15163
Proportion of Variance 0.013490 0.012770 0.01140 0.00979 0.009080 0.00825
Cumulative Proportion 0.668490 0.681250 0.69266 0.70244 0.711520 0.71977
                               PC13     PC14    PC15     PC16     PC17     PC18
Standard deviation   2.097646 1.992618 1.94018 1.857391 1.827337 1.819076
Proportion of Variance 0.007840 0.007080 0.00671 0.006150 0.005950 0.005900
Cumulative Proportion 0.727610 0.734690 0.74140 0.747550 0.753500 0.759400

[1] "Number of PCs with standard deviation larger than 1.5 is 28"

[1] "Cumulative variance proportion for PCs with standard deviation larger than
        1.5 is 81.225 %"
```

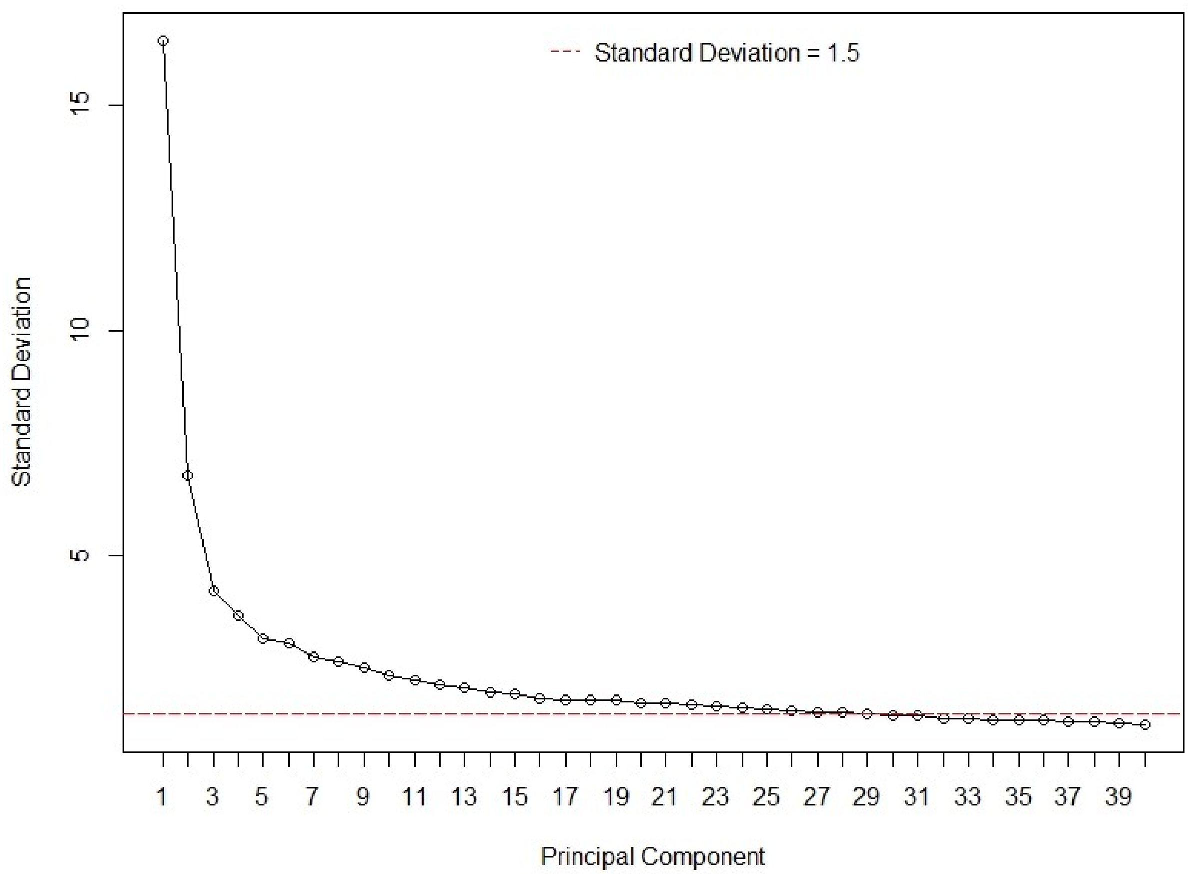

Figure 11: Plot of top 40 PCs and their standard deviations ($\sqrt{Eigenvalue}$).

```
> # Prepare the training set
> trdf_pca=pca$x[,high_pca_idx] # Form a data set from the the principal
      components
> trdf_pca=data.frame(trdf_pca) # Convert to data frame type
> trdf_pca$activity=trdf$activity # Add the class variable
> # Prepare the testing set
> tstdf_pca=predict(pca, newdata=tstdf[,-target]) # Get the principal components
      of testing set
> tstdf_pca=tstdf_pca[,high_pca_idx] # Form a data set from the the principal
      components
> tstdf_pca=data.frame(tstdf_pca) # Convert to data frame type
> tstdf_pca$activity=tstdf$activity # Add the class variable
> # Train a logistic regression model using principal components
> mn_model3=multinom(formstr,trdf_pca,maxit=1000)
```

```
> # Predict using train data (Learning Phase)
> target_pca= which(names(trdf_pca)=="activity")
> mn_pred3_tr=predict(mn_model3, trdf_pca[, -target_pca], type="class")
> mn_cfm3_tr=confusionMatrix(table(trdf_pca[, target_pca], mn_pred3_tr)) #
      Confusion Matrix for train data
> print("PCA Learning Phase Confusion Matrix")
> mn_cfm3_tr
> mn_acc3_tr=round(mn_cfm3_tr$overall[['Accuracy']],4) # Accuracy of predictions
      with train data
> print(paste("PCA Learning Phase Accuracy =",mn_acc3_tr))
> # Performance Parameters
> mn_PM3_tr=mn_cfm3_tr$byClass[, c("Balanced Accuracy", "Precision", "Sensitivity
      ", "Specificity", "Recall")]
> print("PCA Learning-Phase Performance Parameters:")
> mn_PM3_tr
> mn_PMavg3_tr=round(apply(mn_PM3_tr,2,mean),4)
> print("Macro Averages:")
> t(mn_PMavg3_tr)
> mn_prob3_tr=predict(mn_model3, trdf_pca[, -target_pca], type="probs")
> mn_AUC3_tr=multiclass.roc(trdf_pca[, target_pca], mn_prob3_tr)
> print(paste("PCA Learning-Phase AUC:", round(mn_AUC3_tr$auc, 4)))

```

```
[1] "PCA Learning Phase Confusion Matrix"
Confusion Matrix and Statistics

   mn_pred3_tr
       1    2    3    4    5    6    7    8    9   10   11   12
  1  815    8   14    0    0    0    0    0    0    0    0    0
  2    9  721   22    0    0    0    0    0    0    0    0    0
  3   12   23  645    0    0    0    0    0    0    0    0    0
  4    0    0    0  733  156    3    0    0    0    0    0    0
  5    0    0    0  100  911    0    0    0    0    0    0    0
  6    0    0    0    2    0 1003    0    0    0    0    0    0
  7    0    0    0    0    0    0   32    0    0    0    0    0
  8    0    0    0    0    0    0    0   14    0    0    0    0
  9    0    0    0    0    0    0    0    0   39    0   13    0
  10   0    0    0    0    0    0    0    0    0   33    0    8
  11   0    0    0    0    0    0    0    0    9    0   62    0
  12   0    0    0    0    0    0    0    0    0    6    0   43

Overall Statistics
             Accuracy : 0.9292
               95% CI : (0.922, 0.9359)
  No Information Rate : 0.1963
  P-Value [Acc > NIR] : < 2.2e-16
                Kappa : 0.9162

[1] "PCA Learning Phase Accuracy = 0.9292"

[1] "PCA Learning-Phase Performance Parameters:"
          Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1          0.9850489 0.9737157 0.9748804  0.9952174 0.9748804
Class: 2          0.9760792 0.9587766 0.9587766  0.9933817 0.9587766
Class: 3          0.9698879 0.9485294 0.9471366  0.9926393 0.9471366
Class: 4          0.9216433 0.8217489 0.8778443  0.9654423 0.8778443
Class: 5          0.9154536 0.9010880 0.8537957  0.9771115 0.8537957
Class: 6          0.9982832 0.9980100 0.9970179  0.9995485 0.9970179
Class: 7          1.0000000 1.0000000 1.0000000  1.0000000 1.0000000
Class: 8          1.0000000 1.0000000 1.0000000  1.0000000 1.0000000
Class: 9          0.9050436 0.7500000 0.8125000  0.9975872 0.8125000
Class: 10         0.9223358 0.8048780 0.8461538  0.9985177 0.8461538
Class: 11         0.9124939 0.8732394 0.8266667  0.9983212 0.8266667
Class: 12         0.9210115 0.8775510 0.8431373  0.9988858 0.8431373
[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]            0.9523     0.909      0.9115      0.9931 0.9115
[1] "PCA Learning-Phase AUC: 0.9964"
```

```
> # Predict using test data (Generalization Phase)
> mn_pred3_tst=predict(mn_model3, tstdf_pca[, -target_pca], type="class")
> mn_cfm3_tst=confusionMatrix(table(tstdf_pca[, target_pca], mn_pred3_tst)) # Confusion Matrix for test data
> print("PCA Generalization Phase Confusion Matrix")
> mn_cfm3_tst
> mn_acc3_tst=round(mn_cfm3_tst$overall[['Accuracy']],4) # Accuracy of predictions with test data
> print(paste("PCA Generalization Phase Accuracy =", mn_acc3_tst))
> # Check for over-fitting. Criteria: Accuracy change from train to test > 25%
> mn_model3_isOF=abs((mn_acc3_tr-mn_acc3_tst)/mn_acc3_tr)
> mn_model3_isOF=round(mn_model3_isOF,4)
> print(paste("Accuracy drop from training data to test data is", mn_model3_isOF*100,"%"))
> if(mn_model3_isOF>0.25) print("Model is over-fitting") else print("Model is not over-fitting")
> # Performance Parameters
> mn_PM3_tst=mn_cfm3_tst$byClass[, c("Balanced Accuracy", "Precision", "Sensitivity", "Specificity", "Recall")]
> print("PCA Generalization-Phase Performance Parameters:")
> mn_PM3_tst
> mn_PMavg3_tst=round(apply(mn_PM3_tst,2,mean),4)
> print("Macro Averages:")
> t(mn_PMavg3_tst)
> mn_prob3_tst=predict(mn_model3, tstdf_pca[, -target], type="probs")
> mn_AUC3_tst=multiclass.roc(tstdf_pca[, target_pca], mn_prob3_tst)
> print(paste("PCA Generalization-Phase AUC:", round(mn_AUC3_tst$auc, 4)))

[1] "PCA Generalization Phase Confusion Matrix"
Confusion Matrix and Statistics

    mn_pred3_tst
       1   2   3   4   5   6   7   8   9  10  11  12
  1  379   3   6   0   0   0   0   0   0   0   0   1
  2    6 303   8   0   0   0   1   2   0   1   0   0
  3    4  14 287   0   0   0   0   0   1   0   0   1
  4    0   0   0 295  99   6   1   0   0   0   0   0
  5    0   0   0  40 370   2   0   0   0   0   0   0
  6    0   0   0   4   1 400   0   1   0   1   1   0
  7    1   0   0   0   1   0  11   1   0   0   1   0
  8    0   0   0   0   0   0   1   6   0   1   0   1
  9    0   0   0   0   0   0   0   0  11   0  12   0
  10   0   0   0   0   0   0   0   0   0  13   1   5
  11   0   0   0   0   0   0   0   0   2   0  17   0
  12   0   0   0   0   0   0   0   0   0   0   1   7
```

```

Overall Statistics

               Accuracy : 0.9005
                 95% CI : (0.8876, 0.9123)
    No Information Rate : 0.2021
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8821

[1] "PCA Generalization Phase Accuracy = 0.9005"

[1] "Accuracy drop from training data to test data is 3.09 %"

[1] "Model is not over-fitting"

[1] "PCA Generalization-Phase Performance Parameters:"
          Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1          0.9833214 0.9742931 0.9717949  0.9948480 0.9717949
Class: 2          0.9689621 0.9439252 0.9468750  0.9910492 0.9468750
Class: 3          0.9718181 0.9348534 0.9534884  0.9901478 0.9534884
Class: 4          0.9084968 0.7356608 0.8702065  0.9467871 0.8702065
Class: 5          0.8814910 0.8980583 0.7855626  0.9774194 0.7855626
Class: 6          0.9881160 0.9803922 0.9803922  0.9958398 0.9803922
Class: 7          0.8919940 0.7333333 0.7857143  0.9982736 0.7857143
Class: 8          0.7993537 0.6666667 0.6000000  0.9987075 0.6000000
Class: 9          0.8902676 0.4782609 0.7857143  0.9948209 0.7857143
Class: 10         0.9049541 0.6842105 0.8125000  0.9974082 0.8125000
Class: 11         0.7571406 0.8947368 0.5151515  0.9991297 0.5151515
Class: 12         0.7331174 0.8750000 0.4666667  0.9995682 0.4666667

[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]            0.8899    0.8166      0.7895      0.9903 0.7895

[1] "PCA Generalization-Phase AUC: 0.9884"
```

```
[1] "Generalization-Phase Performance Parameters of Logistic-Regression Model without PCA, for Comparison:"
          Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1          0.9814862 0.9691517 0.9691517  0.9938208 0.9691517
Class: 2          0.9661471 0.9314642 0.9432177  0.9890765 0.9432177
Class: 3          0.9750119 0.9348534 0.9598662  0.9901575 0.9598662
Class: 4          0.8826182 0.7630923 0.8138298  0.9514066 0.8138298
Class: 5          0.8757307 0.8398058 0.7863636  0.9650978 0.7863636
Class: 6          0.9981865 0.9828431 1.0000000  0.9963731 1.0000000
Class: 7          0.8560639 0.6666667 0.7142857  0.9978420 0.7142857
Class: 8          0.6862086 0.3333333 0.3750000  0.9974171 0.3750000
Class: 9          0.7743417 0.6956522 0.5517241  0.9969592 0.5517241
Class: 10         0.8673397 0.7368421 0.7368421  0.9978374 0.7368421
Class: 11         0.6514600 0.4210526 0.3076923  0.9952278 0.3076923
Class: 12         0.7690151 0.8750000 0.5384615  0.9995686 0.5384615

[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]             0.857    0.7625      0.7247      0.9892 0.7247
```

Now, we implement the dual-group classification method with PCA and examine if it further improves the imbalance issue. The results below imply that the overall accuracy did not change much, but there is an improvement in the class-specific accuracies of imbalanced classes. Nevertheless, in general, the precision for these classes have worsened.

```
> ## Dual-group Classification with PCA ###
> ##########################################
> # Change the labels of classes 1-6 to 0, and classes 7-12 to 1 for training data
> trdf_svm2=trdf_pca # Load the data to a different variable
> trdf_svm2$activity[class16_idx]=rep(0,length(class16_idx)) # Change the classes 1-6 to 0s
> trdf_svm2$activity[class712_idx]=rep(1,length(class712_idx))# Change the classes 7-12 to 1s
> table(trdf_svm2$activity)

   0    1
5177  259
```

```
> # Create the model for group detection
> trdf_svm2$activity=as.factor(trdf_svm2$activity) # Makes it run classification
> svm_model2=svm(activity~.,data=trdf_svm2, probability=TRUE) # Train an SVM model

```

```
> # Partition the training data into two subsets A and B. Subset A contains classes 1-6, and subset B contains 7-12
> trdfA2=trdf_pca[class16_idx,] # Create subset A
> trdfB2=trdf_pca[class712_idx,] # Create subset B
> # Train a classifier on subset A
> trdfA2$activity=as.factor(trdfA2$activity)
> mn_modelA2=multinom(activity~.,trdfA2,maxit=1000)
> # Train a classifier on subset B
> trdfB2$activity=as.factor(trdfB2$activity)
> mn_modelB2=multinom(activity~.,trdfB2,maxit=1000)
```

```
> # Predict using full train data
> svm_pred2_tr = predict(svm_model2,trdf_pca[, -target_pca], type="class") # Predict with the group detector
> svm_pred2_tr_idx0=which(svm_pred2_tr==0) # Get indeces of data to be predicted by classifier A (classes 1-6)
> svm_pred2_tr_idx1=which(svm_pred2_tr==1) # Get indeces of data to be predicted by classifier B (classes 7-12)
> mn_predA2_tr = predict(mn_modelA2, trdf_pca[svm_pred2_tr_idx0, -target_pca], type="class") # Predict the data that belong to subset A with classifier A
> mn_predB2_tr = predict(mn_modelB2, trdf_pca[svm_pred2_tr_idx1, -target_pca], type="class") # Predict the data that belong to subset B with classifier B
> grp_pred2_tr=rep(0,length(mn_predA2_tr)+length(mn_predB2_tr)) # Create a vector to combine the predictions
> grp_pred2_tr[svm_pred2_tr_idx0]=as.numeric(as.character(mn_predA2_tr))
> grp_pred2_tr[svm_pred2_tr_idx1]=as.numeric(as.character(mn_predB2_tr))
>
> grp_cfm2_tr=confusionMatrix(table(trdf_pca[,target_pca], grp_pred2_tr)) # Confusion Matrix for train data
>
> print("Dual-group with PCA Learning-Phase Confusion Matrix")
> grp_cfm2_tr
> grp_acc2_tr=round(grp_cfm2_tr$overall[['Accuracy']],4) # Accuracy of predictions with train data
> print(paste("Dual-group with PCA Learning-Phase Accuracy =", grp_acc2_tr))
> # Performance Parameters
> grp_PM2_tr=grp_cfm2_tr$byClass[, c("Balanced Accuracy", "Precision", "Sensitivity", "Specificity", "Recall")]
> print("Dual-group with PCA Learning-Phase Performance Parameters:")
> grp_PM2_tr
> grp_PMavg2_tr=round(apply(grp_PM2_tr,2,mean),4)
> print("Macro Averages:")
> t(grp_PMavg2_tr)
```

```
[1] "Dual-group with PCA Learning-Phase Confusion Matrix"
Confusion Matrix and Statistics

   grp_pred2_tr
       1    2    3    4    5    6    7    8    9   10   11   12
  1  815    8   14    0    0    0    0    0    0    0    0    0
  2    9  721   22    0    0    0    0    0    0    0    0    0
  3   12   23  645    0    0    0    0    0    0    0    0    0
  4    0    0    0  733  155    3    1    0    0    0    0    0
  5    0    0    0   99  912    0    0    0    0    0    0    0
  6    0    0    0    2    0 1003    0    0    0    0    0    0
  7    0    3    0    1    1    0   27    0    0    0    0    0
  8    0    0    0    0    0    0    0   14    0    0    0    0
  9    0    0    0    0    0    0    0    0   39    0   13    0
  10   0    0    0    0    0    0    0    0    0   33    0    8
  11   0    0    0    0    0    1    0    0    9    0   61    0
  12   0    0    0    0    0    1    0    0    0    6    0   42

Overall Statistics

               Accuracy : 0.9281
                 95% CI : (0.9209, 0.9348)
    No Information Rate : 0.1965
    P-Value [Acc > NIR] : < 2.2e-16
                  Kappa : 0.9149

[1] "Dual-group with PCA Learning-Phase Accuracy = 0.9281"

[1] "Dual-group with PCA Learning-Phase Performance Parameters:"
          Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1          0.9850489 0.9737157 0.9748804  0.9952174 0.9748804
Class: 2          0.9741722 0.9587766 0.9549669  0.9933775 0.9549669
Class: 3          0.9698879 0.9485294 0.9471366  0.9926393 0.9471366
Class: 4          0.9216433 0.8217489 0.8778443  0.9654423 0.8778443
Class: 5          0.9156339 0.9020772 0.8539326  0.9773352 0.8539326
Class: 6          0.9972940 0.9980100 0.9950397  0.9995483 0.9950397
Class: 7          0.9816806 0.8437500 0.9642857  0.9990754 0.9642857
Class: 8          1.0000000 1.0000000 1.0000000  1.0000000 1.0000000
Class: 9          0.9050436 0.7500000 0.8125000  0.9975872 0.8125000
Class: 10         0.9223358 0.8048780 0.8461538  0.9985177 0.8461538
Class: 11         0.9112297 0.8591549 0.8243243  0.9981350 0.8243243
Class: 12         0.9193502 0.8571429 0.8400000  0.9987003 0.8400000
[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]            0.9503    0.8931      0.9076      0.993 0.9076
```

```
> # Predict using test data
> svm_pred2_tst = predict(svm_model2, tstdf_pca[,-target_pca], type="class") # Predict with the group detector
> svm_pred2_tst_idx0=which(svm_pred2_tst==0) # Get indeces of data to be predicted by classifier A (classes 1-6)
> svm_pred2_tst_idx1=which(svm_pred2_tst==1) # Get indeces of data to be predicted by classifier B (classes 7-12)
>
> mn_predA2_tst = predict(mn_modelA2, tstdf_pca[svm_pred2_tst_idx0, -target_pca], type="class") # Predict the data that belong to subset A with classifier A
> mn_predB2_tst = predict(mn_modelB2, tstdf_pca[svm_pred2_tst_idx1, -target_pca], type="class") # Predict the data that belong to subset B with classifier B
>
> grp_pred2_tst=rep(0,length(mn_predA2_tst)+length(mn_predB2_tst)) # Create a vector to combine the predictions
> grp_pred2_tst[svm_pred2_tst_idx0]=as.numeric(as.character(mn_predA2_tst))
> grp_pred2_tst[svm_pred2_tst_idx1]=as.numeric(as.character(mn_predB2_tst))
>
> grp_cfm2_tst=confusionMatrix(table(tstdf_pca[,target_pca], grp_pred2_tst)) # Confusion Matrix for test data
> print("Dual-group with PCA Generalization-Phase Confusion Matrix")
> grp_cfm2_tst
> grp_acc2_tst=round(grp_cfm2_tst$overall[['Accuracy']],4) # Accuracy of predictions with test data
> print(paste("Dual-group with PCA Generalization-Phase Accuracy =", grp_acc2_tst))
> # Check for over-fitting. Criteria: Accuracy change from train to test > 25%
> grp_model2_isOF=abs((grp_acc2_tr-grp_acc2_tst)/grp_acc2_tr)
> grp_model2_isOF=round(grp_model2_isOF,4)
> print(paste("Accuracy drop from training data to test data is",grp_model2_isOF*100,"%"))
> if(grp_model2_isOF>0.25) print("Model is over-fitting") else print("Model is not over-fitting")
> # Performance Parameters
> grp_PM2_tst=grp_cfm2_tst$byClass[, c("Balanced Accuracy", "Precision", "Sensitivity", "Specificity", "Recall")]
> print("Dual-group with PCA Generalization-Phase Performance Parameters:")
> grp_PM2_tst
> grp_PMavg2_tst=round(apply(grp_PM2_tst,2,mean),4)
> print("Macro Averages:")
> t(grp_PMavg2_tst)
```

```
[1] "Dual-group with PCA Generalization-Phase Confusion Matrix"
Confusion Matrix and Statistics

    grp_pred2_tst
       1   2   3   4   5   6   7   8   9  10  11  12
  1  379   3   6   0   1   0   0   0   0   0   0   0
  2    6 307   8   0   0   0   0   0   0   0   0   0
  3    4  14 288   0   1   0   0   0   0   0   0   0
  4    0   0   0 295 100   6   0   0   0   0   0   0
  5    0   0   0  40 370   2   0   0   0   0   0   0
  6    0   0   0   3   1 403   0   0   0   0   1   0
  7    1   0   0   0   4   0   7   3   0   0   0   0
  8    0   1   0   0   0   0   1   7   0   0   0   0
  9    0   0   0   0   0   0   2   0  10   0  11   0
  10   0   0   0   0   0   0   0   0   0  13   1   5
  11   0   0   0   0   0   0   1   0   2   0  16   0
  12   0   0   0   0   1   0   0   0   0   0   1   6

Overall Statistics
             Accuracy : 0.9013
               95% CI : (0.8885, 0.9131)
    No Information Rate : 0.2051
    P-Value [Acc > NIR] : < 2.2e-16
                Kappa : 0.883

[1] "Dual-group with PCA Generalization-Phase Accuracy = 0.9013"

[1] "Accuracy drop from training data to test data is 2.89 %"

[1] "Model is not over-fitting"

[1] "Dual-group with PCA Generalization-Phase Performance Parameters:"
         Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1         0.9833214 0.9742931 0.9717949  0.9948480 0.9717949
Class: 2         0.9688182 0.9563863 0.9446154  0.9930209 0.9446154
Class: 3         0.9721391 0.9381107 0.9536424  0.9906358 0.9536424
Class: 4         0.9097975 0.7356608 0.8727811  0.9468138 0.8727811
Class: 5         0.8756963 0.8980583 0.7740586  0.9773341 0.7740586
Class: 6         0.9889656 0.9877451 0.9805353  0.9973958 0.9805353
Class: 7         0.8164577 0.4666667 0.6363636  0.9965517 0.6363636
Class: 8         0.8495692 0.7777778 0.7000000  0.9991383 0.7000000
Class: 9         0.9138637 0.4347826 0.8333333  0.9943941 0.8333333
Class: 10        0.9987058 0.6842105 1.0000000  0.9974116 1.0000000
Class: 11        0.7660148 0.8421053 0.5333333  0.9986962 0.5333333
Class: 12        0.7722962 0.7500000 0.5454545  0.9991379 0.5454545
```

```

[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]            0.9013    0.7871      0.8122      0.9904 0.8122
```

```
[1] "PCA without Dual-Group Generalization-Phase Performance Parameters for
         Comparison:"
          Balanced Accuracy Precision Sensitivity Specificity Recall
Class: 1          0.9833214 0.9742931 0.9717949  0.9948480 0.9717949
Class: 2          0.9689621 0.9439252 0.9468750  0.9910492 0.9468750
Class: 3          0.9718181 0.9348534 0.9534884  0.9901478 0.9534884
Class: 4          0.9084968 0.7356608 0.8702065  0.9467871 0.8702065
Class: 5          0.8814910 0.8980583 0.7855626  0.9774194 0.7855626
Class: 6          0.9881160 0.9803922 0.9803922  0.9958398 0.9803922
Class: 7          0.8919940 0.7333333 0.7857143  0.9982736 0.7857143
Class: 8          0.7993537 0.6666667 0.6000000  0.9987075 0.6000000
Class: 9          0.8902676 0.4782609 0.7857143  0.9948209 0.7857143
Class: 10         0.9049541 0.6842105 0.8125000  0.9974082 0.8125000
Class: 11         0.7571406 0.8947368 0.5151515  0.9991297 0.5151515
Class: 12         0.7331174 0.8750000 0.4666667  0.9995682 0.4666667

[1] "Macro Averages:"
     Balanced Accuracy Precision Sensitivity Specificity Recall
[1,]            0.8899    0.8166      0.7895      0.9903 0.7895

[1] "PCA Generalization-Phase AUC: 0.9884"
```