Part A - Case study review

Veterans' Administration Spinal Cord Injury Population Case Study

Background:

This study was carried out in order to predict the length of stay (LOS) for a subset of the patient population, specifically, Spinal cord injuries patients.
The reason behind carrying out a special study for this kind of injuries only is the fact that **these injuries are outliers** compared to other injuries when it comes to the time **they need in the hospital (i.e. LOS).**
This prolonged stay means extra costs for the hospital. Therefore, predicting the length of stay for SCI patients will help expecting the costs, which will translate into a better financial management for these situations.

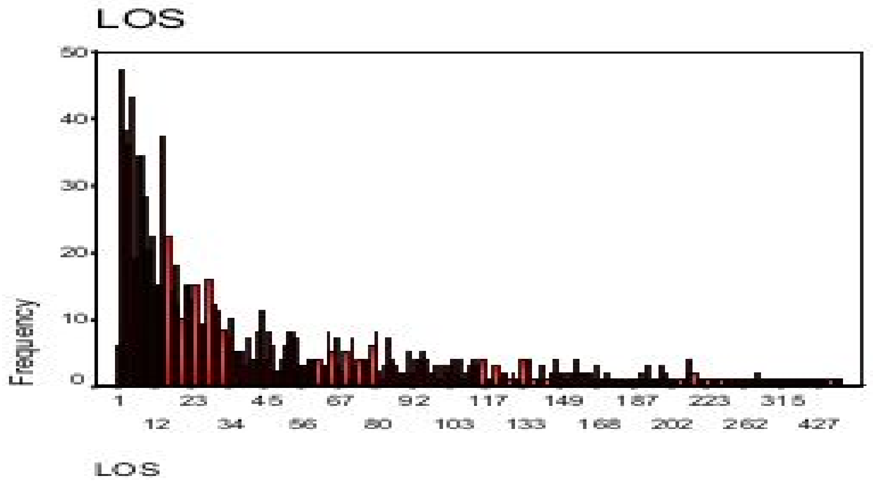
Prediction of Length of stay:

Predicting the length of stay of a patient helps hospitals to accounts for the general costs. It's been already implemented for most hospitals. However, it's been calculated that direct costs for the **first year of a spinal cord injury averages at $223,000** with additional costs of SCI care for **$26,000.**
This is why it is important to consider SCI patients a special case when it comes to costs and length of stay.
The study is carried out for the **Veteran Health Administration(VHA) Hospital**, and its goal is to use data obtained in the past of SCI patients to predict the length of stay for the new SCI patients.

Data description:

The study sample included all
patient episodes of care (patient admissions) in the computerized VistA SCI clinical database from one inpatient SCI unit during the period of study.
The data was about **597 SCI patients with 1107 admissions** between the **years 1989 and 2000.**
The data was cleaned and preprocessed, 161 unique nursing diagnosis was found in the data. These were later clustered into **20 diagnostic categories.** All those were repeated for 11 times at least. Other diagnosis with less than 11 occurrences were put under "Miscellaneous".
The LOS for SCI patients **range**d between **1 day to 770+ days.** The **mean was 55.76** days of stay.

Figure 4: Length of Stay



Neural networks:

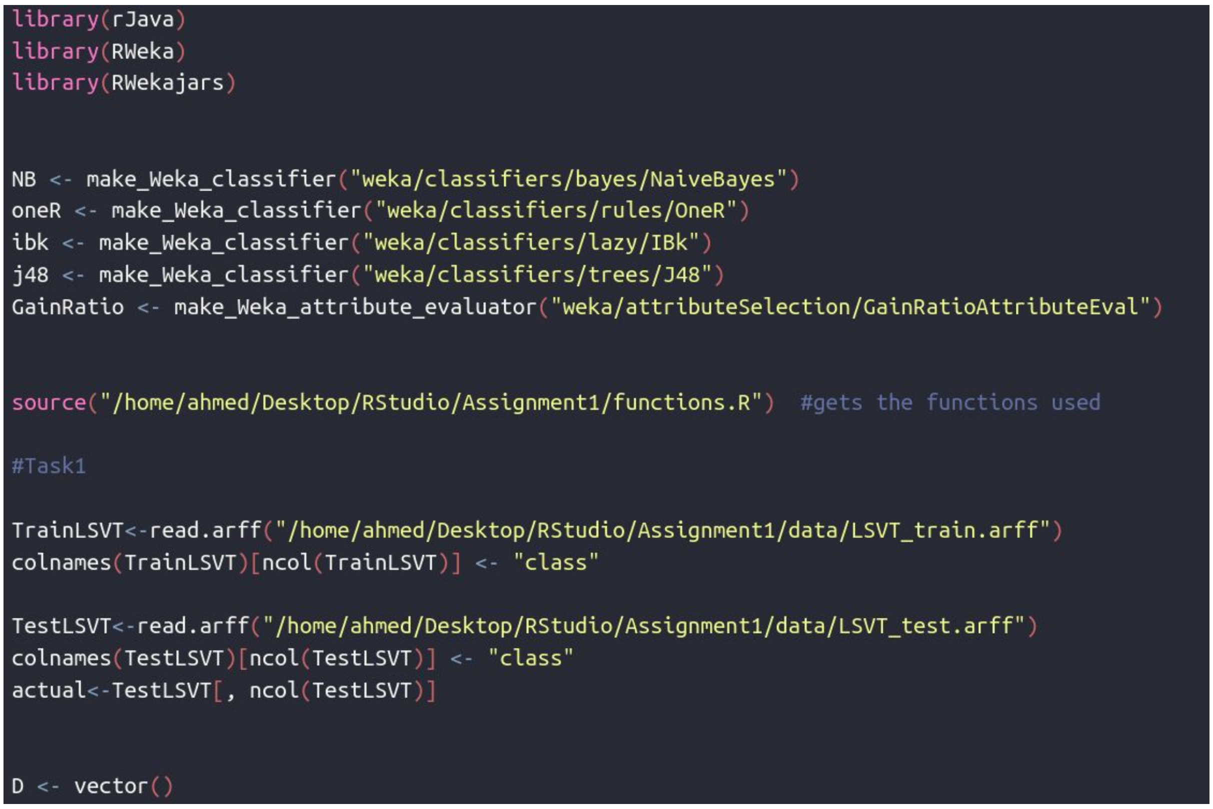
4 types of Artificial neural networks (ANNs) were used on this data set. Artificial neural networks were inspired by observing how the human nervous system works. The human neural network saves information by **strengthening and weakening links between neurons.** And that's how the ANNs work as well with links strengths named **weights**, and nodes as **neurons**.
Neural networks used were; **Multilayer Perceptron MLP,Prune ANN, Dynamic ANN, and Radial basis function ANN (RBF).**
Their performance (Accuracy) was **78%, 78.38%, 77.94%**, and 77.58% respectively.
Features were given weights by the neural networks.
The Prune neural network picks certain features and use them for classification in a process called pruning. **The MLP** uses backpropagation to train the model.
The Dynamic ANN creates an initial model and modifies it by adding and removing hidden units (neurons).
Finally, **the RBF** ANN is similar to a feed-forward function.

Outcomes and benefits:

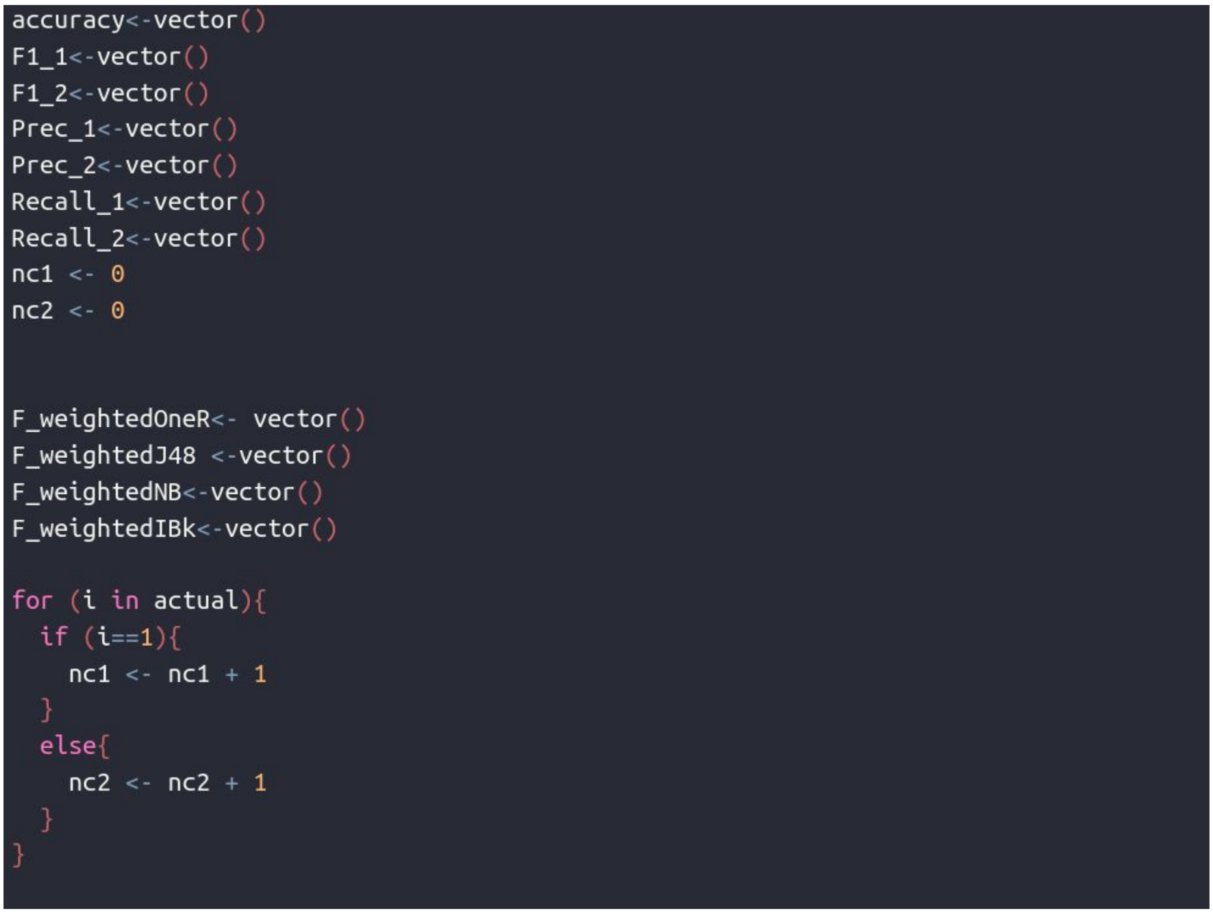
Nursing diagnosis were used because they are the initial diagnosis provided when the patient is admitted. This way when the model is deployed, **the hospital can expect the LOS** with at least a **77% degree** of confidence and therefore eliminating any unnecessary costs. This will reduce the annual costs significantly by expecting the LOS.
SCI costs to individuals and society were calculated to be **$9.7 bil per year.** This can be heavily reduced by knowing how much equipment is needed when patients are first admitted, which will give hospitals more flexibility when it comes to buying **equipment in bulk.**
The data preprocessing was really proficient. The data was visualised first and similar attributes were combined. The ANNs gave a 77%+ accuracy, which is a good accuracy.
I noticed that, when they explained how ANNs work, they did not mention the bias that is usually added to the equation $(x_1w_1+x_2w_2+x_3w_3+\ldots\ldots.x_iw_i)$ bias is usually added to this equation as $((x_1w_1+b_1)+(x_2w_2+b_2)+(x_3w_3+b_3)+\ldots\ldots.(x_iw_i+b_i))$.
The analysts ran ANN scripts and **iterated** through possibilities of **number of layers** and **number of neurons** in each layer to optimise the results for each of the ANNs.

Part B - Task 1

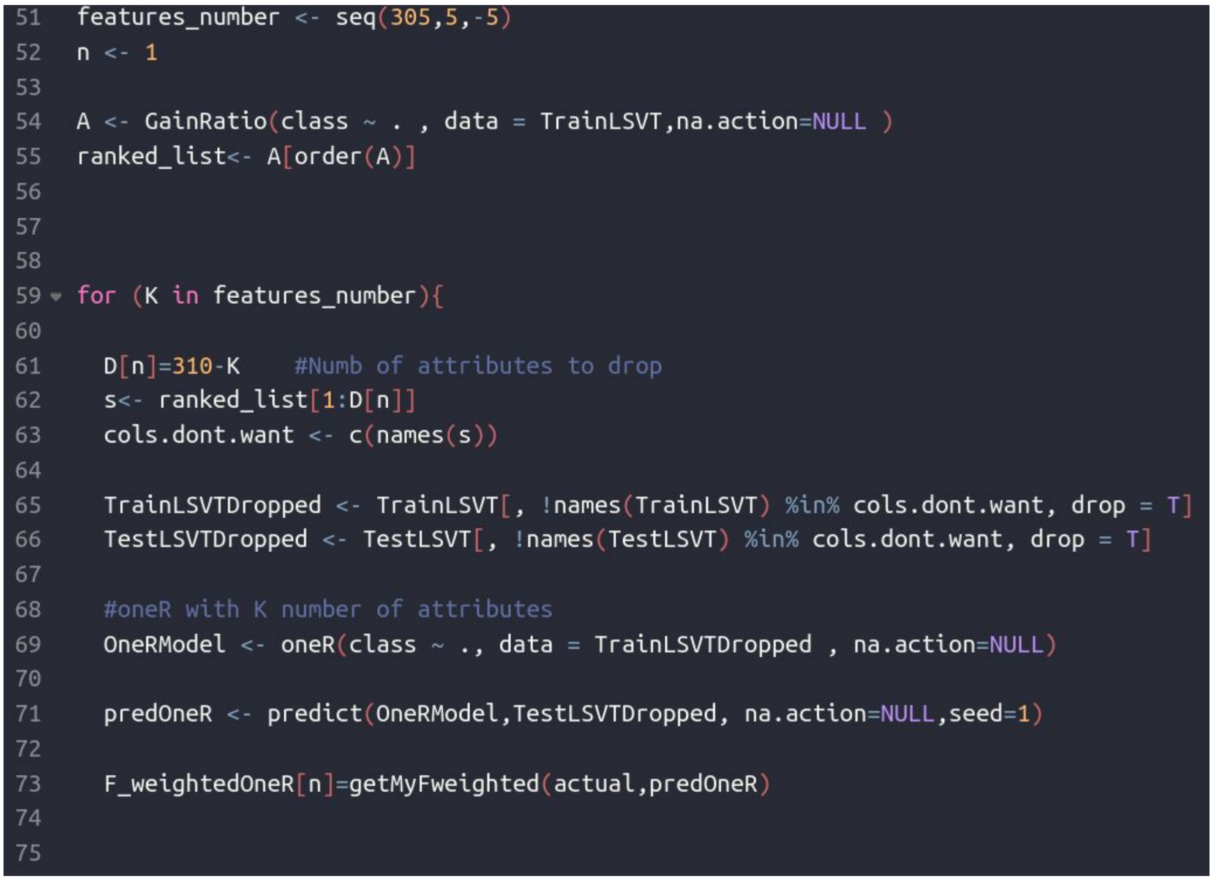
a) The code:



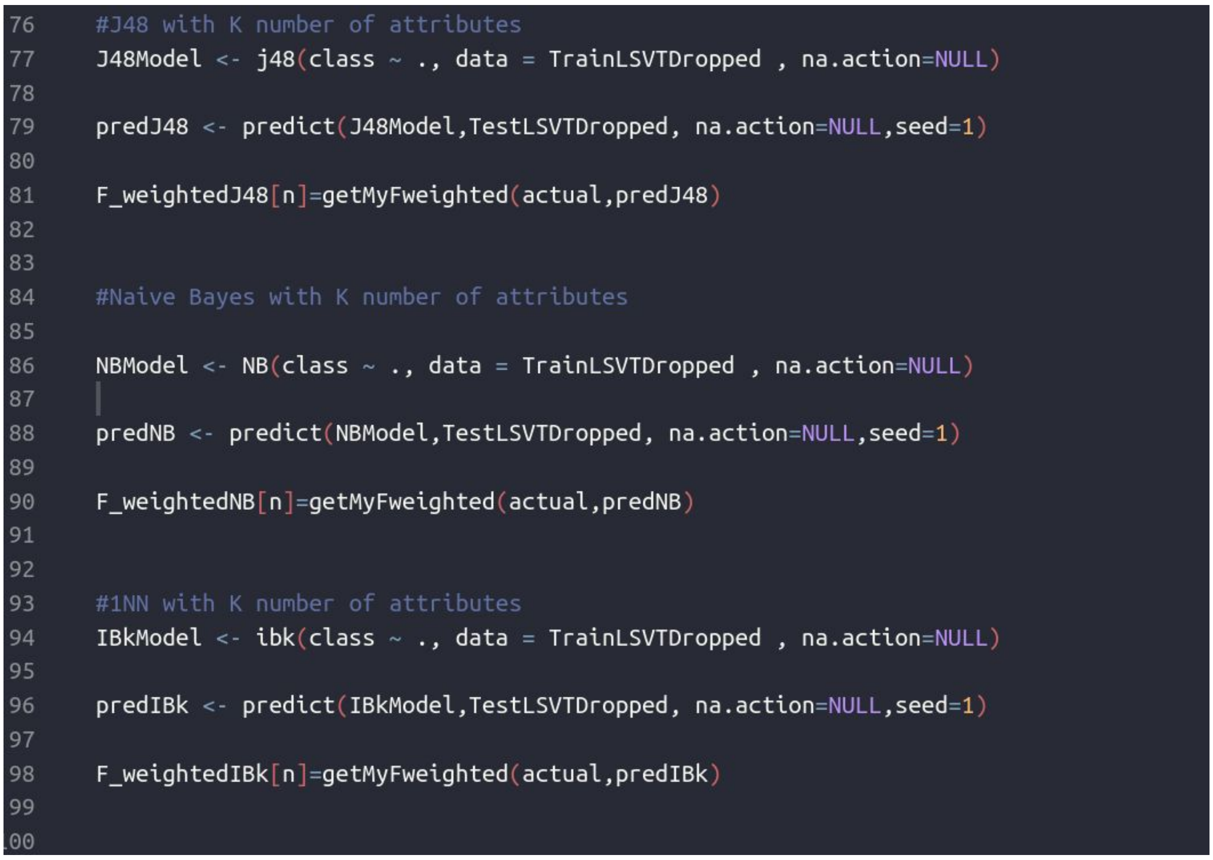
```r
library(rJava)
library(RWeka)
library(RWekajars)


NB <- make_Weka_classifier("weka/classifiers/bayes/NaiveBayes")
oneR <- make_Weka_classifier("weka/classifiers/rules/OneR")
ibk <- make_Weka_classifier("weka/classifiers/lazy/IBk")
j48 <- make_Weka_classifier("weka/classifiers/trees/J48")
GainRatio <- make_Weka_attribute_evaluator("weka/attributeSelection/GainRatioAttributeEval")


source("/home/ahmed/Desktop/RStudio/Assignment1/functions.R")  #gets the functions used

#Task1

TrainLSVT<-read.arff("/home/ahmed/Desktop/RStudio/Assignment1/data/LSVT_train.arff")
colnames(TrainLSVT)[ncol(TrainLSVT)] <- "class"

TestLSVT<-read.arff("/home/ahmed/Desktop/RStudio/Assignment1/data/LSVT_test.arff")
colnames(TestLSVT)[ncol(TestLSVT)] <- "class"
actual<-TestLSVT[, ncol(TestLSVT)]


D <- vector()
```



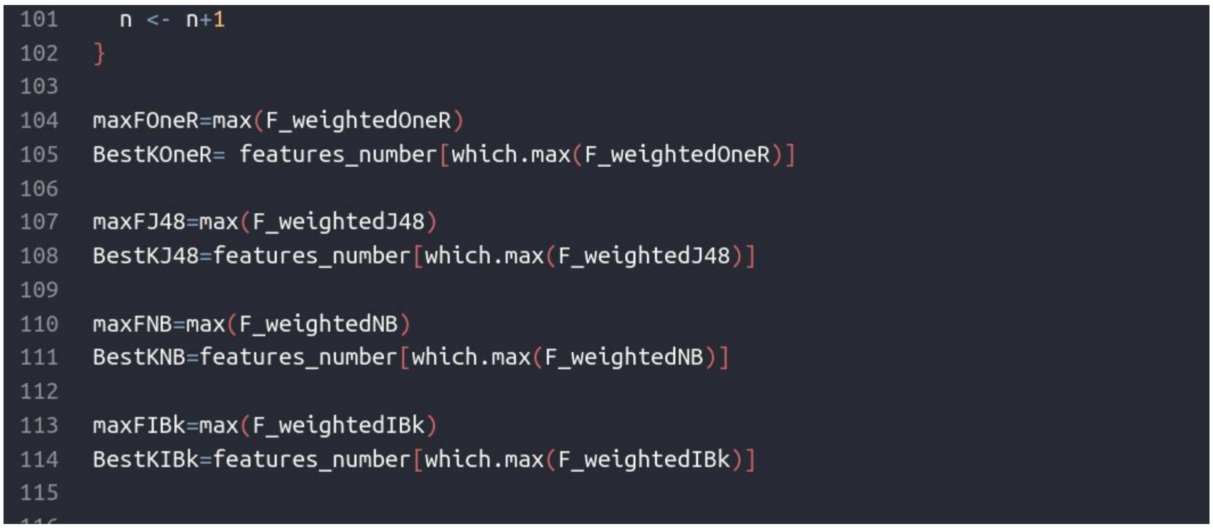
```r
accuracy<-vector()
F1_1<-vector()
F1_2<-vector()
Prec_1<-vector()
Prec_2<-vector()
Recall_1<-vector()
Recall_2<-vector()
nc1 <- 0
nc2 <- 0


F_weightedOneR<- vector()
F_weightedJ48 <-vector()
F_weightedNB<-vector()
F_weightedIBk<-vector()

for (i in actual){
  if (i==1){
    nc1 <- nc1 + 1
  }
  else{
    nc2 <- nc2 + 1
  }
}
```



```r
features_number <- seq(305,5,-5)
n <- 1

A <- GainRatio(class ~ . , data = TrainLSVT,na.action=NULL )
ranked_list<- A[order(A)]



for (K in features_number){

  D[n]=310-K    #Numb of attributes to drop
  s<- ranked_list[1:D[n]]
  cols.dont.want <- c(names(s))

  TrainLSVTDropped <- TrainLSVT[, !names(TrainLSVT) %in% cols.dont.want, drop = T]
  TestLSVTDropped <- TestLSVT[, !names(TestLSVT) %in% cols.dont.want, drop = T]

  #oneR with K number of attributes
  OneRModel <- oneR(class ~ ., data = TrainLSVTDropped , na.action=NULL)

  predOneR <- predict(OneRModel,TestLSVTDropped, na.action=NULL,seed=1)

  F_weightedOneR[n]=getMyFweighted(actual,predOneR)


```



```r
  #J48 with K number of attributes
  J48Model <- j48(class ~ ., data = TrainLSVTDropped , na.action=NULL)

  predJ48 <- predict(J48Model,TestLSVTDropped, na.action=NULL,seed=1)

  F_weightedJ48[n]=getMyFweighted(actual,predJ48)


  #Naive Bayes with K number of attributes

  NBModel <- NB(class ~ ., data = TrainLSVTDropped , na.action=NULL)

  predNB <- predict(NBModel,TestLSVTDropped, na.action=NULL,seed=1)

  F_weightedNB[n]=getMyFweighted(actual,predNB)


  #1NN with K number of attributes
  IBkModel <- ibk(class ~ ., data = TrainLSVTDropped , na.action=NULL)

  predIBk <- predict(IBkModel,TestLSVTDropped, na.action=NULL,seed=1)

  F_weightedIBk[n]=getMyFweighted(actual,predIBk)


```



```
  n <- n+1
}

maxFOneR=max(F_weightedOneR)
BestKOneR= features_number[which.max(F_weightedOneR)]

maxFJ48=max(F_weightedJ48)
BestKJ48=features_number[which.max(F_weightedJ48)]

maxFNB=max(F_weightedNB)
BestKNB=features_number[which.max(F_weightedNB)]

maxFIBk=max(F_weightedIBk)
BestKIBk=features_number[which.max(F_weightedIBk)]

```

Code explanation:

This code is for iterating through K values (numbers of features kept) from **305** descending to **5** with **intervals of 5.**
At first, it imports functions from the functions file which contains the following functions:

1. **getMyFweighted(actualData,PredData):**
Which takes in the actual and the predicted data by the model and returns F weighted value as a result.
It is used in all of the tasks.

2. **dropMyCols(BestK,dataset):**
It takes in the K value and the data set and returns the set after keeping K number of features.
It is used in task 3 because I didn't want to repeat code for dropping the best K value for each model before working on B and Z.

3. **rebalanceMyData:**
It just uses the resample method of rebalancing to rebalance dataset provided.
It is used in task 3.

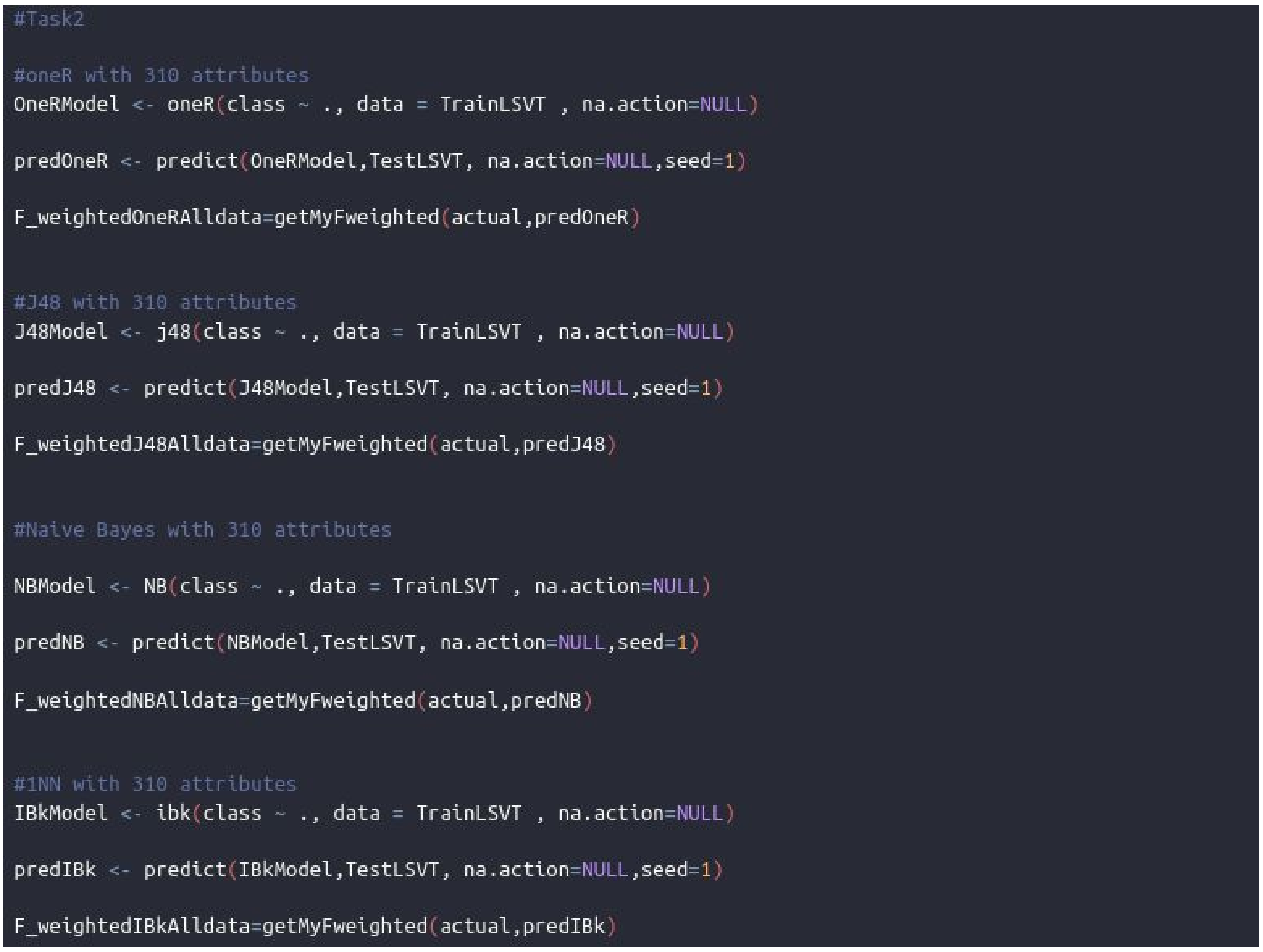
I stored D values(**number of features dropped**) in the vector, named the last column in the test and train datasets as "class".

To avoid repeating the same code, I made a vector for each of the 4 classifiers to save the **F weighted values** (example : F_weightedOneR<- vector()).
In each iteration, the F weighted value is saved into those vectors.
This way I covered part b without repeating all the code.

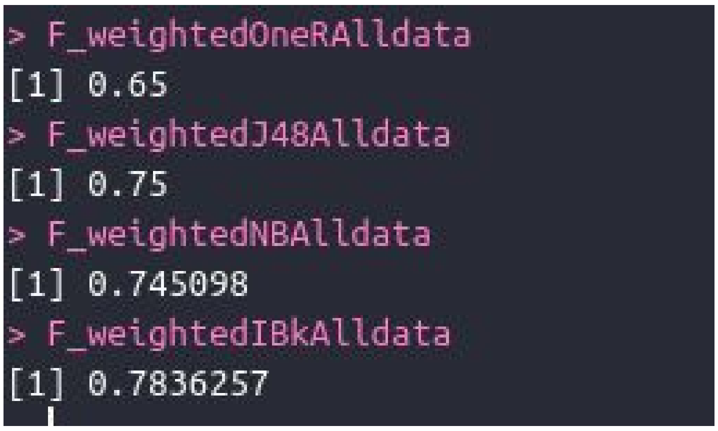
After getting the best F weighted value for each of the classifiers by applying the max() function to them, I used the **features_number[which.max(F_weightedOneR)]** to get the **best K value for each classifier.**

Task 2:

a) F weighted values before any change:

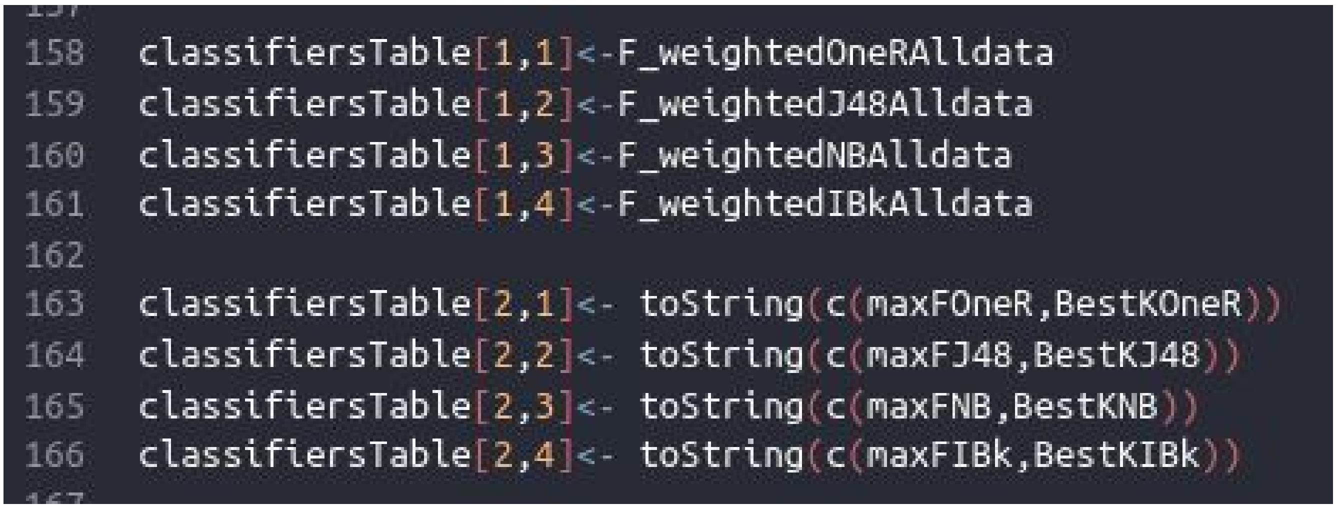


```
#Task2

#oneR with 310 attributes
OneRModel <- oneR(class ~ ., data = TrainLSVT , na.action=NULL)

predOneR <- predict(OneRModel,TestLSVT, na.action=NULL,seed=1)

F_weightedOneRAlldata=getMyFweighted(actual,predOneR)


#J48 with 310 attributes
J48Model <- j48(class ~ ., data = TrainLSVT , na.action=NULL)

predJ48 <- predict(J48Model,TestLSVT, na.action=NULL,seed=1)

F_weightedJ48Alldata=getMyFweighted(actual,predJ48)


#Naive Bayes with 310 attributes

NBModel <- NB(class ~ ., data = TrainLSVT , na.action=NULL)

predNB <- predict(NBModel,TestLSVT, na.action=NULL,seed=1)

F_weightedNBAlldata=getMyFweighted(actual,predNB)


#1NN with 310 attributes
IBkModel <- ibk(class ~ ., data = TrainLSVT , na.action=NULL)

predIBk <- predict(IBkModel,TestLSVT, na.action=NULL,seed=1)

F_weightedIBkAlldata=getMyFweighted(actual,predIBk)
```

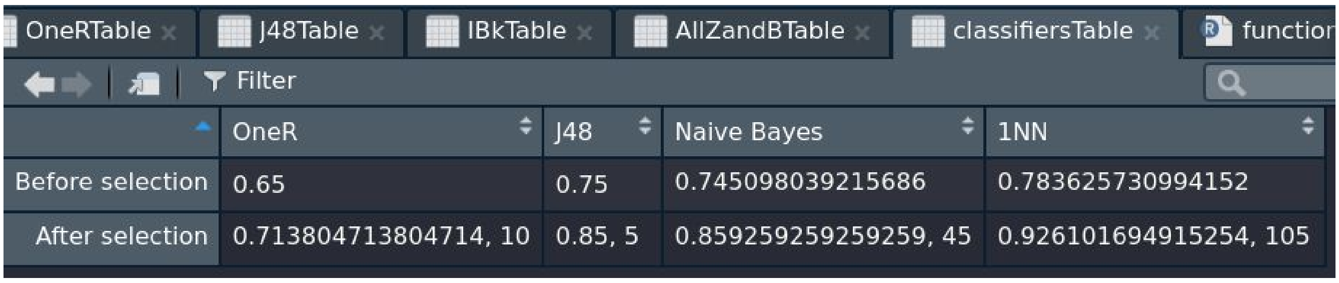


```
> F_weightedOneRAlldata
[1] 0.65
> F_weightedJ48Alldata
[1] 0.75
> F_weightedNBAlldata
[1] 0.745098
> F_weightedIBkAlldata
[1] 0.7836257
```

b) Creating the table:



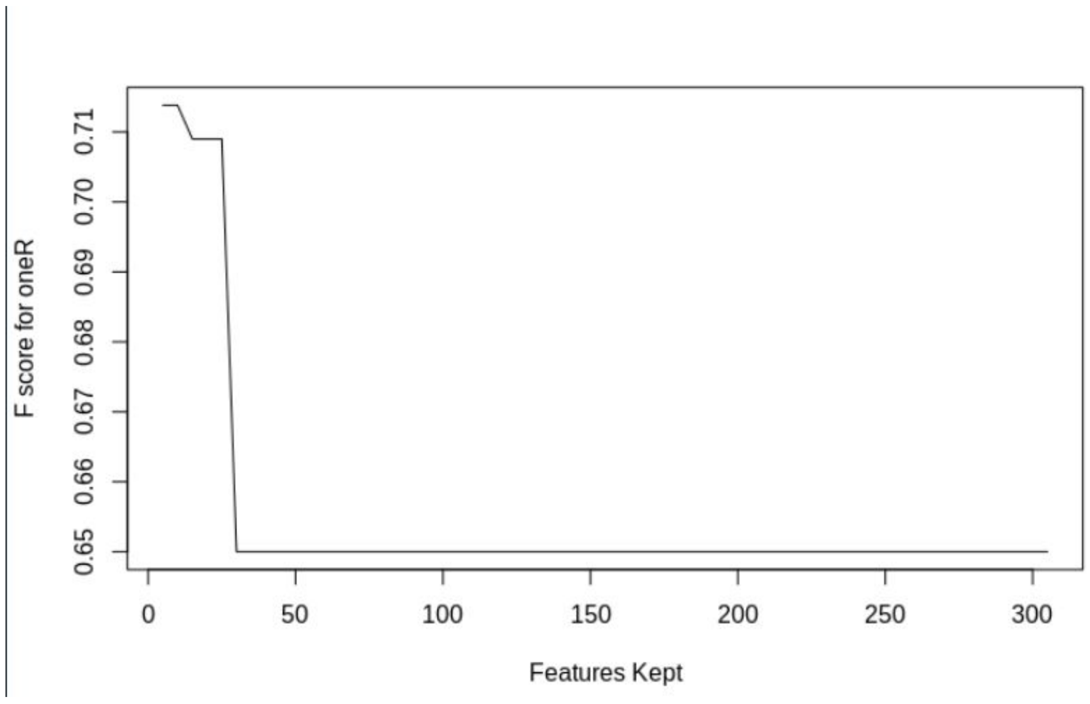
```
classifiersTable[1,1]<-F_weightedOneRAlldata
classifiersTable[1,2]<-F_weightedJ48Alldata
classifiersTable[1,3]<-F_weightedNBAlldata
classifiersTable[1,4]<-F_weightedIBkAlldata

classifiersTable[2,1]<- toString(c(maxFOneR,BestKOneR))
classifiersTable[2,2]<- toString(c(maxFJ48,BestKJ48))
classifiersTable[2,3]<- toString(c(maxFNB,BestKNB))
classifiersTable[2,4]<- toString(c(maxFIBk,BestKIBk))
```



	OneR	J48	Naive Bayes	1NN
Before selection	0.65	0.75	0.745098039215686	0.783625730994152
After selection	0.713804713804714, 10	0.85, 5	0.859259259259259, 45	0.926101694915254, 105

c)

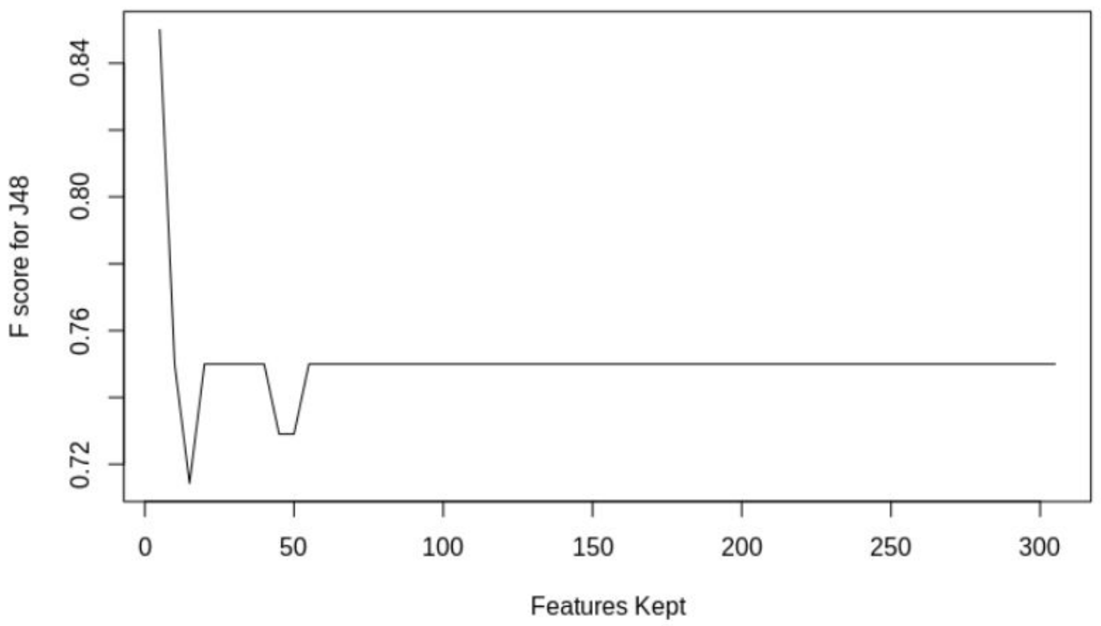
One R:



From this chart, we can notice that as the number of features kept decreases, the F-score becomes better. This is because of the nature of the One R classifier. It takes one attribute into consideration when training. The attribute selection process decreases the number of attributes available, which makes the One R classifier's job easier as it's picking the attributes with best gain rate. Which means the best attributes that can help classifying the test sample with the least error.

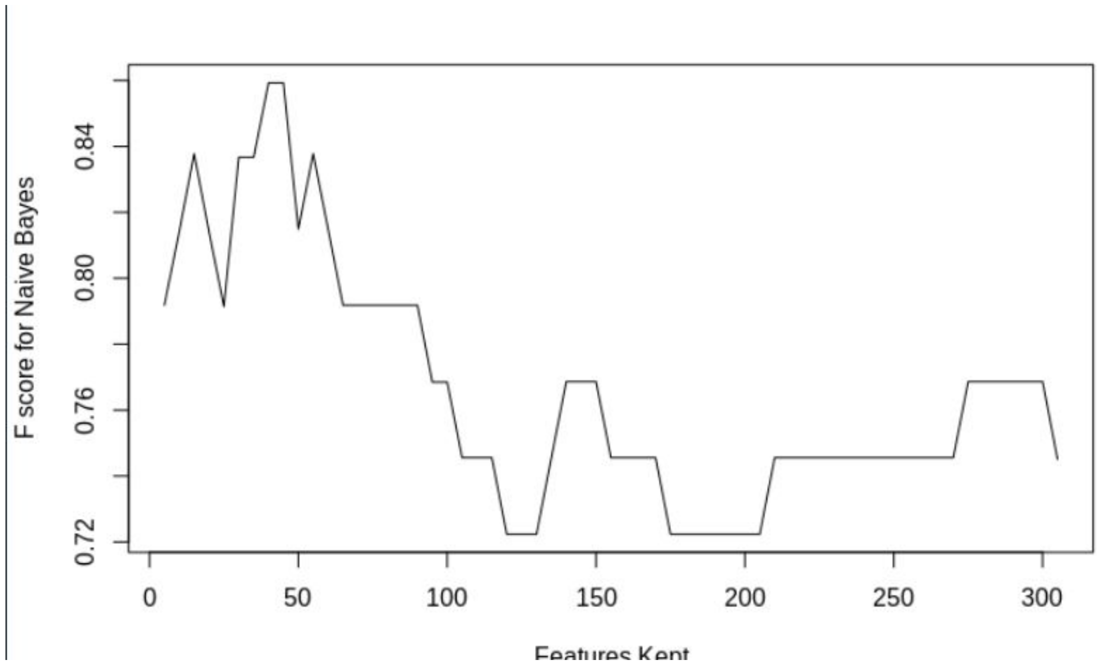
However, with attributes at more than 35, the results are the same as the same attribute is being picked by the one R classifier for classification.

J48 (Decision tree):



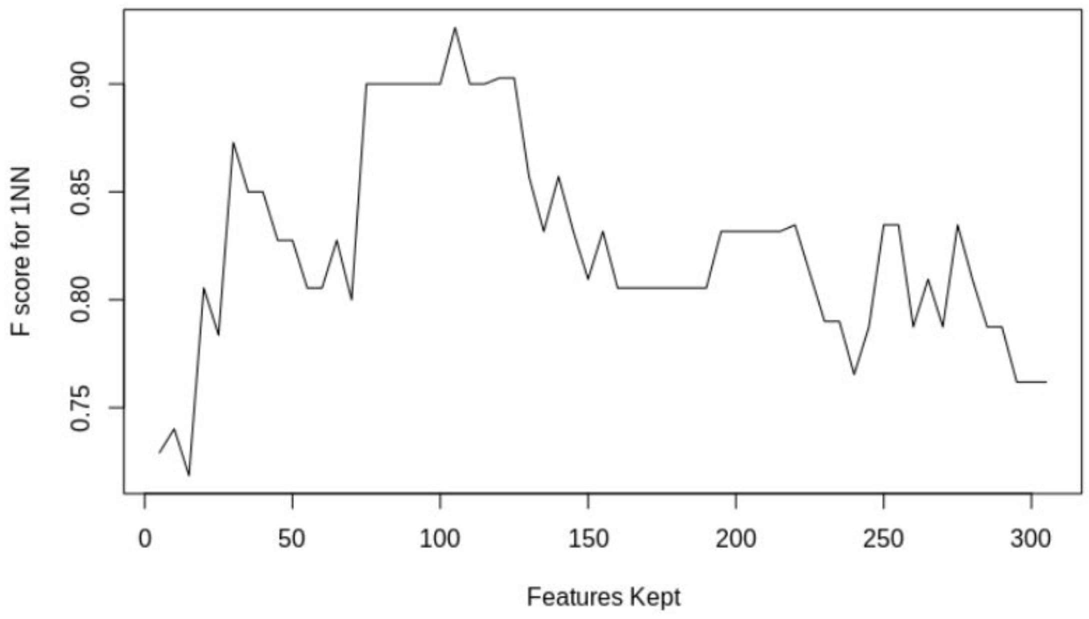
For J48, generally, the F-score peaks when we pick a low number (5, 10,or 15) of attributes. This is because of the nature of J48, where large number of attributes leads to overfitting the model (performs better on the training set). In addition to that, Gain rate is highly relevant to how decision trees work. Which is why when it is used, it enhances the performance improves significantly.

Naive Bayes:



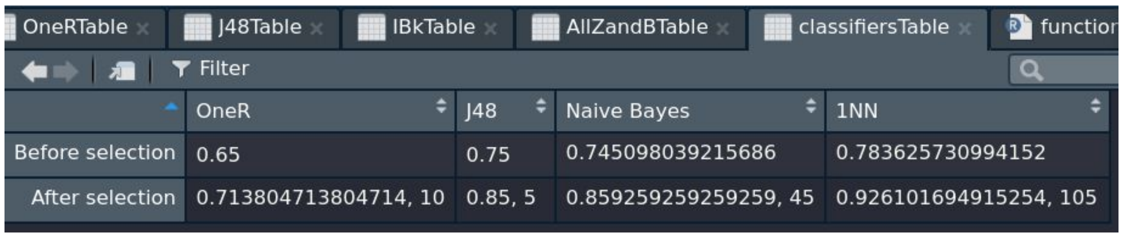
When using Naive Bayes, this classifier depends on the probabilities of different events happening together.
With a high number of features (Events), which creates a lot of unnecessary probabilities. This is why we can see that decreasing the number of attributes significantly changed the F-score for the better.
However, when the number of features decreases significantly, the classifier's performance decreases as it needs to consider the right number of events to get the best performance.

IBk (or 1NN):



For 1 Nearest neighbor, 105 attributes kept is where its F-score peaks. A low number of features is not enough (underfitting) as the classifier needs some more attributes to determine which attributes fall on class 1 or 2.
Too many attributes cause overfitting where it performs not as good on the testing set.

d)

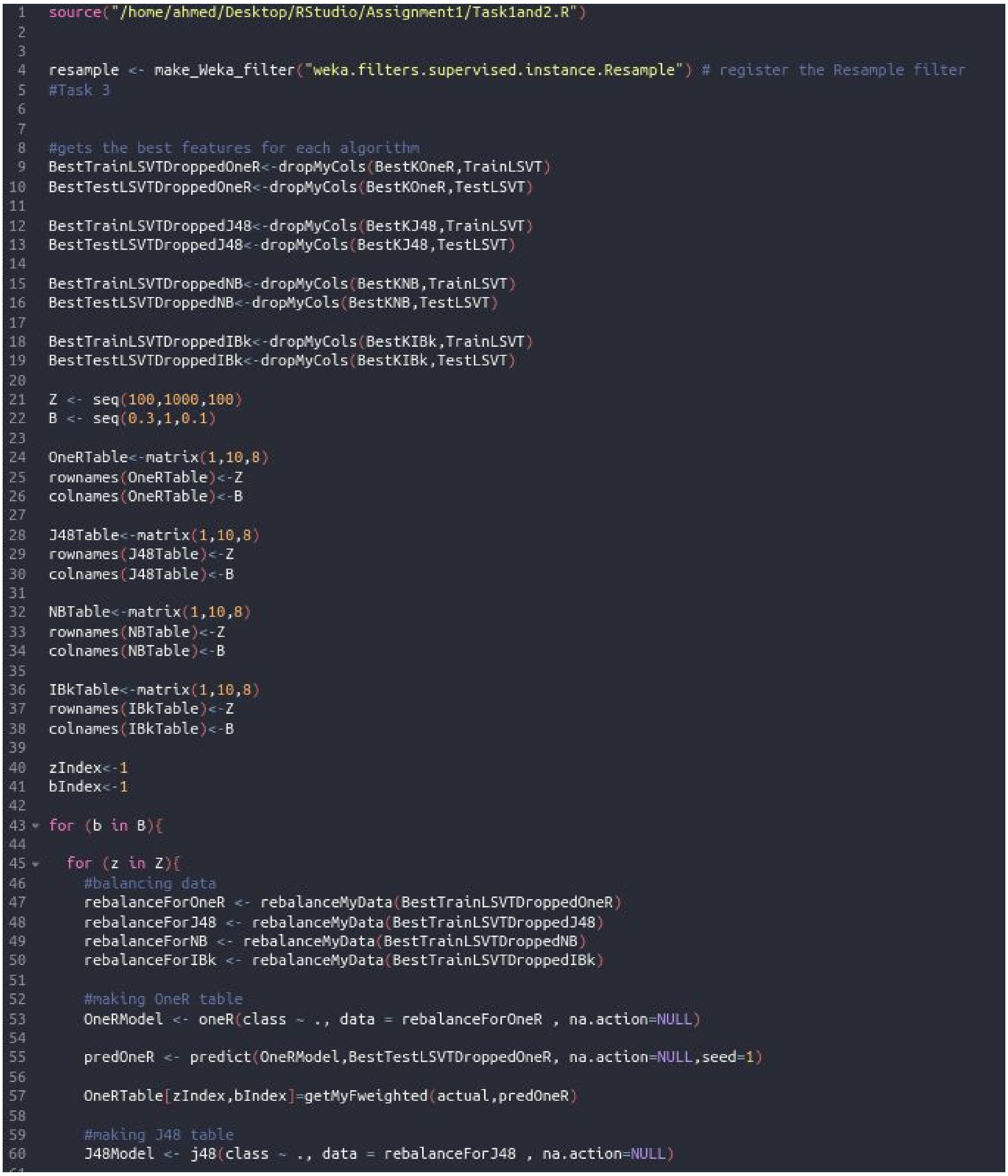


	OneR	J48	Naive Bayes	1NN
Before selection	0.65	0.75	0.745098039215686	0.783625730994152
After selection	0.713804713804714, 10	0.85, 5	0.859259259259259, 45	0.926101694915254, 105

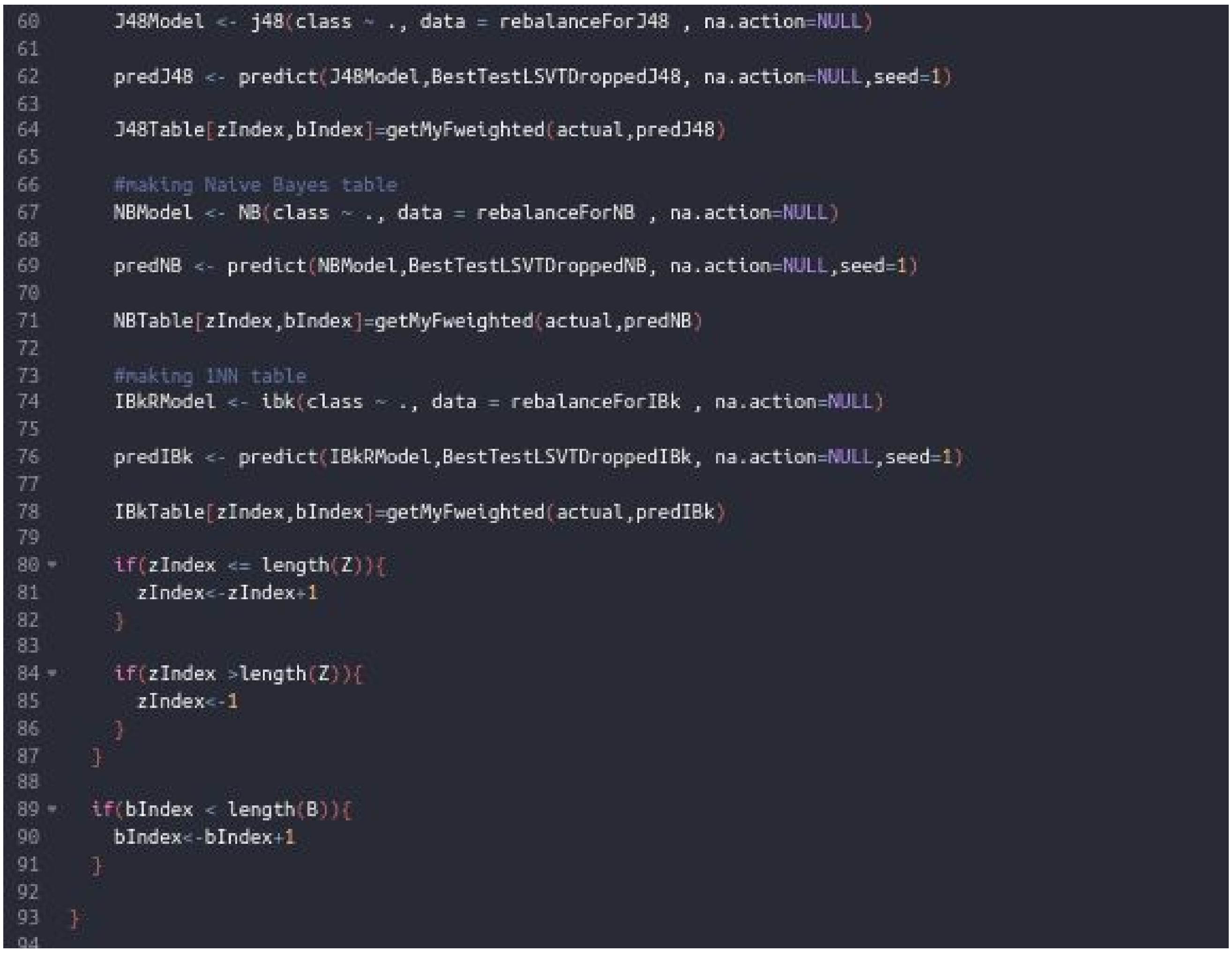
IBk performs the best after feature selection with 105 attributes left and an **F weighted** of **0.9261.**

Task 3:

a)

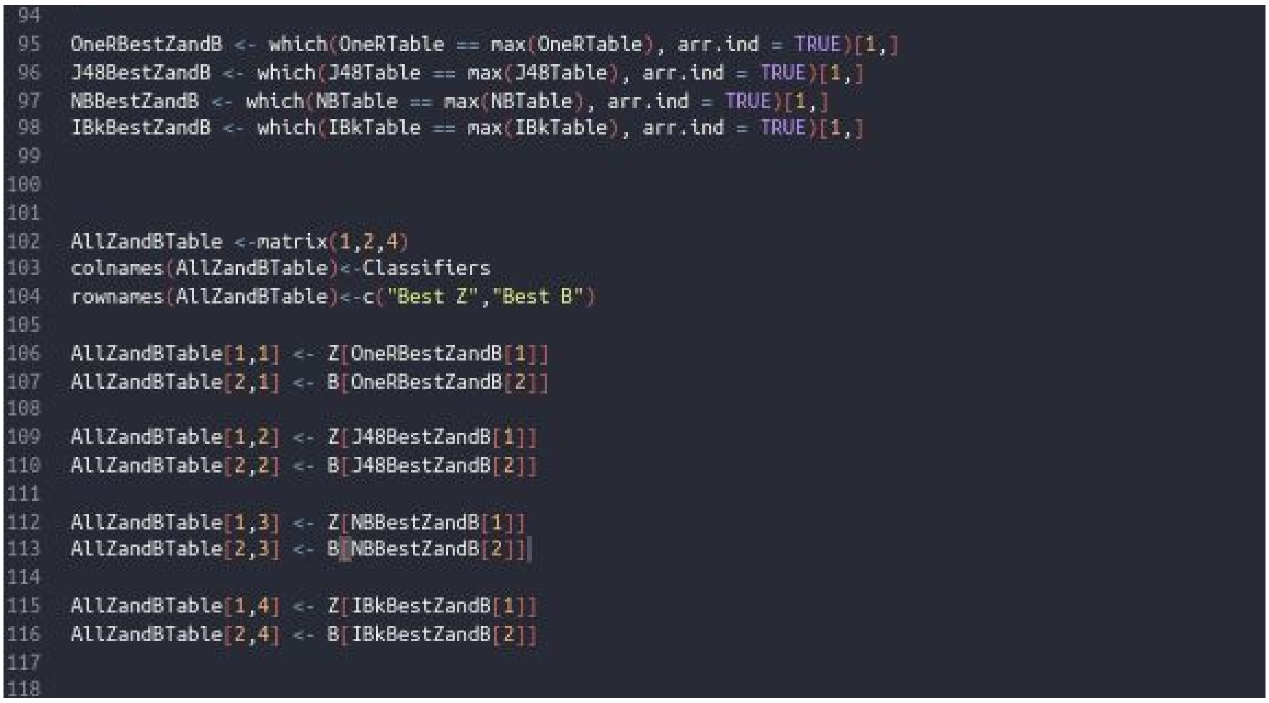


```
source("/home/ahmed/Desktop/RStudio/Assignment1/Task1and2.R")


resample <- make_Weka_filter("weka.filters.supervised.instance.Resample") # register the Resample filter
#Task 3


#gets the best features for each algorithm
BestTrainLSVTDroppedOneR<-dropMyCols(BestKOneR,TrainLSVT)
BestTestLSVTDroppedOneR<-dropMyCols(BestKOneR,TestLSVT)

BestTrainLSVTDroppedJ48<-dropMyCols(BestKJ48,TrainLSVT)
BestTestLSVTDroppedJ48<-dropMyCols(BestKJ48,TestLSVT)

BestTrainLSVTDroppedNB<-dropMyCols(BestKNB,TrainLSVT)
BestTestLSVTDroppedNB<-dropMyCols(BestKNB,TestLSVT)

BestTrainLSVTDroppedIBk<-dropMyCols(BestKIBk,TrainLSVT)
BestTestLSVTDroppedIBk<-dropMyCols(BestKIBk,TestLSVT)

Z <- seq(100,1000,100)
B <- seq(0.3,1,0.1)

OneRTable<-matrix(1,10,8)
rownames(OneRTable)<-Z
colnames(OneRTable)<-B

J48Table<-matrix(1,10,8)
rownames(J48Table)<-Z
colnames(J48Table)<-B

NBTable<-matrix(1,10,8)
rownames(NBTable)<-Z
colnames(NBTable)<-B

IBkTable<-matrix(1,10,8)
rownames(IBkTable)<-Z
colnames(IBkTable)<-B

zIndex<-1
bIndex<-1

for (b in B){

  for (z in Z){
    #balancing data
    rebalanceForOneR <- rebalanceMyData(BestTrainLSVTDroppedOneR)
    rebalanceForJ48 <- rebalanceMyData(BestTrainLSVTDroppedJ48)
    rebalanceForNB <- rebalanceMyData(BestTrainLSVTDroppedNB)
    rebalanceForIBk <- rebalanceMyData(BestTrainLSVTDroppedIBk)

    #making OneR table
    OneRModel <- oneR(class ~ ., data = rebalanceForOneR , na.action=NULL)

    predOneR <- predict(OneRModel,BestTestLSVTDroppedOneR, na.action=NULL,seed=1)

    OneRTable[zIndex,bIndex]=getMyFweighted(actual,predOneR)

    #making J48 table
    J48Model <- j48(class ~ ., data = rebalanceForJ48 , na.action=NULL)
```



```
      J48Model <- j48(class ~ ., data = rebalanceForJ48 , na.action=NULL)

      predJ48 <- predict(J48Model,BestTestLSVTDroppedJ48, na.action=NULL,seed=1)

      J48Table[zIndex,bIndex]=getMyFweighted(actual,predJ48)

      #making Naive Bayes table
      NBModel <- NB(class ~ ., data = rebalanceForNB , na.action=NULL)

      predNB <- predict(NBModel,BestTestLSVTDroppedNB, na.action=NULL,seed=1)

      NBTable[zIndex,bIndex]=getMyFweighted(actual,predNB)

      #making 1NN table
      IBkRModel <- ibk(class ~ ., data = rebalanceForIBk , na.action=NULL)

      predIBk <- predict(IBkRModel,BestTestLSVTDroppedIBk, na.action=NULL,seed=1)

      IBkTable[zIndex,bIndex]=getMyFweighted(actual,predIBk)

      if(zIndex <= length(Z)){
        zIndex<-zIndex+1
      }

      if(zIndex >length(Z)){
        zIndex<-1
      }
    }

    if(bIndex < length(B)){
      bIndex<-bIndex+1
    }

  }
```

b)



```
OneRBestZandB <- which(OneRTable == max(OneRTable), arr.ind = TRUE)[1,]
J48BestZandB <- which(J48Table == max(J48Table), arr.ind = TRUE)[1,]
NBBestZandB <- which(NBTable == max(NBTable), arr.ind = TRUE)[1,]
IBkBestZandB <- which(IBkTable == max(IBkTable), arr.ind = TRUE)[1,]



AllZandBTable <-matrix(1,2,4)
colnames(AllZandBTable)<-Classifiers
rownames(AllZandBTable)<-c("Best Z","Best B")

AllZandBTable[1,1] <- Z[OneRBestZandB[1]]
AllZandBTable[2,1] <- B[OneRBestZandB[2]]

AllZandBTable[1,2] <- Z[J48BestZandB[1]]
AllZandBTable[2,2] <- B[J48BestZandB[2]]

AllZandBTable[1,3] <- Z[NBBestZandB[1]]
AllZandBTable[2,3] <- B[NBBestZandB[2]]

AllZandBTable[1,4] <- Z[IBkBestZandB[1]]
AllZandBTable[2,4] <- B[IBkBestZandB[2]]
```

J48:

	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
100	0.792766373411535	0.792766373411535	0.87683615819209	0.854111405835544	0.854111405835544	0.83170890188434	0.83170890188434	0.83170890188434
200	0.87683615819209	0.87683615819209	0.83170890188434	0.778305084745763	0.778305084745763	0.671604938271605	0.765432098765432	0.606802721088435
300	0.666666666666667	0.666666666666667	0.666666666666667	0.666666666666667	0.639376218323587	0.639376218323587	0.639376218323587	0.639376218323587
400	0.666666666666667	0.666666666666667	0.666666666666667	0.666666666666667	0.639376218323587	0.639376218323587	0.692998955067921	0.610963748894783
500	0.692998955067921	0.610963748894783	0.610963748894783	0.610963748894783	0.718518518518519	0.7	0.7	0.7
600	0.639376218323587	0.639376218323587	0.639376218323587	0.729039548022599	0.66951770016631	0.66951770016631	0.66951770016631	0.66951770016631
700	0.598537095088819	0.598537095088819	0.729039548022599	0.729039548022599	0.666666666666667	0.729039548022599	0.729039548022599	0.729039548022599
800	0.598537095088819	0.666666666666667	0.666666666666667	0.666666666666667	0.598537095088819	0.598537095088819	0.598537095088819	0.598537095088819
900	0.666666666666667	0.598537095088819	0.598537095088819	0.598537095088819	0.598537095088819	0.598537095088819	0.598537095088819	0.598537095088819
1000	0.624691358024691	0.624691358024691	0.624691358024691	0.624691358024691	0.624691358024691	0.624691358024691	0.624691358024691	0.624691358024691

IBk:

	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
100	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392
200	0.872891423140888	0.85	0.85	0.85	0.85	0.85	0.85	0.87683615819209
300	0.87683615819209	0.87683615819209	0.87683615819209	0.87683615819209	0.87683615819209	0.87683615819209	0.87683615819209	0.854111405835544
400	0.872891423140888	0.872891423140888	0.872891423140888	0.872891423140888	0.872891423140888	0.872891423140888	0.872891423140888	0.872891423140888
500	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254
600	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.902740937223696	0.902740937223696	0.902740937223696	0.902740937223696
700	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254
800	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254
900	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254
1000	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254	0.926101694915254

NB:

	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
100	0.883343477378778	0.883343477378778	0.883343477378778	0.883343477378778	0.883343477378778	0.883343477378778	0.883343477378778	0.883343477378778
200	0.860576923076923	0.860576923076923	0.860576923076923	0.837789661319073	0.837789661319073	0.883343477378778	0.860576923076923	0.860576923076923
300	0.860576923076923	0.860576923076923	0.860576923076923	0.837789661319073	0.860576923076923	0.814901960784314	0.860576923076923	0.883343477378778
400	0.837789661319073	0.814901960784314	0.860576923076923	0.860576923076923	0.860576923076923	0.860576923076923	0.860576923076923	0.814901960784314
500	0.883343477378778	0.883343477378778	0.883343477378778	0.859259259259259	0.883343477378778	0.883343477378778	0.883343477378778	0.883343477378778
600	0.883343477378778	0.860576923076923	0.860576923076923	0.860576923076923	0.883343477378778	0.859259259259259	0.859259259259259	0.859259259259259
700	0.860576923076923	0.860576923076923	0.883343477378778	0.883343477378778	0.860576923076923	0.837789661319073	0.837789661319073	0.837789661319073
800	0.883343477378778	0.860576923076923	0.837789661319073	0.837789661319073	0.837789661319073	0.837789661319073	0.860576923076923	0.837789661319073
900	0.860576923076923	0.837789661319073	0.837789661319073	0.837789661319073	0.837789661319073	0.837789661319073	0.837789661319073	0.837789661319073
1000	0.883343477378778	0.883343477378778	0.837789661319073	0.837789661319073	0.837789661319073	0.837789661319073	0.837789661319073	0.837789661319073

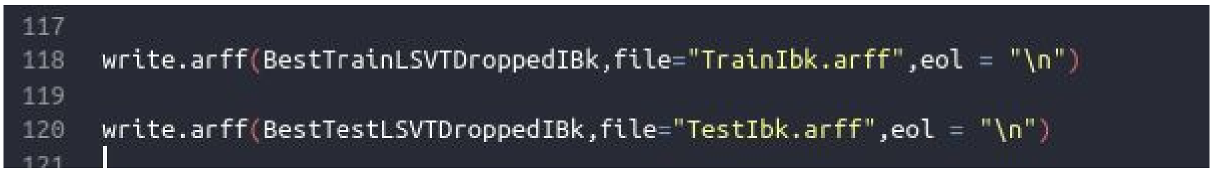
OneR:

	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
100	0.743355650233313	0.743355650233313	0.743355650233313	0.743355650233313	0.743355650233313	0.743355650233313	0.743355650233313	0.698752228163993
200	0.740229885057471	0.740229885057471	0.740229885057471	0.740229885057471	0.740229885057471	0.778305084745763	0.778305084745763	0.537037037037037
300	0.514285714285714	0.514285714285714	0.714285714285714	0.581730769230769	0.714285714285714	0.581730769230769	0.581730769230769	0.581730769230769
400	0.603367823087847	0.603367823087847	0.624691358024691	0.603367823087847	0.55	0.55	0.55	0.603367823087847
500	0.708222811671087	0.55	0.577777777777778	0.687459389213775	0.531977401129944	0.687459389213775	0.687459389213775	0.687459389213775
600	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392
700	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392
800	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392
900	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392	0.805481874447392
1000	0.805481874447392	0.7	0.7	0.7	0.7	0.7	0.679774011299435	0.679774011299435

	OneR	J48	Naive Bayes	1NN
Best Z	600	200	100	500
Best B	0.3	0.3	0.3	0.3

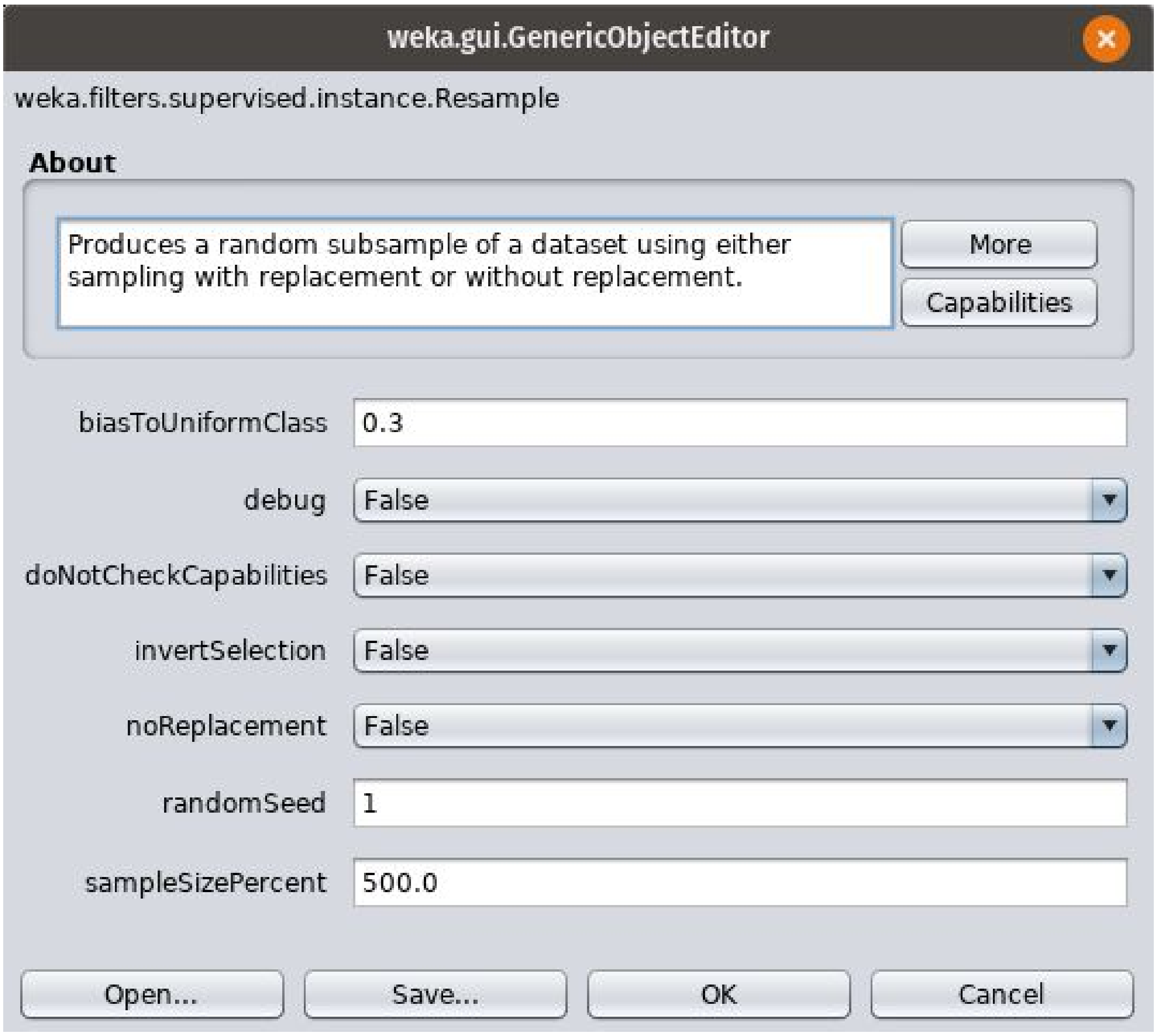
c)

1) The “Visualize threshold curve” was not active when we supplied the testing set because the number of features was different that the training set where we dropped some features.
To go around that problem, I wrote some R code to save the best performing kept features for both the training and testing set. The best performing classifier was IBk or 1NN, and the best K for it was 105.



```
write.arff(BestTrainLSVTDroppedIBk,file="TrainIbk.arff",eol = "\n")

write.arff(BestTestLSVTDroppedIBk,file="TestIbk.arff",eol = "\n")
```

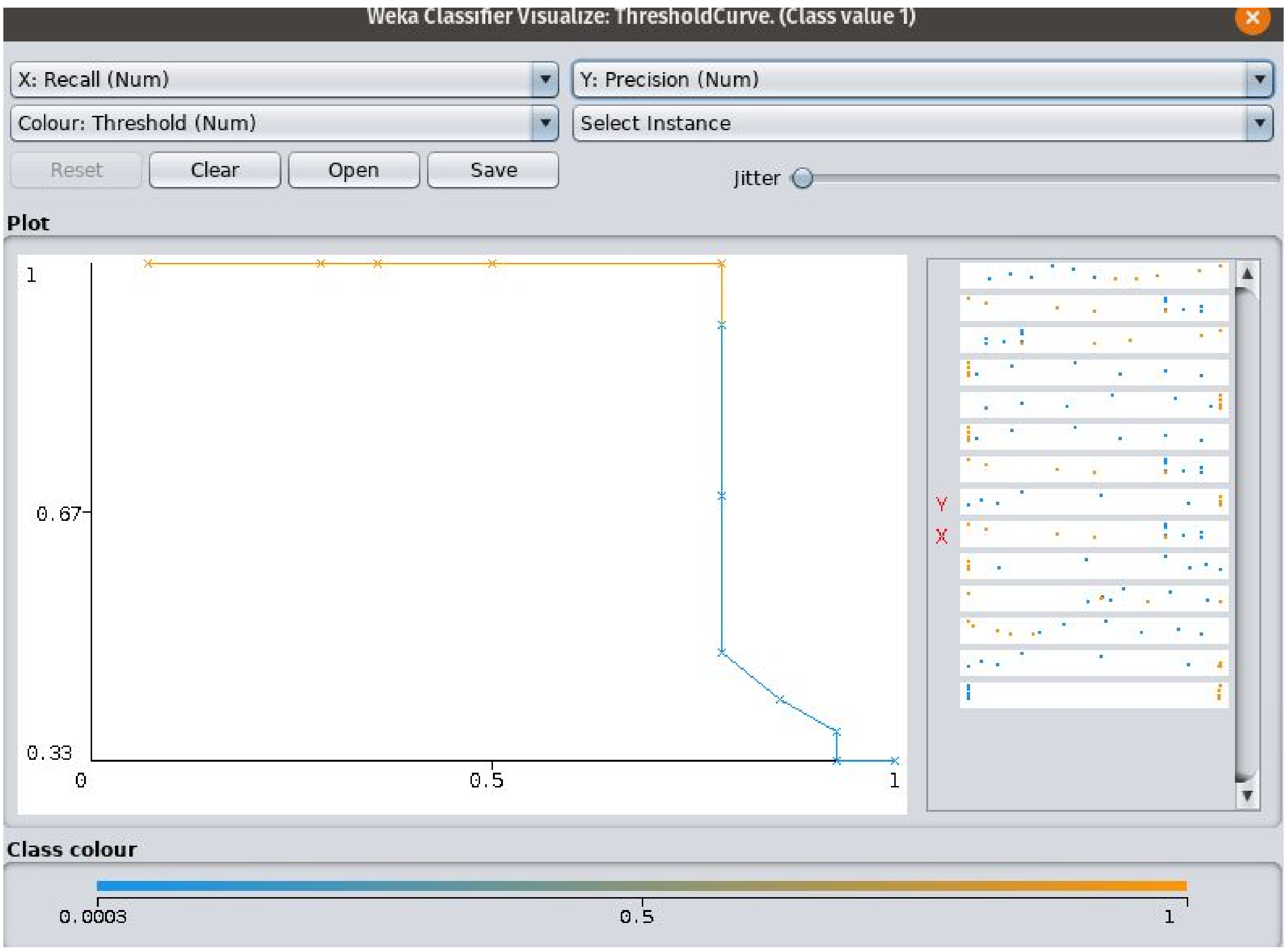
Then, I applied the resample rebalancing method with a biasToUniformClass of 0.3 and a sampleSizePercent of 500 using weka on the training set only.



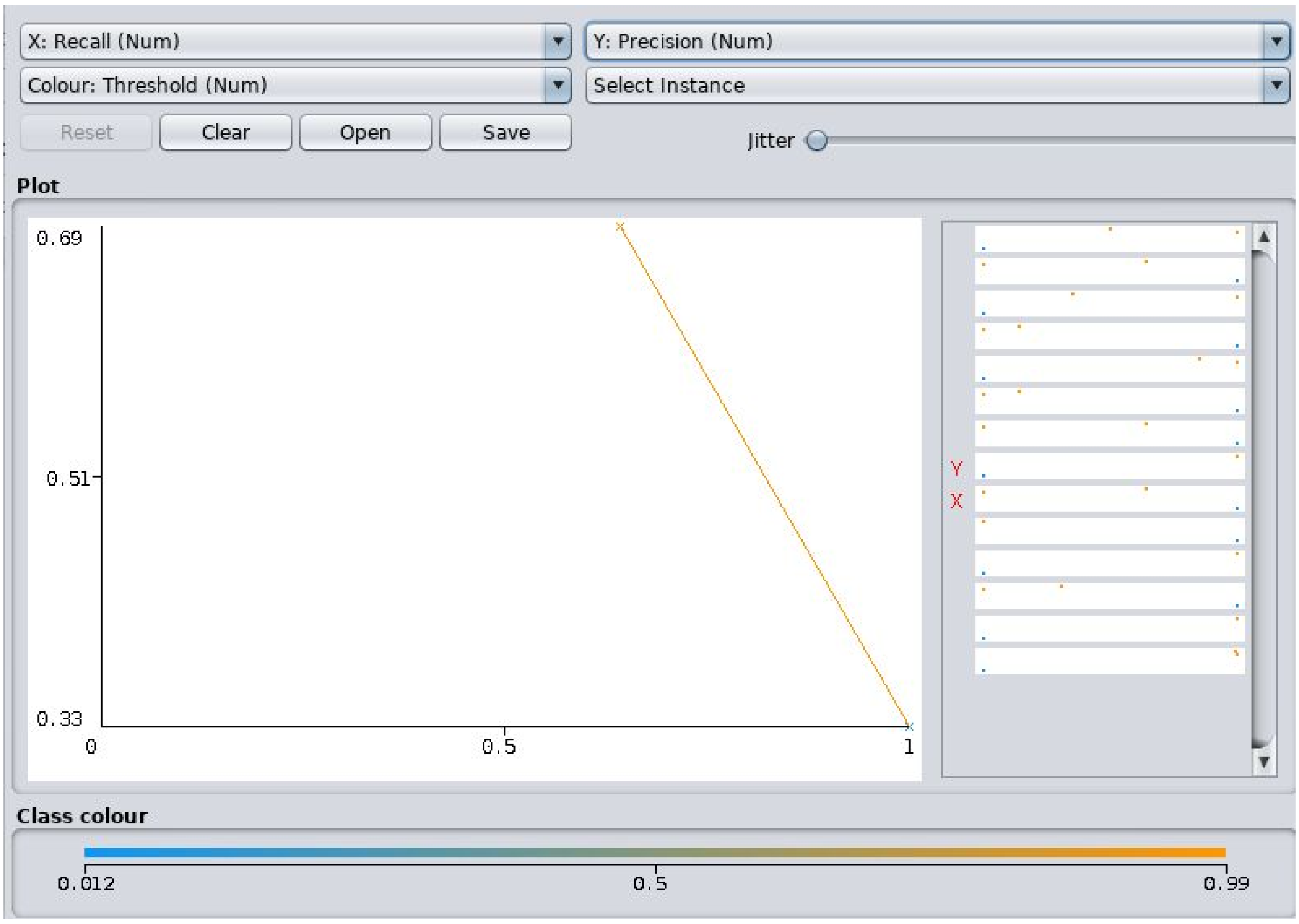
weka.filters.supervised.instance.Resample

Produces a random subsample of a dataset using either sampling with replacement or without replacement.

biasToUniformClass: 0.3
debug: False
doNotCheckCapabilities: False
invertSelection: False
noReplacement: False
randomSeed: 1
sampleSizePercent: 500.0

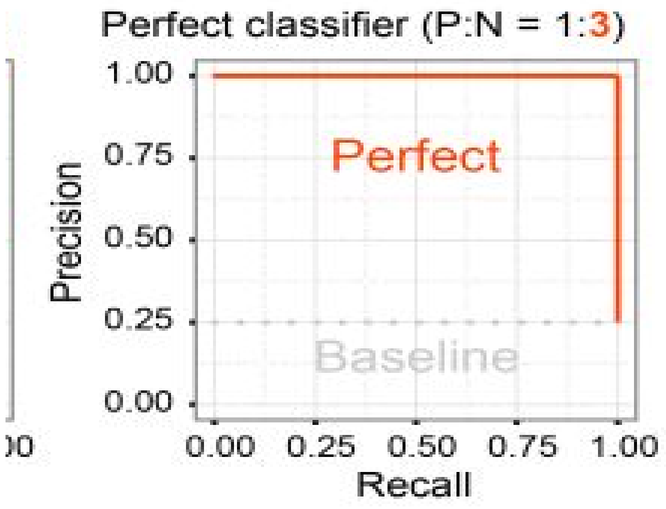

Weka Classifier Visualize: ThresholdCurve. (Class value 1)
X: Recall (Num)
Y: Precision (Num)
Colour: Threshold (Num)
Select Instance
Reset
Clear
Open
Save
Jitter
Plot
1
0.67
0.33
0
0.5
1
Y
X
Class colour
0.0003
0.5
1

2)


X: Recall (Num)
Y: Precision (Num)
Colour: Threshold (Num)
Select Instance
Reset
Clear
Open
Save
Jitter
Plot
0.69
0.51
0.33
0
0.5
1
Y
X
Class colour
0.012
0.5
0.99

d)

Precision is a measurement of the percentage of true positives (in this case Classified as class 1 and is actually a class 1) to the total number of instances classified as 1.
Recall is a measurement of the percentage of true positives (in this case Classified as class 1 and is actually a class 1) to the total number of instances that are actually class 1.

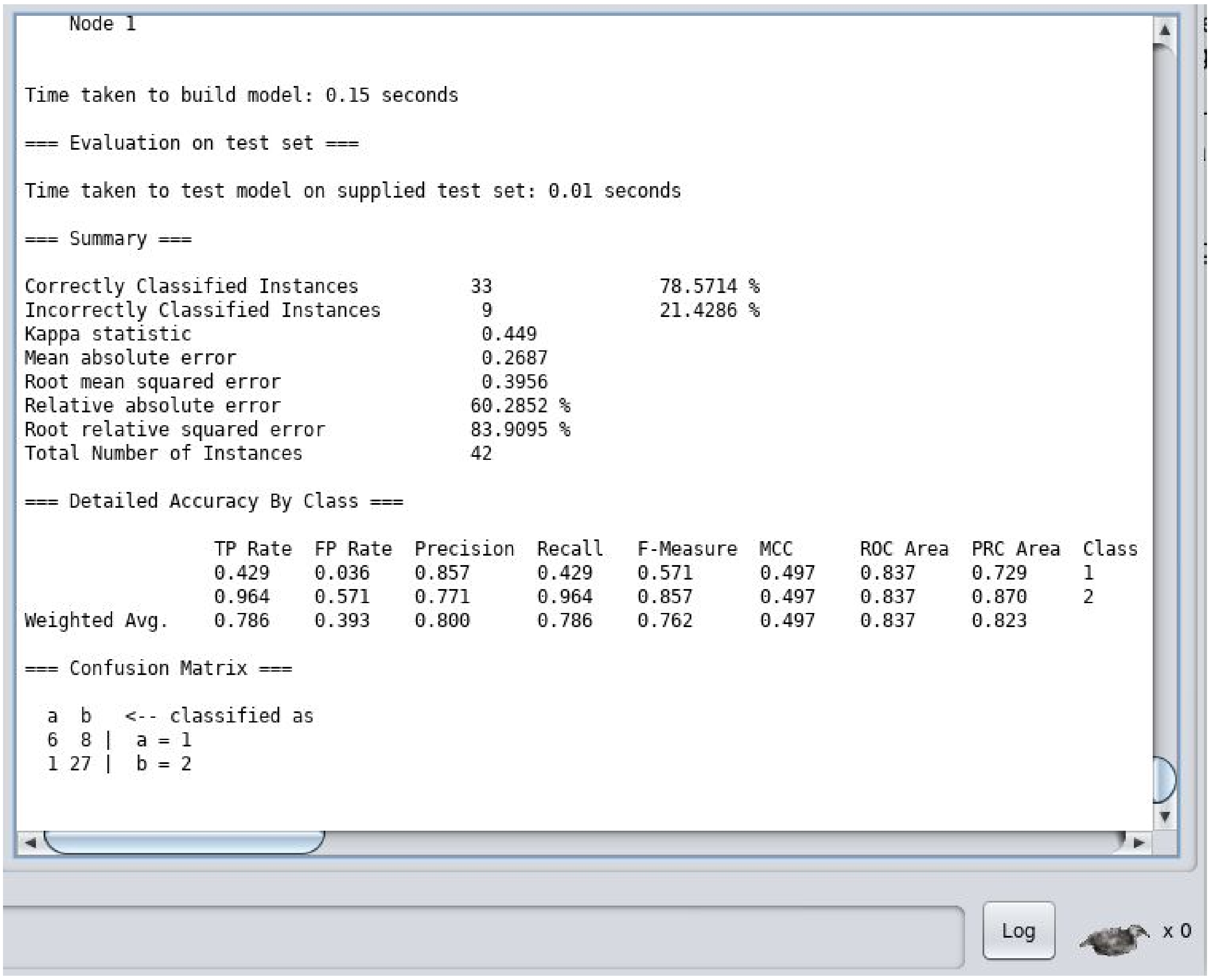


This is what a perfect PRC should look like when the classifier is performing perfectly. In the perfect PRC the precision and recall stay at their best all the time.
A good classifier tries to keep precision at a good level when increasing the recall.

In the first curve in part c, we can see that the classifier's precision is dropping fast as the recall increases, while on the second curve (i.e. after feature selection and rebalancing is carried out), the precision is not decreasing at the same rate when increasing recall.
This means that the classifier is now closer to being a perfect classifier than it was before.

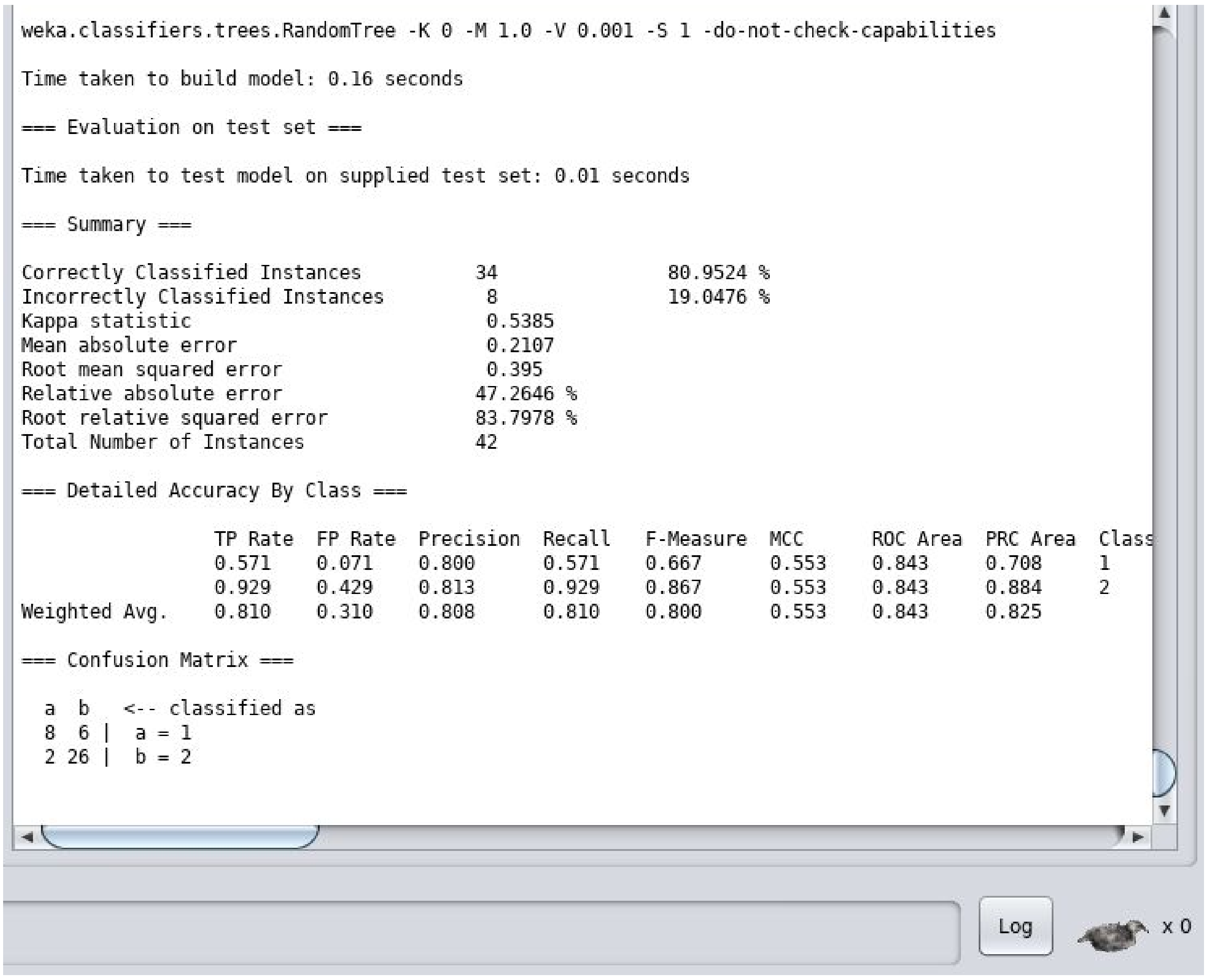
Task 4:

Multilayer perceptron MLP:



```
    Node 1


Time taken to build model: 0.15 seconds

=== Evaluation on test set ===

Time taken to test model on supplied test set: 0.01 seconds

=== Summary ===

Correctly Classified Instances          33               78.5714 %
Incorrectly Classified Instances         9               21.4286 %
Kappa statistic                          0.449
Mean absolute error                      0.2687
Root mean squared error                  0.3956
Relative absolute error                 60.2852 %
Root relative squared error             83.9095 %
Total Number of Instances               42

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.429    0.036    0.857      0.429    0.571      0.497    0.837     0.729     1
                 0.964    0.571    0.771      0.964    0.857      0.497    0.837     0.870     2
Weighted Avg.    0.786    0.393    0.800      0.786    0.762      0.497    0.837     0.823

=== Confusion Matrix ===

  a  b   <-- classified as
  6  8 |  a = 1
  1 27 |  b = 2
```

Random forest:



```
weka.classifiers.trees.RandomTree -K 0 -M 1.0 -V 0.001 -S 1 -do-not-check-capabilities

Time taken to build model: 0.16 seconds

=== Evaluation on test set ===

Time taken to test model on supplied test set: 0.01 seconds

=== Summary ===

Correctly Classified Instances          34               80.9524 %
Incorrectly Classified Instances         8               19.0476 %
Kappa statistic                          0.5385
Mean absolute error                      0.2107
Root mean squared error                  0.395
Relative absolute error                 47.2646 %
Root relative squared error             83.7978 %
Total Number of Instances               42

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.571    0.071    0.800      0.571    0.667      0.553    0.843     0.708     1
                 0.929    0.429    0.813      0.929    0.867      0.553    0.843     0.884     2
Weighted Avg.    0.810    0.310    0.808      0.810    0.800      0.553    0.843     0.825

=== Confusion Matrix ===

  a  b   <-- classified as
  8  6 |  a = 1
  2 26 |  b = 2
```

a)

F-weighted for One R = 0.65
F-weighted for J48 = 0.75
F-weighted for Naive Bayes = 0.745
F-weighted for IBk = 0.784
F-weighted for meta with MLP = 0.762
F-weighted for meta with Random forest - 0.8

When using MLP as a stacking classifier, the F weighted value was higher than all of each of the classifiers on their own except for the the IBk model that had an F weighted of 0.784.
The improvement can be explained by the nature of stacking, which is having a classifier (MLP) validating the predictions of the ensemble classifiers (J48, IBk, and Naive Bayes). However, IBk seems to be the best classifier when performing on its own. As a result of this, the IBk performs better alone than when in an ensemble, because it is more decisive that way.

When using Random forest, the meta-learner performed better than all the other classifiers. This is probably because Random forest is an Ensemble itself and having two layers of ensemble, one with the three classifiers and one as the Random forest (the validator) improves the results.

b)

the meta-learner choice had a significant impact on the results. Random forest performed better. Which is probably because the size of the data set is not that large. Performance can certainly be refined for both meta-learners by tuning how many layers and neurons the MLP has and how many branches the Random forest has.