Business Data Mining (IDS 572)

Random Forest

Source: R and Data Mining: Examples and Case Studies book by Y. Zhao)

Package “randomForest” is used below to build a predictive model for the iris data. There are two limitations with function randomForest(). First, it cannot handle data with missing values, and users have to impute data before feeding them into the function. Second, there is a limit of 32 to the maximum number of levels of each categorical attribute. Attributes with more than 32 levels have to be transformed first before using randomForest().
An alternative way to build a random forest is to use function “cforest()” from package party, which is not limited to the above maximum levels. However, generally speaking, categorical variables with more levels will make it require more memory and take longer time to build a random forest.

Splitting the data into training (70%) and testing (30%)

```
> #splitting the data into training and testing
> ind = sample(2,nrow(iris), replace= TRUE, prob = c(0.7,0.3))
> trainData = iris[ind==1,]
> testData = iris[ind==2,]
```

Installing and calling the package

```
> #loading library
> #installing the package needs to be done only once
> install.packages("randomForest")
> #calling the package using the library function needs to be done each and every time
> library(randomForest)
```

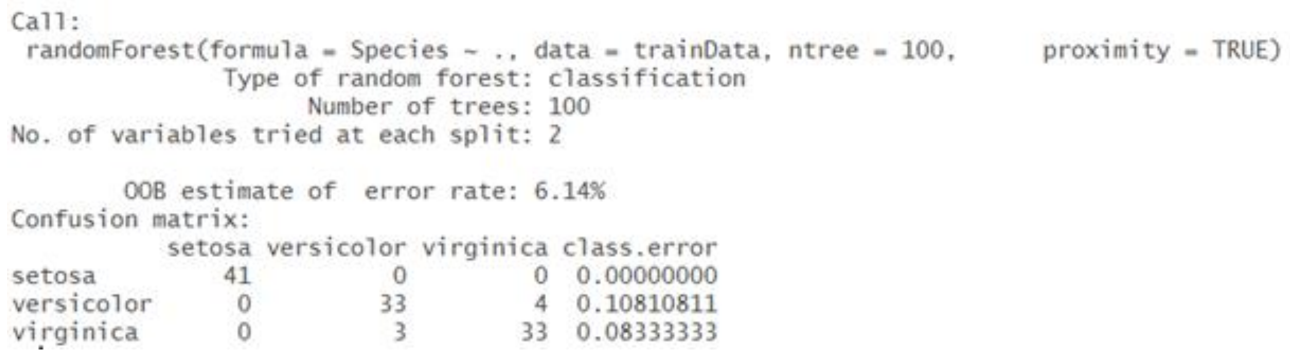
Creating the model on the training data set

```
> #creating the model
> rf = randomForest(Species ~ . , data = trainData,  ntree = 100,  mtry = 3, proximity = TRUE, replace = TRUE, sampsize = if (replace) nrow(trainData) else ceiling(0.65*nrow(trainData)), importance = TRUE )
> # ntree denotes the number of trees to grow. This should not be set to too small number. mtry = number of variables randomly sampled as candidates at each split. You can use the following formula to set mtry:
> # mtry=if (!is.null(Species) && !is.factor(Species)) max(floor(ncol(trainData)/3), 1) else floor(sqrt(ncol(trainData)))
> table(predict(rf), trainData$Species)
```

If mtry is the total number of features in the data set then this is the same as bagging.

After that, we can print the model, attributes, and also plot the error rates with various number of trees.

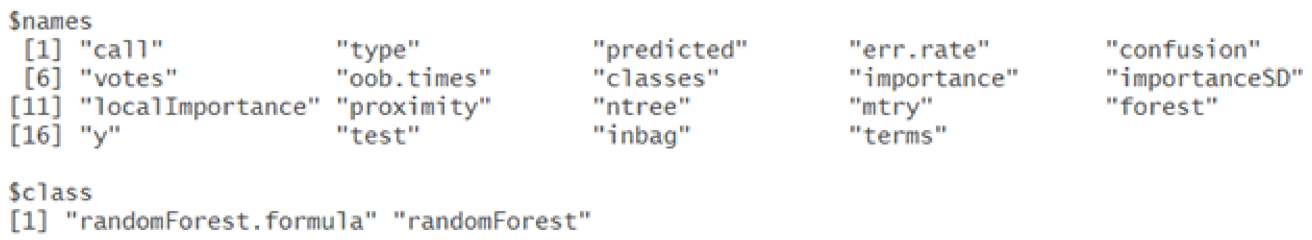
```
> #results- different functions to view results
> print(rf)
```



```
Call:
 randomForest(formula = Species ~ ., data = trainData, ntree = 100,      proximity = TRUE)
               Type of random forest: classification
                     Number of trees: 100
No. of variables tried at each split: 2

        OOB estimate of  error rate: 6.14%
Confusion matrix:
           setosa versicolor virginica class.error
setosa         41          0         0  0.00000000
versicolor      0         33         4  0.10810811
virginica       0          3        33  0.08333333
```

```
>attributes(rf)
```

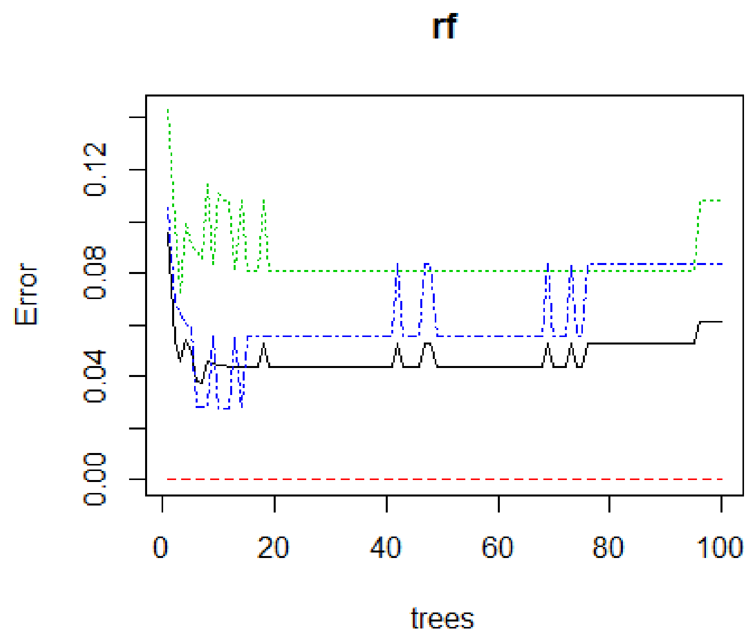


```
$names
 [1] "call"            "type"            "predicted"       "err.rate"        "confusion"
 [6] "votes"           "oob.times"       "classes"         "importance"      "importanceSD"
[11] "localImportance" "proximity"       "ntree"           "mtry"            "forest"
[16] "y"               "test"            "inbag"           "terms"

$class
[1] "randomForest.formula" "randomForest"
```

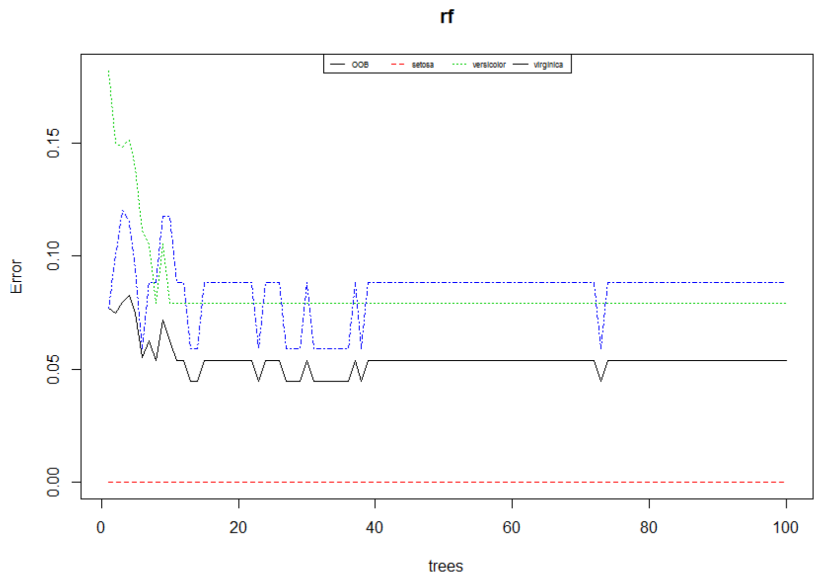
We plot the error rates with various number of trees.

```
>plot(rf)
```



The black line is the error rate on OOB, the red curve is the error rate for Setosa class, the green and blue curve are for Versicolor and Virginica classes. You can access the error rate for each class using rf$err.rate.
To add the legend to this graph you can use the following code:

```
> rndF1.legend <- if (is.null(rf$TestData$err.rate)) {colnames(rf$err.rate)} else {colnames(rf$TestData$err.rate)}
> legend("top", cex =0.5, legend=rndF1.legend, lty=c(1,2,3), col=c(1,2,3), horiz=T)
```

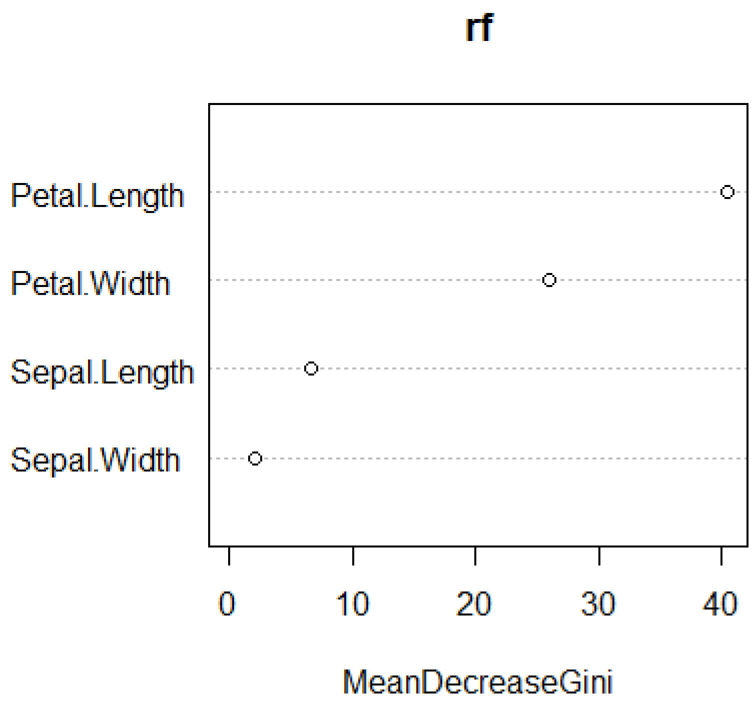


The importance of variables can be obtained with functions “importance()” and “varImpPlot()”.

```
>importance(rf)
># include type = 1 in the importance function to get the important variables based on MeanDecreaseAccuracy for regression trees
```

```
             MeanDecreaseGini
Sepal.Length         6.631367
Sepal.Width          2.115984
Petal.Length        40.441571
Petal.Width         26.045288
```

```
>#plot variable importance
>varImpPlot(rf)
```

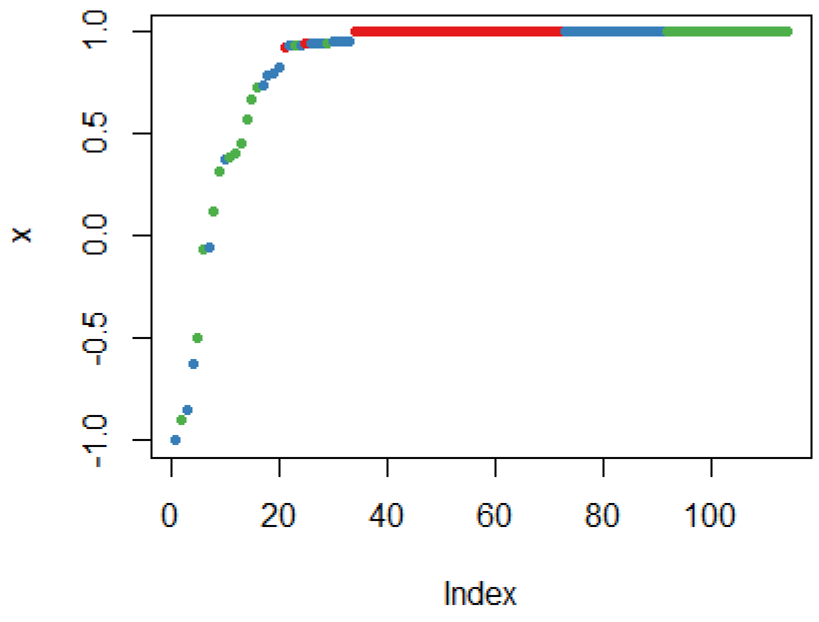


Finally, the built random forest is tested on test data, and the result is checked with functions “table()” and “margin()”. The margin of a data point is as the proportion of votes for the correct class minus maximum proportion of votes for other classes. Generally speaking, positive margin means correct classification.

```
> #get accuracy of prediction
> irisPred = predict(rf, newdata = testData)
> table(irisPred, testData$Species)
```

```
irisPred     setosa versicolor virginica
  setosa          9          0         0
  versicolor      0         13         0
  virginica       0          0        14
```

```
>plot(margin(rf, testData$Species))
> # The margin of a data point is defined as the proportion of votes for the correct class minus maximum proportion of votes for the other classes. Thus under majority votes, positive margin means correct classification.
```



You can pull out individual trees from the rf and look at their structure.

```
>getTree(rf, k =1, labelVar = TRUE)
> #Try this and check the results
```

To get the proximity matrix, you need to first make sure that the proximity is true if rf. Then you can get the matrix using

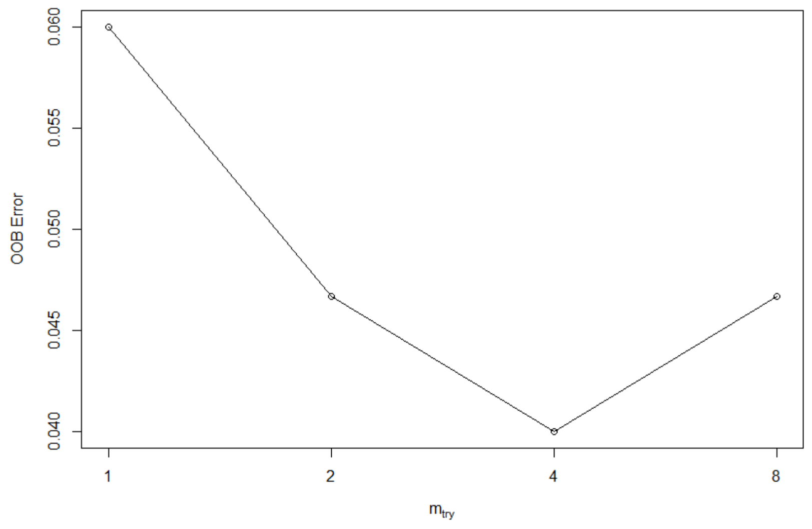
```
>rf$proximity
```

The proximity matric from randomForest is used to update the imputation of the NAs. You can use “rfImpute()” function to do so.

```
> ## artificially drop some data values.
for (i in 1:4) iris [sample(150, sample(20)), i] = NA
>iris.imputed = rfImpute(Species ~ ., data = iris,  ntree = 500)
>iris.rf = randomForest(Species ~., data = iris.imputed)
```

The function tuneRF() can be used to search for the optimal value (with respect to OOB error estimate) of mtry for random forest.

```
>tuneRF(iris[,-5], iris[,5], stepFactor = 0.5)
>#the first element is the matrix of predictors
># the second element is the target variable
># at each iteration, mtry is inflated or deflated by stepFactor value
```



To use bagged decision tree we can use the function “bagging()” in the package “adabag”.

```
> library(rpart)
> iris.bagging = bagging(Species ~ ., data = trainData, mfinal = 10)
> # mfinal is the number of trees to use. The default option is mfinal = 100 iterations
> # you can use the control argument used in rpart tree.
```

To get the in sample and out of sample error we can use the following code

```
>sum(iris.bagging$class != trainData$Species)/nrow(trainData)
>predict(iris.bagging, newdata = testData)$error
```

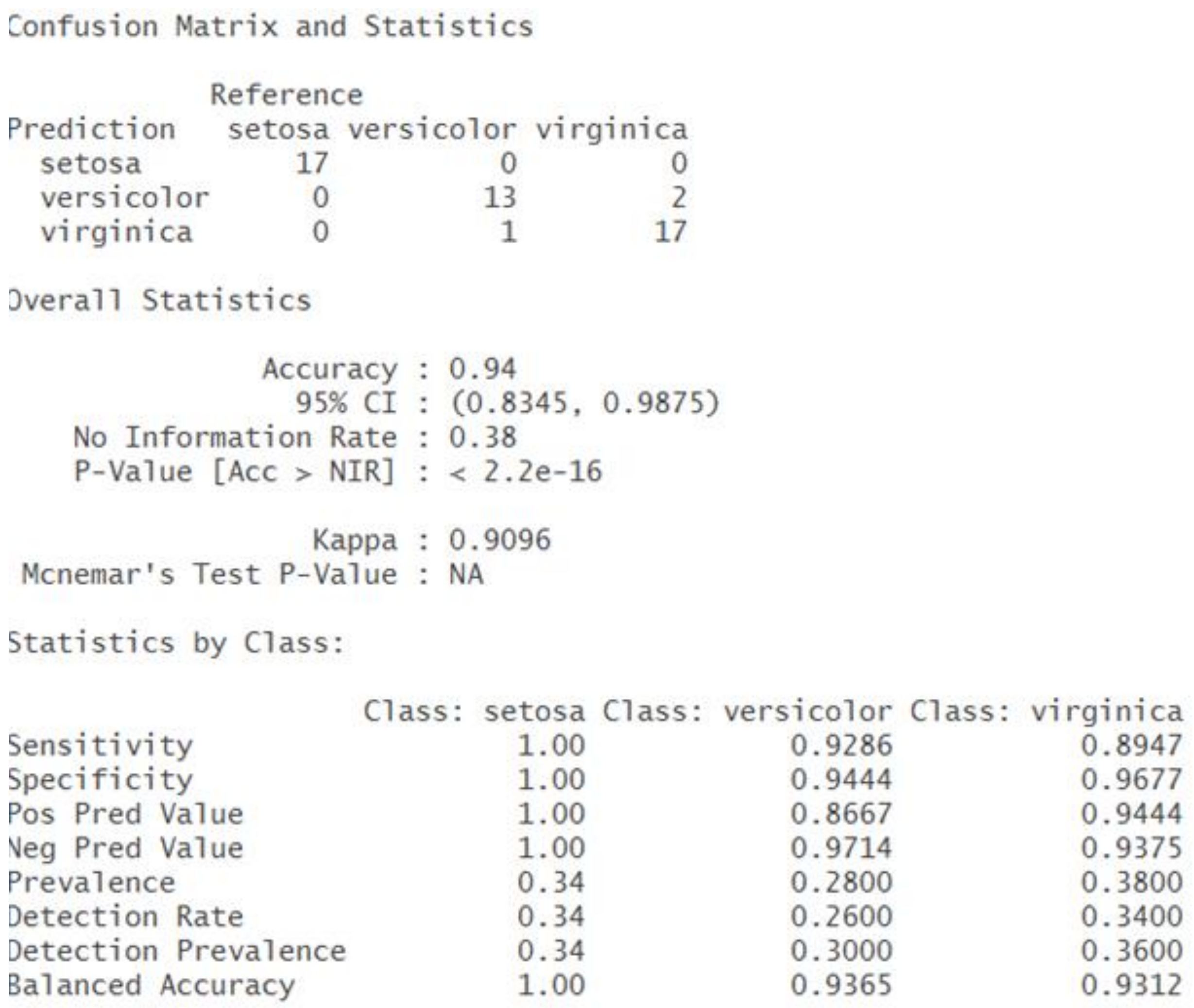
For boosting you can use the function “boosting()”.

```
> iris.adaboost = boosting(Species ~ ., data = trainData, boos = TRUE, mfinal = 10)
> # if boos = TRUE (default), a boostrap sample of the training data set is drawn using
the weights for each observation on that iteration. If FALSE, each observation is used
with its weights.
```

```
>#additional results

>#if not installed
>install.packages("caret")

>#calling the package library
>library(caret)
>caret::confusionMatrix(irisPred, testData$Species)
```



```
Confusion Matrix and Statistics

            Reference
Prediction   setosa versicolor virginica
  setosa         17          0         0
  versicolor      0         13         2
  virginica       0          1        17

Overall Statistics

               Accuracy : 0.94
                 95% CI : (0.8345, 0.9875)
    No Information Rate : 0.38
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9096
 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: setosa Class: versicolor Class: virginica
Sensitivity                   1.00            0.9286           0.8947
Specificity                   1.00            0.9444           0.9677
Pos Pred Value                1.00            0.8667           0.9444
Neg Pred Value                1.00            0.9714           0.9375
Prevalence                    0.34            0.2800           0.3800
Detection Rate                0.34            0.2600           0.3400
Detection Prevalence          0.34            0.3000           0.3600
Balanced Accuracy             1.00            0.9365           0.9312
```