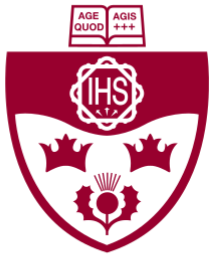


SAINT MARY'S UNIVERSITY SINCE 1802

One University. One World. Yours.

Master of Science in Computing and Data Analytics

Data and Text Mining
MCDA 5580

Assignment 2

Submitted by:

Siddhartha Lahkar (A00430620)

Sidharth Bhalla (A00431562)

Divya Chainani (A00432519)

Submitted to:

Dr. Pawan Lingras

Trishla Shah

SAINT MARY'S UNIVERSITY SINCE 1802
One University. One World. Yours.

Table of Contents

SAINT MARY'S UNIVERSITY SINCE 1802
One University. One World. Yours.

Executive Summary

This report gives the detailed explanation of the various classification algorithm used and the parameters tuned in order to predict the acceptability of the car. The dataset considered in the process was Car Data which contains information about the features of a car such as Price, Maintenance, Doors, Seats, Safety, ShouldBuy. The dataset was divided into the Train and Test datasets. The model was created on the Train dataset while tuning parameters. This model was further tested with the Test dataset.

The classification algorithms used in the process were decision tree and random forest. The process was initiated with decision tree. The classification model was constructed with the training dataset while utilizing the decision trees. The model was achieved with 94% accuracy and was able to best predict the "good" class of ShouldBuy variable. This model was then tested on the test dataset.

The similar approach was followed while creating model with Random Forest. The Random Forest model was able to predict 96% of the values in training dataset correctly and was working the best for classifying "acc" class of ShouldBuy. This model was then tested on the test dataset.

Random Forest Algorithm was further utilized by the K-fold Cross Validation for creating an accurate model which helped in finding out the most significant variable(s) that were useful for predicting the "Car Acceptability". It was observed that 4 parameters namely Safety, Seats, Maintenance and Price were important characteristics for car acceptance. Also, out of the four parameters Safety was the most and Price was the least important features.

Objective

The scope of the assignment is to utilize various classification algorithms and figure out the one that works the best in calculating the significant features which could be used further for predicting the car acceptability. The dataset used for prediction is the famous car dataset. The algorithms applied on the dataset would help in predicting whether a customer would buy/accept a car, provided various conditions or features associated with the vehicle.

Data Summary

The data considered is the car data that contains 7 variables out of which 6 are Independent (or Explanatory/Predictor) variables and one is Dependent variable.

Independent/Explanatory Variables: -

- Price (high, low, medium, vhigh)
- Maintenance (high, low, medium, vhigh)
- Doors (2, 3, 4, 5more)
- Seats (2, 4, more)
- Storage (big, med, small)
- Safety (high, low, med)

Dependent/Response Variable: -

- ShouldBuy (acc, good, unacc, vgood)

The Independent variables refer to the attributes of the car and the dependent variable gives the "Car Acceptability" result. All variables are categorical in nature and have 3-4 levels in each. There are total of 1728 records in the dataset. For performing various tests, the car data was divided into two categories – Training dataset and Test Dataset where the ratio of Training and Test datasets is 70:30 of the overall dataset.

Design, Method and Approach

The approach taken to augment the classification of the dataset required the following tasks to be performed: -

1. Load data into RStudio
2. Extract the required libraries
3. Create Decision Tree Model using the train dataset for car
4. Test the newly constructed Decision Tree Model using the test dataset
5. Create Random Forest Model using the train dataset
6. Test the Random Forest Model using the test dataset
7. Create Random Forest Model on dataset using K-fold Cross Validation
8. Inferring the most significant variables for predicting the model

Classification Algorithm (Decision Tree)

A classification algorithm was created based on the model of decision tree. Some popular libraries used to create and visualize the model are – *rpart* and *rpart.plot.* Below are the steps which explains the process of creating a decision tree:

1. Creating Train and Test datasets from the given dataset.
2. Creating decision tree model on the training set with default parameters
3. Fine Tuning the model by changing the tuning parameters.
4. Testing the tuned model on the test dataset.

1. Creation of Train and Test datasets

The sample() function was used to split the car dataset into Train and Test datasets. 70% of data was used for train data and 30% for the test dataset.

```
set.seed(100)
train <- sample(nrow(carData), 0.7*nrow(carData), replace = FALSE)
TrainSet <- carData[train,]
TestSet <- carData[-train,]
summary(TrainSet)
summary(TestSet)
```

Figure 1: Code for splitting the datasets

Below figures show the summary of the datasets created:

```
> summary(TrainSet)
   price      maintenance    doors       seats       storage      safety
 high :313   high :287   2    :305   2    :406   big  :416   high:396
 low  :292   low  :317   3    :300   4    :399   med  :383   low :412
 med  :305   med  :303   4    :295   more:404   small:410   med :401
 vhigh:299   vhigh:302   5more:309
 shouldBuy
 acc  :264
 good : 52
 unacc:856
 vgood: 37
```

Figure 2: Train dataset summary

```
> summary(TestSet)
   price      maintenance    doors       seats       storage      safety     shouldBuy
 high :119   high :145   2    :127   2    :170   big  :160   high:180   acc  :120
 low  :140   low  :115   3    :132   4    :177   med  :193   low :164   good : 17
 med  :127   med  :129   4    :137   more:172   small:166   med :175   unacc:354
 vhigh:133   vhigh:130   5more:123                                      vgood: 28
```

Figure 3: Testset (Test dataset) summary

2. Training the model

Using the train dataset, the decision tree was created using the rpart(). Function which utilizes the default Gini index criteria to split the records. The minsplit value in the function was set to 30 initially. The minsplit being the minimum number of observations that must exist in a node to further split it into the next level of the decision tree.

```
#Creating the tree
treecar = rpart(shouldBuy~., data = TrainSet, method = "class",  control = rpart.control(minsplit = 30))
treecar
```

Figure 4: creating the decision tree using rpart()

Post execution of rpart(), the tree was plotted using rpart.plot()

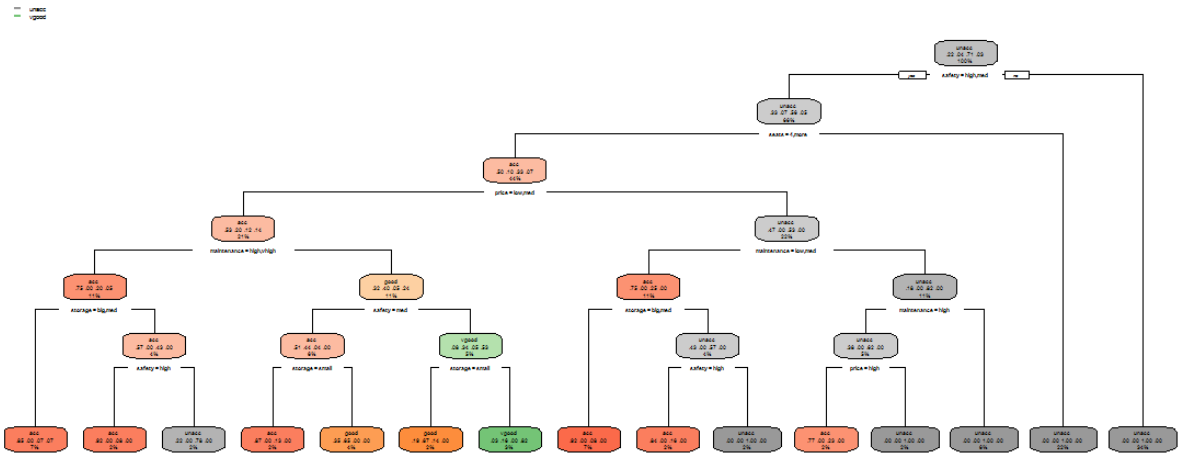


Figure 5: Decision Tree with minsplits = 30

```
n= 1209

node), split, n, loss, yval, (yprob)
      * denotes terminal node

 1) root 1209 353 unacc (0.21836228 0.04301075 0.70802316 0.03060380)
   2) safety=high,med 797 353 unacc (0.33124216 0.06524467 0.55708908 0.04642409)
     4) seats=4,more 526 262 acc (0.50190114 0.09885932 0.32889734 0.07034221)
       8) price=low,med 258 121 acc (0.53100775 0.20155039 0.12403101 0.14341085)
        16) maintenance=high,vhigh 127  32 acc (0.74803150 0.00000000 0.20472441 0.04724409)
          32) storage=big,med 80  12 acc (0.85000000 0.00000000 0.07500000 0.07500000) *
          33) storage=small 47  20 acc (0.57446809 0.00000000 0.42553191 0.00000000)
            66) safety=high 24   2 acc (0.91666667 0.00000000 0.08333333 0.00000000) *
            67) safety=med 23   5 unacc (0.21739130 0.00000000 0.78260870 0.00000000) *
        17) maintenance=low,med 131  79 good (0.32061069 0.39694656 0.04580153 0.23664122)
          34) safety=med 72  35 acc (0.51388889 0.44444444 0.04166667 0.00000000)
            68) storage=small 23   3 acc (0.86956522 0.00000000 0.13043478 0.00000000) *
            69) storage=big,med 49  17 good (0.34693878 0.65306122 0.00000000 0.00000000) *
          35) safety=high 59  28 vgood (0.08474576 0.33898305 0.05084746 0.52542373)
            70) storage=small 21   7 good (0.19047619 0.66666667 0.14285714 0.00000000) *
            71) storage=big,med 38   7 vgood (0.02631579 0.15789474 0.00000000 0.81578947) *
       9) price=high,vhigh 268 127 unacc (0.47388060 0.00000000 0.52611940 0.00000000)
        18) maintenance=low,med 139  35 acc (0.74820144 0.00000000 0.25179856 0.00000000)
          36) storage=big,med 90   7 acc (0.92222222 0.00000000 0.07777778 0.00000000) *
          37) storage=small 49  21 unacc (0.42857143 0.00000000 0.57142857 0.00000000)
            74) safety=high 25   4 acc (0.84000000 0.00000000 0.16000000 0.00000000) *
            75) safety=med 24   0 unacc (0.00000000 0.00000000 1.00000000 0.00000000) *
        19) maintenance=high,vhigh 129  23 unacc (0.17829457 0.00000000 0.82170543 0.00000000)
          38) maintenance=high 60  23 unacc (0.38333333 0.00000000 0.61666667 0.00000000)
            76) price=high 30   7 acc (0.76666667 0.00000000 0.23333333 0.00000000) *
            77) price=vhigh 30   0 unacc (0.00000000 0.00000000 1.00000000 0.00000000) *
          39) maintenance=vhigh 69   0 unacc (0.00000000 0.00000000 1.00000000 0.00000000) *
     5) seats=2 271   0 unacc (0.00000000 0.00000000 1.00000000 0.00000000) *
   3) safety=low 412   0 unacc (0.00000000 0.00000000 1.00000000 0.00000000) *
```

Figure 6: Decision Tree rules

Post creation of the decision tree as shown above, the confusion matrix was created to find the number of true/false positive records and true/false negative records which were identified by the model.

```
> treeCar
       predcar
        acc good unacc vgood
  acc   237   21     5     1
  good    0   46     0     6
  unacc  29    3   824     0
  vgood   6    0     0    31
```

Figure 7: Confusion matrix

From the above table it is inferred that, the total true positives for 'acc' are 237 where as 27 (21 + 5+ 1) records of class 'acc' are identified as 'good', 'unacc' and 'vgood'. Similarly we can read the said values for other classes as well.

```
> sum(diag(treeCar))/sum(treeCar)
[1] 0.9412738
```

Figure 8: Accuracy of the model

The sum of diagonals by sum of all records in the confusion matrix table gives the accuracy of the decision tree. Higher the accuracy value, better is the decision tree at classifying records. In this scenario the value is 0.9412738 which is closer to the value of 1, and hence it can be said that the model has a high accuracy and able to classify, if not all records correctly.

3. Fine Tuning using different minsplit values: -

Using minsplit = 20:

```
> treeCar1 =table(TrainSet[,7],predcar1)
> treeCar1
       predcar1
        acc good unacc vgood
  acc   247   11     5     1
  good    0   46     0     6
  unacc  29    3   824     0
  vgood   6    0     0    31
> sum(diag(treeCar1))/sum(treeCar1)
[1] 0.9495451
```

Figure 9: Confidence Matrix and accuracy with minsplit = 20

Using minsplit = 10:

```
> treeCar2 =table(TrainSet[,7],predcar2)
> treeCar2
       predcar2
        acc good unacc vgood
  acc   247   11     5     1
  good    0   46     0     6
  unacc  29    3   824     0
  vgood   6    0     0    31
> sum(diag(treeCar2))/sum(treeCar2)
[1] 0.9495451
```

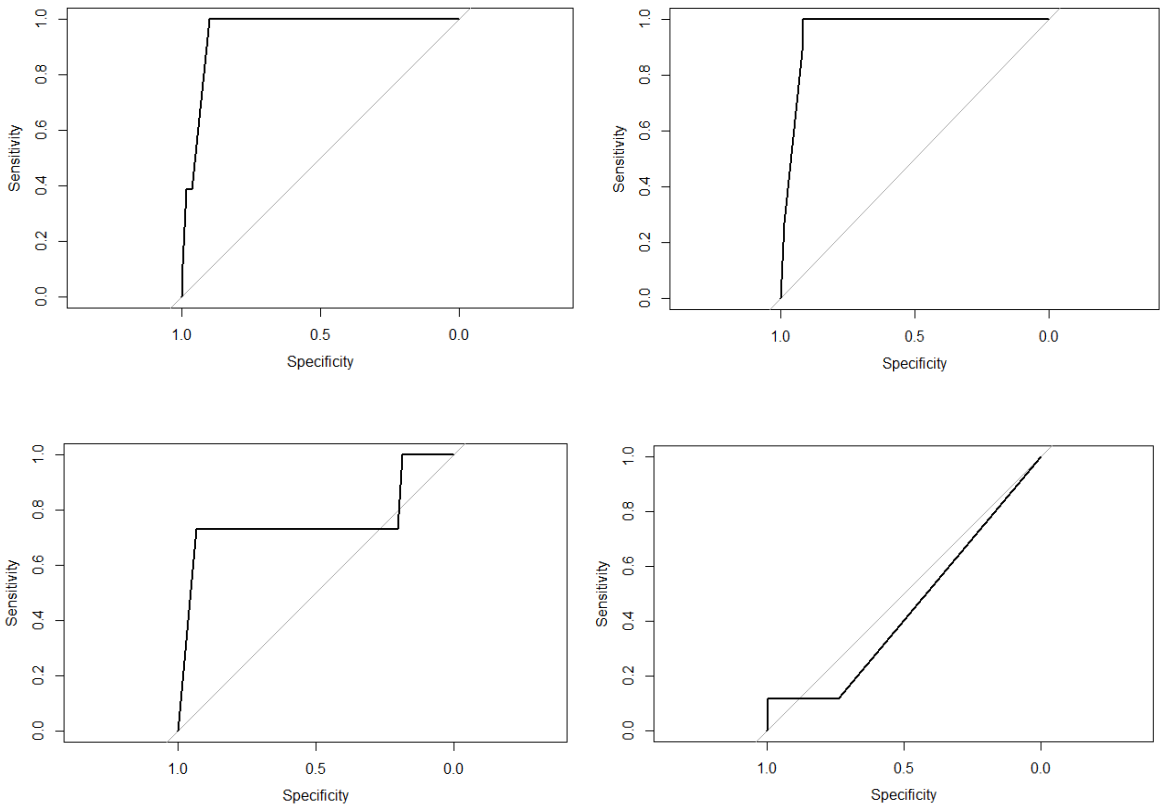
Figure 10: Confusion matrix and accuracy with minsplit =10

The recursive operation on the classification model does not show any significant changes on the accuracy level of the model and thus our minsplit remains unchanged during the remaining course of analysis.

ROC curves and Significant Class:

Next, the ROC (Receiver Operating Characteristic Curve) was checked which helps in checking the probability of the outcomes in a classification model.

Class	Area under the curve
Acc	0.9536
Good	0.9594
Unacc	0.7579
Vgood	0.4419



Sensitivity defines how many correct values the model recognizes positively
Specificity defines the negative values which the model identifies correctly

When we compared the value of AUC for all the classes we inferred that the class "good" of the *shouldBuy* column has the highest area under the curve value and is the most significant class which the decision tree can predict with highest accuracy.

4.Testing

The model was further tested and this time the test dataset was fed into the model. Below are the inferences which are drawn from this:

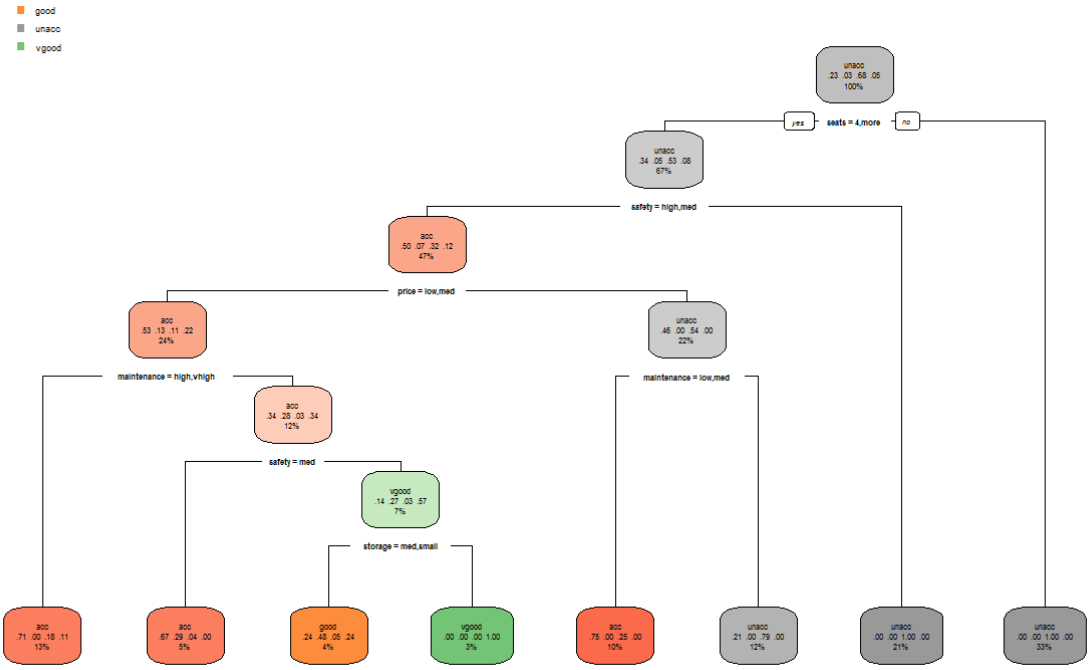


Figure 11: Decision tree using test dataset and minsplit = 30

```
> treeCarCM=table(TestSet[,7],predCar)
> treeCarCM
       predCar
        acc good unacc vgood
  acc   102    5    13     0
  good    7   10     0     0
  unacc  26    1   327     0
  vgood   7    5     0    16
> sum(diag(treeCarCM))/sum(treeCarCM)
[1] 0.8766859
```

Figure 12: Confusion matrix and accuracy value

Class	Area under the curve
Acc	0.8348
Good	0.9431

Unacc	0.8819
Vgood	0.6944

The test dataset gives us a low accuracy value of 0.87 compared to that of 0.94 of the train dataset. However, we observe that "good" remains as the most significant class which is predicted by the model with highest AUC value.

Decision trees are very simple and easy to understand models; however, they have very low predictive power. In fact, they are called weak learners.

Random Forest

Following are the steps followed for utilizing Random Forest on the dataset:-

1. Creating Train and Test datasets from the given dataset.
2. Creating Random Forest Model from Train dataset keeping parameters values default with the help of Random Forest Package.
3. Fine Tuning the model by changing the tuning parameters.
4. Testing the tuned model on the test dataset.

1.Creating Train and Test datasets from the given dataset

Random Forest was applied on the same training and test datasets which were used for decision tree creation (mentioned above).

2.Creating Random Forest with default parameters

The Random Forest model was first created on the train dataset with the parameters having their default values. The model was created with the help of randomForest() function of the "randomForest" library which implements Breiman's random forest algorithm for classification and regression. The model was able to predict the values with 99% accuracy and worked best for predicting "ShouldBuy" = "acc" and "good". Below figures show the result of the confusion matrix.

```
        yc
rfp      acc good unacc vgood
  acc    263    0     1     1
  good     1   52     0     0
  unacc    0    0   855     0
  vgood    0    0     0    36
> sum(diag(rfCM))/sum(rfCM)
[1] 0.9975186
```

Figure 12: Confusion matrix and accuracy value of Train dataset with default parameters

3.Fine Tuning of Random Forest

Building the Classifier:-

The parameters used for tuning the random forest model are :-

1. Ntree:- Ntree specifies the number of trees to grow. In the default S3 method ntree is set as 500.
2. Mtry:- Mtry specifies the number of independent variables randomly sampled at each split. In the model created with default parameters, mtry was taken as 2.
3. Nodesize:- Nodesize specifies the number of records to be kept in the terminal nodes .

The Random Forest model was tuned by changing the value of the parameter values as :-

1. Ntree:- The model was checked for various values of ntree such as 200,300,400,500,600 out of which the best results were observed with ntree=**500**.
2. Mtry:- Similar to ntree, the model also checked for multiple mtry values like 2,3,4,5,6. The model gave accurate results for mtry = **4**.
3. Nodesize:- The model was tuned with various nodesize values such as 10 & 50 . The model gave accurate results for nodesize =**10**.

Below figure shows the random forest generated after tuning the parameters (ntree = 500, mtry = 4, nodesize = 10).

```
Call:
 randomForest(x = xc, y = yc, ntree = 500, mtry = 4, nodesize = 10)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 4

        OOB estimate of  error rate: 4.71%
Confusion matrix:
      acc good unacc vgood class.error
acc   244   11     8     1  0.07575758
good    0   46     0     6  0.11538462
unacc  23    2   831     0  0.02920561
vgood   6    0     0    31  0.16216216
```

Figure 13: Model created after tuning parameters

Making Predictions:-

After tuning the default model with the various parameter values, the model was achieved with **96%** predicting accuracy. Below figure shows the confusion matrix generated from the tuned model and it's accuracy.

```
        yc
rfp      acc good unacc vgood
  acc    250    0    15     6
  good    12   50     3     0
  unacc    2    0   838     0
  vgood    0    2     0    31
> sum(diag(rfCM))/sum(rfCM)
[1] 0.9669148
```

Figure 14: Confusion matrix and accuracy of the tuned model

In order to visualize and quantify the accuracy of the model, ROC was used to get the area under the curve for "acc","good","unacc" and "vgood" respectively.

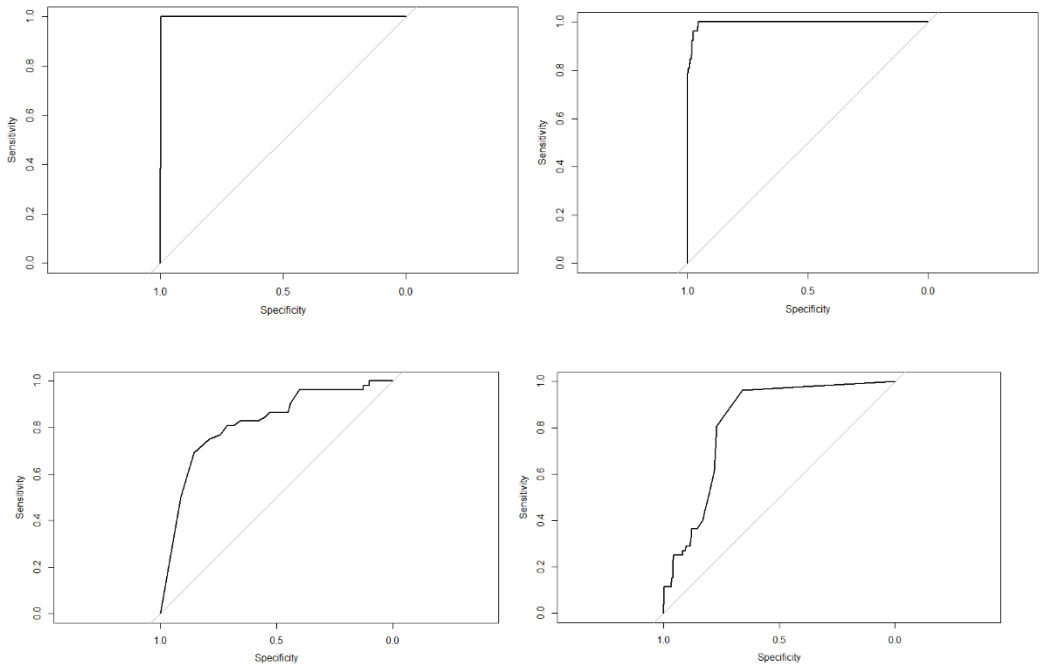


Figure 15: ROC curves of the dependent variable classes

From the above figures , it is observed that the AUC =1 (almost) for response variable="acc". Thus, it can be said that the model was very close to the optimal point of predicting the class as "acc" in the dataset.

4.Testing the tuned model

The model created was now tested with the test dataset. This test is performed in order to check the efficiency of the created model. For testing, same set of rules (applied while training the model) were applied on the test dataset. It was observed that the model was able to predict the values with 97% accuracy.

```
        yc
rfp      acc good unacc vgood
  acc    118    3     4     1
  good     0   11     1     0
  unacc    2    0   349     0
  vgood    0    3     0    27
> sum(diag(rfCM))/sum(rfCM)
[1] 0.973025
```

Figure 16: Confusion matrix and accuracy value of Test dataset

K-Fold Cross Validation using Caret package

The K-fold cross validation resampling procedure was used to calculate the K bins (K=10) of dataset iteratively and perform classification using random forest algorithm. The random forest classification technique was specified as the parameter in the function used to train the model for the car dataset. Since K=10 was passed as the reference value, therefore, the algorithm becomes 10-fold Cross Validation and hence the steps followed during the training.

The procedure of K-fold cross validation:

1. Shuffling the car dataset randomly.

2. Dividing the dataset into K-groups.

3. Following the below steps for each group-
   - Randomly selecting one group as the test dataset from the overall set.
   - Considering the rest of the set as the training data set for the classification.
   - Fitting the model on the training data set and then evaluating results against the results of the test data set.
   - Persisting the evaluated scores from the results and then discarding the model.

4. Summarizing the model classification capability using the average of persisted evaluation scores.

1. Creating Random Forest using K-fold cross validation: -

The random forest was created using the K-fold cross validation method for classifying the car dataset into the 4 classes of 'ShouldBuy'. The Train function was used to train the model which requires various parameters such as independent data frame (predictor variables), dependent data vector (classification classes), method, metric, trControl, tuneGrid, tuneLength. The Train function is described briefly below with all the explored parameters used for training and fine tuning the model.

Train function

The train function requires fine tuning parameters, fits the model on the specified classification or regression routine and then calculates the performance measure such as 'Accuracy' from the resampling of K-fold method.

Here, the train function parameters utilized in the model fitting are specified:-

a. **x =** carData[,1:6]
It is the independent data set or predictor variables.

b. **y =** carData[,7]
The 'ShouldBuy' column with the classification classes – unacc, acc, good, vgood.

c. **metric =** 'Accuracy'
The metric of the model is calculated on the basis of three features – mtry (number of predictor variables out of 6 used to train the data), Accuracy and Kappa.

d. **trControl =** control
control = trainControl(method="repeatedcv", number=10, repeats=3)
The *trainControl* function with the required K-fold sampling arguments was passed as the reference value to the *trControl* parameter of train function. The *trainControl* function manages the computational nuances of the train function and it captures the below arguments values-

- **method** = 'repeatedcv'
  repeatedcv = the repeated cross validation sampling was performed for fitting the model.
- **number** = 10

The number 10 stands for the K-fold, thus, 10-fold cross validation

- **repeats** = 3
  It is the number of iterations defined, for which, the entire 10-fold resampling was performed.

Model fitting with Train Function

The train function was executed for training the model with all the above-specified function arguments. The rf_default was the random forest model fitted on the car data with 98.53 % accuracy for 'mtry' = 6 i.e. the model considered all the 6 predictor variables for fitting the most accurate model. Below code and the figure depicts the random forest results obtained without fine-tuning the model.

control <- trainControl(method="repeatedcv", number=10, repeats=3)

rf_default <- train(x,y, method="rf", metric=metric, trControl=control)

```
> rf_default <- train(x,y, method="rf", metric=metric, trControl=control)
> print(rf_default)
Random Forest

1728 samples
   6 predictor
   4 classes: 'acc', 'good', 'unacc', 'vgood'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 1554, 1555, 1556, 1556, 1555, 1555, ...
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
  2     0.9737618  0.9433304
  4     0.9828290  0.9630267
  6     0.9853350  0.9681679

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 6.
```

Figure 17: Random Forest creation with default parameters

2. Fine Tuning of Random Forest

The model was fine-tuned for various arguments in the *trainControl* and *train* function. Following are the parameters utilized for fine-tuning the model :-

a. **search =** "random"
   The default grid search becomes ineffective when various optimizing parameters are utilized. Therefore, the random search (of TrainControl function) was used to randomly select the combination of fine tuning parameters, here, mtry was selected randomly for each sampling.

b. tuneLength = 10

The tuneLength argument of train function was used in sync with search argument of trainControl function which dictates the algorithm to select different number of values for the main parameter. Here, restricting tuneLenght to value 2 resulted in consideration of mtry = 2 and mtry = 6 only for fitting the model.

c. **tuneGrid** = expand.grid(.mtry=cmtry) where cmtry <- c(2,4)
The tuneGrid argument of train function explicitly allowed to specify the mtry valus for fitting the model.

d. **tuneRF**(x,y,stepFactor = 1.5, ntreeTry = 400, doBest = TRUE)
The tuneRF function of random forest library was used to check the graph of mtry vs. OOB (Out-Of-Bag) error rate i.e. the prediction error rate for different mtry values.

Random Search and TuneLength Parameters

Upon using the combination of search and tunelength parameters, it was observed that the model now considered mtry values{1,2,3,5,6} which were selected in a random fashion in each sampling iteration. The below figure for *rf_search* (random forest) was obtained on executing the following code. The model was 98.5 % accurate and fitted with mtry =6.

control *<- trainControl(method="repeatedcv", number=10, repeats=3, verboseIter = TRUE,* ***search*** *= "random")*
rf_search *<- train(x,y, method="rf", metric=metric,* ***tuneLength*** *= 10, trControl=control)*

```
+ Fold10.Rep3: mtry=6
- Fold10.Rep3: mtry=6
Aggregating results
Selecting tuning parameters
Fitting mtry = 6 on full training set
> print(rf_search)
Random Forest

1728 samples
   6 predictor
   4 classes: 'acc', 'good', 'unacc', 'vgood'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 1554, 1555, 1556, 1556, 1555, 1555, ...
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
  1     0.8152113  0.5018649
  2     0.9727928  0.9412334
  3     0.9781991  0.9532246
  5     0.9847547  0.9670349
  6     0.9853339  0.9682029

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 6.
```

Figure 18: Random Forest creation with default parameters

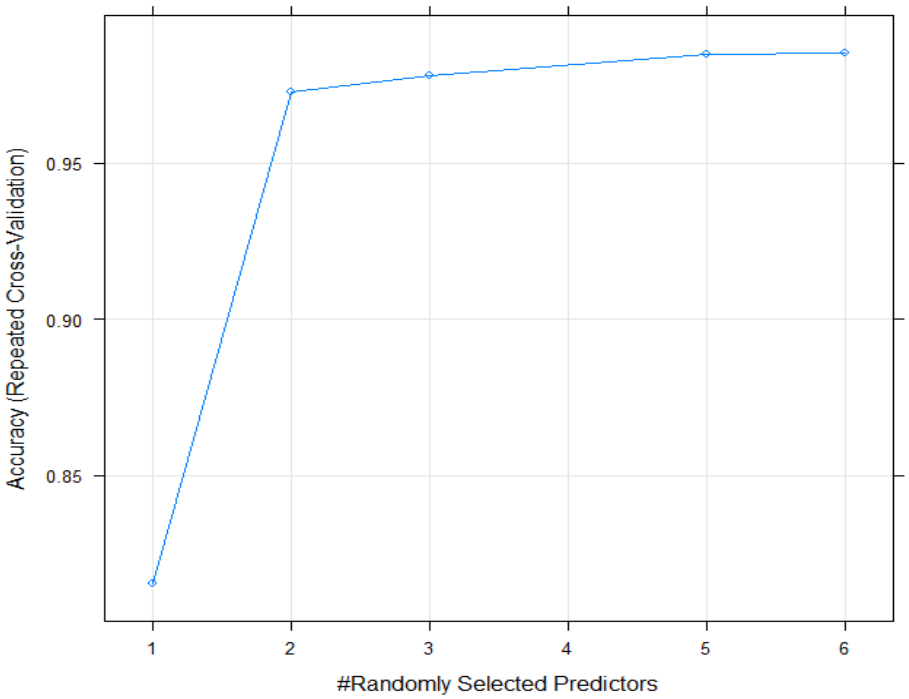


Figure 19: Graph of Accuracy with mtry

TuneGrid Parameter

The above combination does not allow feasibility to change the number of variable selection while creating the random forest. This is facilitated by tuneGrid argument. Upon using the tuneGrid parameter, it was observed that the model now considered all the possible mtry values from 1 to 6. The model was trained accordingly and the accuracy as well as Kappa was calculated for all the mtry values. The below figure for ***rf_tuneGrid*** (random forest) is obtained on executing the following code. Again, the model was 98.5 % accurate and fitted with mtry =6 value.

control *<- trainControl(method="repeatedcv", number=10, repeats=3, verboseIter = TRUE)*

cmtry <- c(1,2,3,4,5,6)

tunegrid <- expand.grid(.mtry=cmtry)

rf_tuneGrid *<- train(x,y, method="rf", metric=metric,* ***tuneGrid*** *= tunegrid, trControl=control)*

```
> print(rf_tuneGrid)
Random Forest

1728 samples
   6 predictor
   4 classes: 'acc', 'good', 'unacc', 'vgood'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 1554, 1555, 1556, 1556, 1555, 1555, ...
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
  1     0.8154230  0.5030646
  2     0.9718373  0.9391051
  3     0.9782013  0.9532786
  4     0.9828279  0.9629765
  5     0.9847581  0.9670669
  6     0.9855288  0.9686138

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 6.
```

Figure 20: Tuned Random Forest Model

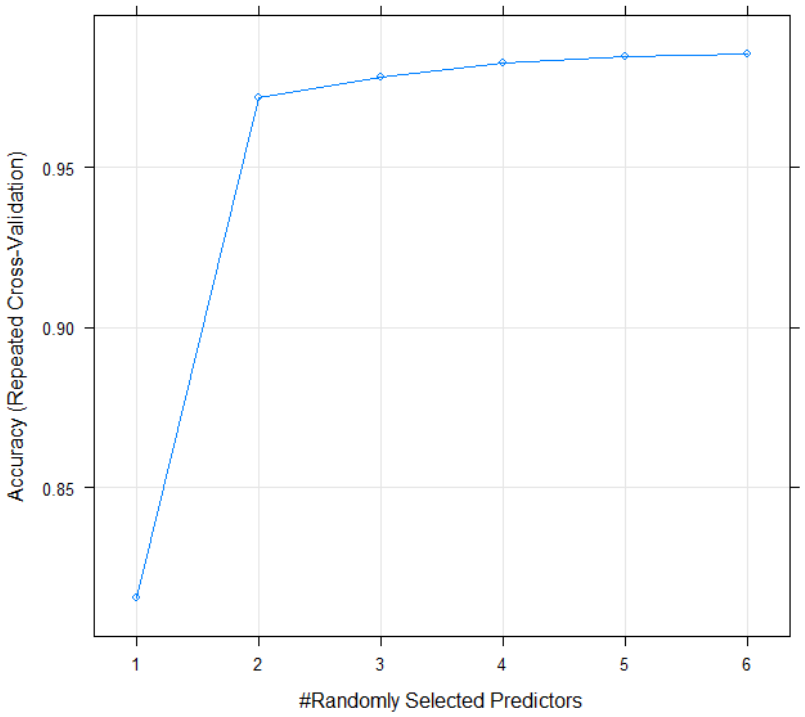


Figure 21: Graph of Accuracy with mtry of the tuned model

TuneRF function for comparison of mtry with OOB error rate

The tuneRF function was used to check the error rate between different mtry values(discussed above). The below figure and graph depicts the OOB error rate for the mtry values. The error rate was least for mtry =6 as it takes into consideration all the variables

while creating the model. Hence, the next mtry values with less error rate were considered and accuracy check was done to finalize the mtry which gives the best predicting model.

```
> trf<-tuneRF(x,y,stepFactor = 1.5, ntreeTry = 400, doBest = TRUE)
mtry = 2  OOB error = 2.55%
Searching left ...
Searching right ...
mtry = 3        OOB error = 1.97%
0.2272727 0.05
mtry = 4        OOB error = 1.74%
0.1176471 0.05
mtry = 6        OOB error = 1.56%
0.1 0.05
```

Figure 22: OOB error rate of the mtry values

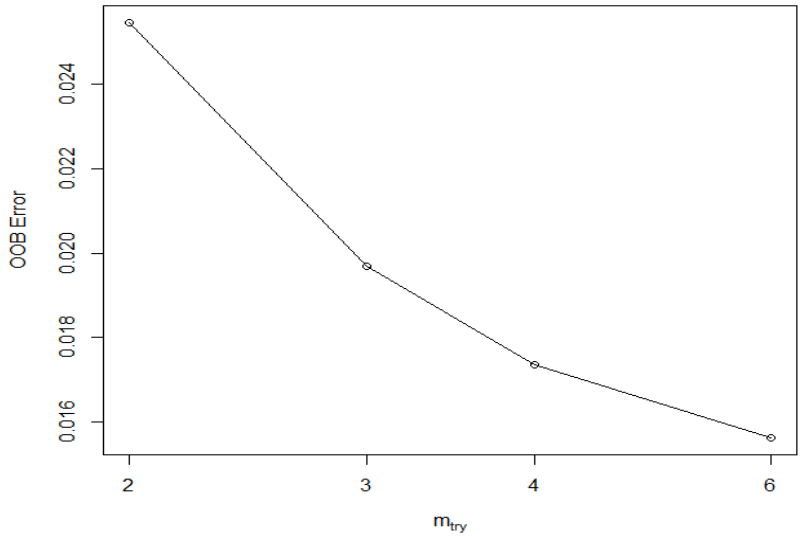


Figure 23: Graphical representation of OOB error rate of the mtry values

Final Model Fitting with TuneGrid: -

The final model was fitted with tuneGrid argument for mtry values 2 and 4. The below figure of ***rf_2_vs_4*** (random forest) is obtained on executing the following code.

control *<- trainControl(method="repeatedcv", number=10, repeats=3, verboseIter = TRUE)*

*cmtry <- c(**2,4**)*

tunegrid <- expand.grid(.mtry=cmtry)

rf_2_vs_4 *<- train(x,y, method="rf", metric=metric,* ***tuneGrid*** *= tunegrid, trControl=control)*

```
> print(rf_2_vs_4)
Random Forest

1728 samples
   6 predictor
   4 classes: 'acc', 'good', 'unacc', 'vgood'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 1554, 1555, 1556, 1556, 1555, 1555, ...
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
  2     0.9729911  0.9417012
  4     0.9837947  0.9650731

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 4.
```

Figure 24: Tuned Random Forest Model

The model was **98.37** % accurate and fitted with mtry value **4** against 2.

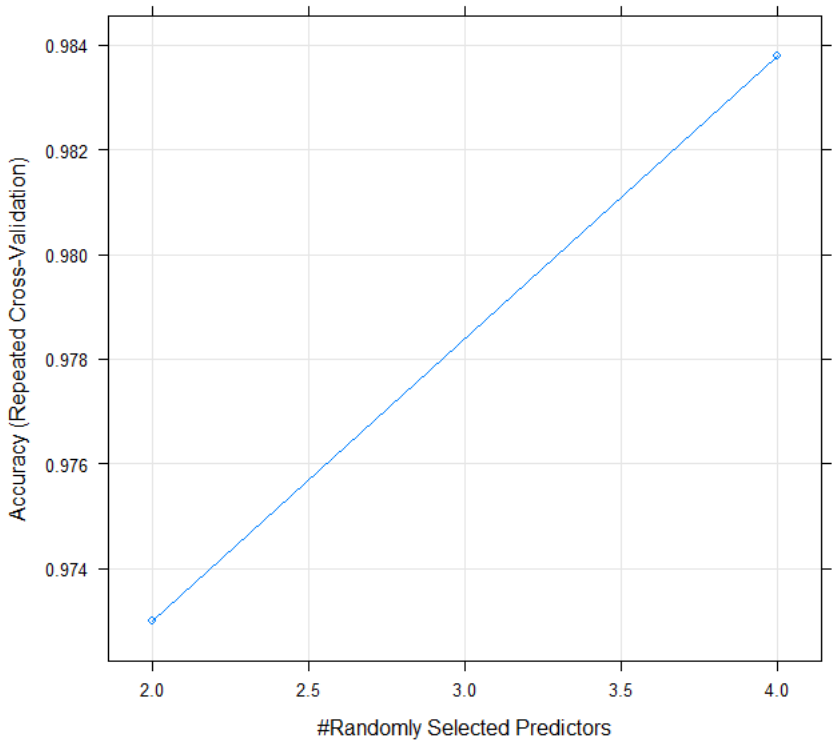


Figure 25: Graphical representation of accuracy with mtry

3. Best predictor variables

Once the model is created, the same was used to find out the important variables/features that can be best used to predict the acceptability of car. In order to achieve this varImp() function was used on the random forest model created above. The importance of variables was first extracted on the scale of 0-100 where variable with value 0 is least important and 100 being

the most important. The same is plotted with the help of GGplot. Below is the code used for calculating the variable importance and plotting the same along with the figures.

varImp(rf_default)

```
> varImp(rf_2_vs_4)
rf variable importance

            Overall
safety       100.00
seats         75.17
maintenance   49.73
price         33.95
storage       25.61
doors          0.00
```

Figure 26: Calculating the variable importance

ggplot(varImp(rf_default))

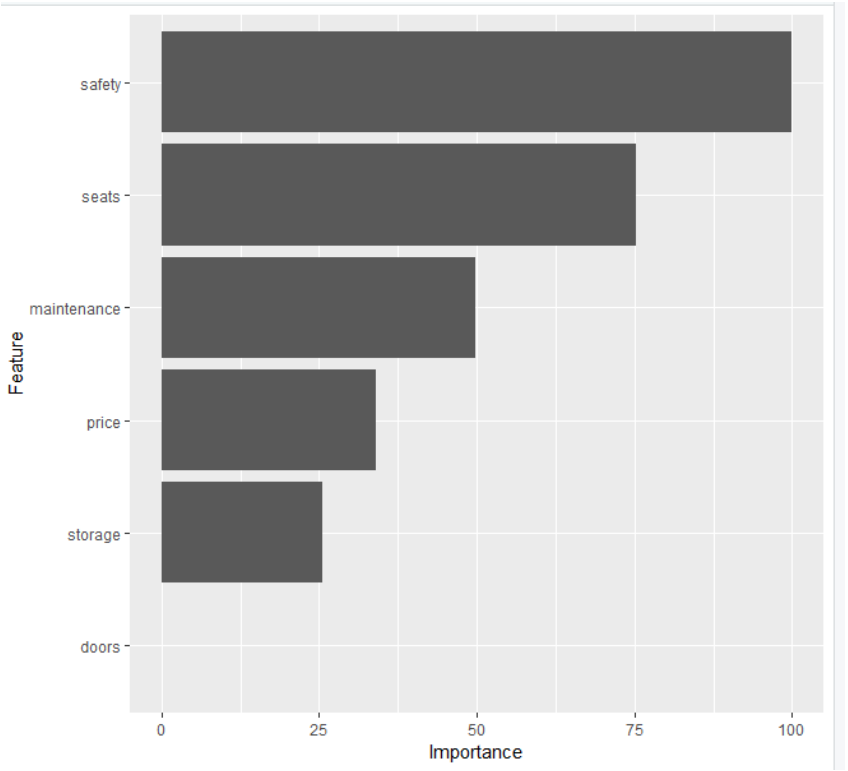


Figure 27: Graphical representation of features with their importance

Conclusion

Based on the above analysis from K-fold cross validation using random forest algorithm, it was concluded that there are 4 parameters on the basis of mtry namely Safelty, Seats, Maintenance and Price which hold importance in predicting the "Car Acceptability". For any car to be accepted, it should check the parameters in the order :- Safety->Seats->Maintenance->Price where Safety being the most important and Price the least important among the four.

References

[1] Title-How to implement Random Forests in R. Retrieved from *URL*:

https://www.r-bloggers.com/how-to-implement-random-forests-in-r/

[2] Title-A gentle introduction to k-fold Cross-Validation. Retrieved from *URL*:

https://machinelearningmastery.com/k-fold-cross-validation/

[3] Title-Model Training and Tuning. Retrieved from *URL*:

https://topepo.github.io/caret/model-training-and-tuning.html#control

[4] Title-Tune Machine Learning Algorithms in R. Retrieved from *URL*:

https://machinelearningmastery.com/tune-machine-learning-algorithms-in-r/

[5] Title-Random Forest. Retrieved from *URL*:

https://www.rdocumentation.org/packages/randomForest/versions/4.6-14/topics/randomForest

[6] Title-Train Control. Retrieved from *URL*:

https://www.rdocumentation.org/packages/caret/versions/6.0-84/topics/trainControl

[7] Title-Train. Retrieved from *URL*:

https://www.rdocumentation.org/packages/caret/versions/4.47/topics/train

Appendix

1. R- code for Decision Tree :-

```
#Importing the libraries

library(rpart)

library(rpart.plot)

#Import Dataset

car <- read.csv("car.csv", header = T)

car

# Splitting the datasets

set.seed(100)

train <- sample(nrow(car), 0.7*nrow(car), replace = FALSE)

TrainSet <- car[train,]

ValidSet <- car[-train,]

summary(TrainSet)

summary(ValidSet)

#Creating the tree

treecar = rpart(shouldBuy~., data = TrainSet, method = "class",  control =
rpart.control(minsplit = 30))

treecar

rpart.plot(treecar)

#Predict to create confusion matrix

predcar = predict(treecar, newdata = TrainSet, type = "class")

predcar

treeCar=table(TrainSet[,7],predcar)

treeCar

sum(diag(treeCar))/sum(treeCar)

predcarprob = predict(treecar, newdata = TrainSet, type = "prob")

predcarprob
```

```
library(pROC)

install.packages("pROC")

roc(TrainSet[,7],predcarprob[,2])

roc(TrainSet[,7],predcarprob[,1])

roc(TrainSet[,7],predcarprob[,3])

roc(TrainSet[,7],predcarprob[,4])

#ROC curve between Sensitivity and specificity

plot(roc(TrainSet[,7],predcarprob[,2]))

plot(roc(TrainSet[,7],predcarprob[,1]))

plot(roc(TrainSet[,7],predcarprob[,3]))

plot(roc(TrainSet[,7],predcarprob[,4]))

#-----------------------------------------

#Creating the tree while tuning minsplit =20

treecar1 = rpart(shouldBuy~., data = TrainSet, method = "class",  control =
rpart.control(minsplit = 20))

treecar1

predcar1 = predict(treecar1, newdata = TrainSet, type = "class")

predcar1

treeCar1 =table(TrainSet[,7],predcar1)

treeCar1

sum(diag(treeCar1))/sum(treeCar1)

#-----------------------------------------

#Creating the tree while tuning minsplit =10

treecar2 = rpart(shouldBuy~., data = TrainSet, method = "class",  control =
rpart.control(minsplit = 10))

treecar2

predcar2 = predict(treecar2 , newdata = TrainSet, type = "class")

predcar2

treeCar2 =table(TrainSet[,7],predcar2)
```

```
treeCar2

sum(diag(treeCar2))/sum(treeCar2)

#-------------------------Test Dataset-------------------------------

#Creating the tree

treecarvalid = rpart(shouldBuy~., data = ValidSet, method = "class",  control =
rpart.control(minsplit = 30))

treecarvalid

rpart.plot(treecarvalid)

#Predict to create confusion matrix

predcarvalid = predict(treecarvalid, newdata = ValidSet, type = "class")

predcarvalid

treeCarvalid =table(ValidSet[,7],predcarvalid)

treeCarvalid

sum(diag(treeCarvalid))/sum(treeCarvalid)



predcarprobvalid = predict(treecarvalid, newdata = ValidSet, type = "prob")

predcarprobvalid

library(pROC)

install.packages("pROC")

roc(ValidSet[,7],predcarprobvalid[,1])

roc(ValidSet[,7],predcarprobvalid[,2])

roc(ValidSet[,7],predcarprobvalid[,3])

roc(ValidSet[,7],predcarprobvalid[,4])

#ROC curve between Sensitivity and specificity

plot(roc(TrainSet[,7],predcarprob[,2]))

plot(roc(TrainSet[,7],predcarprob[,1]))

plot(roc(TrainSet[,7],predcarprob[,3]))
```

```
plot(roc(TrainSet[,7],predcarprob[,4]))
```

2. R- code for Random Forest :-

R-Script :-

```
library(randomForest)
library(pROC)
carData=read.csv("car.csv",header=T)
summary(carData)
set.seed(100)
#Constructing training and test dataset
train <- sample(nrow(carData), 0.7*nrow(carData), replace = FALSE)
TrainSet <- carData[train,]
TestSet <- carData[-train,]
summary(TrainSet)
summary(TestSet)
#Constructing model with defalut parameters
xc=TrainSet[,1:6]
yc=TrainSet[,7]
rf=randomForest(xc,yc)
rfp=predict(rf,xc)
rfCM=table(rfp,yc)
rfCM
sum(diag(rfCM))/sum(rfCM)
rfProb=predict(rf,xc,type="prob")
roc(TrainSet[,7],rfProb[,1])
plot(roc(TrainSet[,7],rfProb[,1]))
#Constructing tuned model with training dataset
xc=TrainSet[,1:6]
```

```
yc=TrainSet[,7]
rf=randomForest(xc,yc,ntree = 500,mtry = 4,nodesize = 10)
rfp=predict(rf,xc)
rfCM=table(rfp,yc)
rfCM
sum(diag(rfCM))/sum(rfCM)
rfProb=predict(rf,xc,type="prob")
roc(TrainSet[,7],rfProb[,2])
plot(roc(TrainSet[,7],rfProb[,2]))

#validating model on test dataset
xc=TestSet[,1:6]
yc=TestSet[,7]
rf=randomForest(xc,yc,ntree = 500,mtry = 4,nodesize = 10)
rfp=predict(rf,xc)
rfCM=table(rfp,yc)
rfCM
sum(diag(rfCM))/sum(rfCM)
rfProb=predict(rf,xc,type="prob")
roc(TestSet[,7],rfProb[,2])
plot(roc(TestSet[,7],rfProb[,2]))
```

3. R- code for Random Forest using K-Fold Cross Validation: -

```
# Random Forest with Default values but not fine-tuned.
library(caret)
library(randomForest)
```

```
control <- trainControl(method="repeatedcv", number=10, repeats=3)

seed <- 7

metric <- "Accuracy"

set.seed(seed)

rf_default <- train(x,y, method="rf", metric=metric, trControl=control)

print(rf_default)

varImp(rf_default)

ggplot(varImp(rf_default))
```

 # Random Forest using 'Random Search' and 'TuneLength' Parameters

```
library(caret)

control <- trainControl(method="repeatedcv", number=10, repeats=3, verboseIter = TRUE,
search = "random")

seed <- 7

metric <- "Accuracy"

set.seed(seed)

rf_search <- train(x,y, method="rf", metric=metric,tuneLength = 10, trControl=control)

print(rf_search)

plot(rf_search)
```

 # Random Forest using TuneGrid parameter for all mtry values.

```
library(caret)

control <- trainControl(method="repeatedcv", number=10, repeats=3, verboseIter = TRUE)

seed <- 7

metric <- "Accuracy"

set.seed(seed)
```

```
cmtry <- c(1,2,3,4,5,6)
tunegrid <- expand.grid(.mtry=cmtry)
rf_tuneGrid <- train(x,y, method="rf", metric=metric,tuneGrid = tunegrid, trControl=control)
print(rf_tuneGrid)
plot(rf_tuneGrid)
```

 # tuneRF for OOB vs mtry

```
trf<-tuneRF(x,y,stepFactor = 1.5, ntreeTry = 400, doBest = TRUE)
```

 # Final Model fitted with TuneGrid parameter for mtry = 2 && 4

```
library(caret)
control <- trainControl(method="repeatedcv", number=10, repeats=3, verboseIter = TRUE)
seed <- 7
metric <- "Accuracy"
set.seed(seed)
cmtry <- c(2,4)
tunegrid <- expand.grid(.mtry=cmtry)
rf_2_vs_4 <- train(x,y, method="rf", metric=metric,tuneGrid = tunegrid, trControl=control)
print(rf_2_vs_4)
plot(rf_2_vs_4)
varImp(rf_2_vs_4)
ggplot(varImp(rf_2_vs_4))
```