# Big Data - Assignment 1

*Aditya Bhadoria*

J1719461
(mailto:#)adityavsb@gmail.com (mailto:adityavsb@gmail.com)

*Adrián Ramírez*

47292486R
(mailto:#)adrian.ramirez.rio@gmail.com (mailto:adrian.ramirez.rio@gmail.com)

*8th October 2015*

# Modeling the sinking of the RMS Titanic

## Needed libraries

```
library(caret)
```

```
## Loading required package: lattice
## Loading required package: ggplot2
```

```
library(rpart)
library(rpart.plot)
library(randomForest)
```

```
## randomForest 4.6-12
## Type rfNews() to see new features/changes/bug fixes.
```

## Cleaning the data and building train and test sets

### Getting the data

Read data and transform PassengerId, Survived, Pclass, SibSp and Parch to factor variables.

```
passengers = read.csv("/home/adri/Escritorio/BigData/Coursework2/titanic/titanic.csv", na.strings = "")
passengers$PassengerId = as.factor(passengers$PassengerId)
passengers$Survived = as.factor(passengers$Survived)
passengers$Pclass = as.factor(passengers$Pclass)
passengers$SibSp = as.factor(passengers$SibSp)
passengers$Parch = as.factor(passengers$Parch)
summary(passengers)
```

```
##   PassengerId  Survived Pclass
##  1      :  1   0:549    1:216
##  2      :  1   1:342    2:184
##  3      :  1            3:491
##  4      :  1
##  5      :  1
##  6      :  1
##  (Other):885
##                                     Name         Sex           Age
##  Abbing, Mr. Anthony                   :  1   female:314   Min.   : 0.42
##  Abbott, Mr. Rossmore Edward           :  1   male  :577   1st Qu.:20.12
##  Abbott, Mrs. Stanton (Rosa Hunt)      :  1                Median :28.00
##  Abelson, Mr. Samuel                   :  1                Mean   :29.70
##  Abelson, Mrs. Samuel (Hannah Wizosky):  1                3rd Qu.:38.00
##  Adahl, Mr. Mauritz Nils Martin        :  1                Max.   :80.00
##  (Other)                               :885                NA's   :177
##  SibSp   Parch        Ticket         Fare                Cabin
##  0:608   0:678   1601    :  7   Min.   :  0.00   B96 B98    :  4
##  1:209   1:118   347082  :  7   1st Qu.:  7.91   C23 C25 C27:  4
##  2: 28   2: 80   CA. 2343:  7   Median : 14.45   G6         :  4
##  3: 16   3:  5   3101295 :  6   Mean   : 32.20   C22 C26    :  3
##  4: 18   4:  4   347088  :  6   3rd Qu.: 31.00   D          :  3
##  5:  5   5:  5   CA 2144 :  6   Max.   :512.33   (Other)    :186
##  8:  7   6:  1   (Other) :852                    NA's       :687
##  Embarked
##  C   :168
##  Q   : 77
##  S   :644
##  NA's:  2
##
##
##
```

# Clean data

## **Question 1:** Which are the variables that are discarded and why?

First we have to remove from the dataframe those variables that are not considered meaningful, i.e. literals, strings and ids (‘PassengerId’, ‘Name’ and ‘Ticket’).

```
data = subset(passengers, select=-c(PassengerId, Name, Ticket))
```

Second, we must discard or impute (out of the scope) those patterns with missing values. For that purpose, we first look for missing values in each variable.

```
sapply(data, function(x) sum(is.na(x)))
```

```
## Survived   Pclass      Sex      Age    SibSp    Parch     Fare    Cabin
##        0        0        0      177        0        0        0      687
## Embarked
##        2
```

We find 3 variables with missing values: ‘Age’, ‘Cabin’ and ‘Embarked’. The ‘Cabin’ variable has 687 missing values, which means that 77.1043771% of the passengers have no assigned Cabin. For this reason we consider this variable can also be discarded.

```
data = subset(data, select=-c(Cabin))
```

With respect to the ‘Age’ variable, only 19.8653199% passengers don’t have a value, so we can discard those passengers for the analysis getting still a dataset with 714 patterns.

```
data = na.omit(data)
```

We check again for missing values and see there aren’t any now.

```
sapply(data, function(x) sum(is.na(x)))
```

```
## Survived   Pclass      Sex      Age    SibSp    Parch     Fare Embarked
##        0        0        0        0        0        0        0        0
```

A summary of the resulting dataset is shown below

```
summary(data)
```

```
##  Survived Pclass      Sex           Age        SibSp   Parch
##  0:424    1:184   female:259   Min.   : 0.42   0:469   0:519
##  1:288    2:173   male  :453   1st Qu.:20.00   1:183   1:110
##           3:355                Median :28.00   2: 25   2: 68
##                                Mean   :29.64   3: 12   3:  5
##                                3rd Qu.:38.00   4: 18   4:  4
##                                Max.   :80.00   5:  5   5:  5
##                                                8:  0   6:  1
##       Fare        Embarked
##  Min.   :  0.00   C:130
##  1st Qu.:  8.05   Q: 28
##  Median : 15.65   S:554
##  Mean   : 34.57
##  3rd Qu.: 33.00
##  Max.   :512.33
##
```

# Create train and test datasets

We will split the dataset into 80-20 % for train and test with respect to the target variable ‘Survived’, keeping the balance between number of patterns from positive and negative classes (this feature is provided by the createDataPartition function).

```
#Set seed for random numbers to build always the same datasets (to ease interpreta
tion)
set.seed(123)
train.idx = createDataPartition(data$Survived, p=0.8, list=FALSE)
trainData = data[train.idx,]
trainData.predictors = subset(trainData, select=-c(Survived))
trainData.target = trainData$Survived
testData = data[-train.idx,]
testData.predictors = subset(testData, select=-c(Survived))
testData.target = testData$Survived
```

We hence get a train dataset with 571 patterns and a test dataset with 141 patterns. Additionally two variables are created for each of these two datasets, one that contains the predictor variables and one that contains the targets (this will make code clearer in the rest of the assignment).

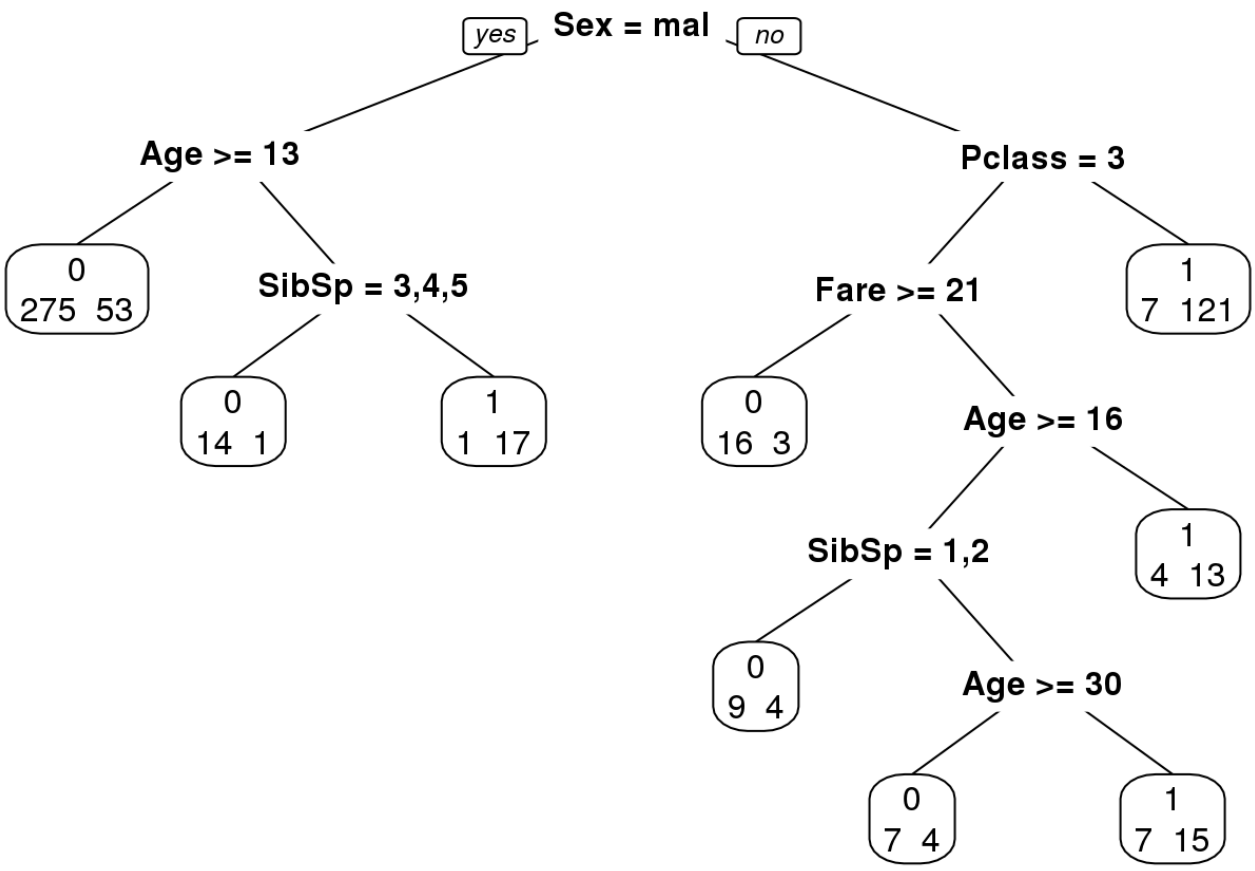
# Training a single decision tree

We will now proceed to train a decision tree (CART algorithm) using the train dataset.

```
defaultTree = rpart(Survived ~ ., trainData)
```

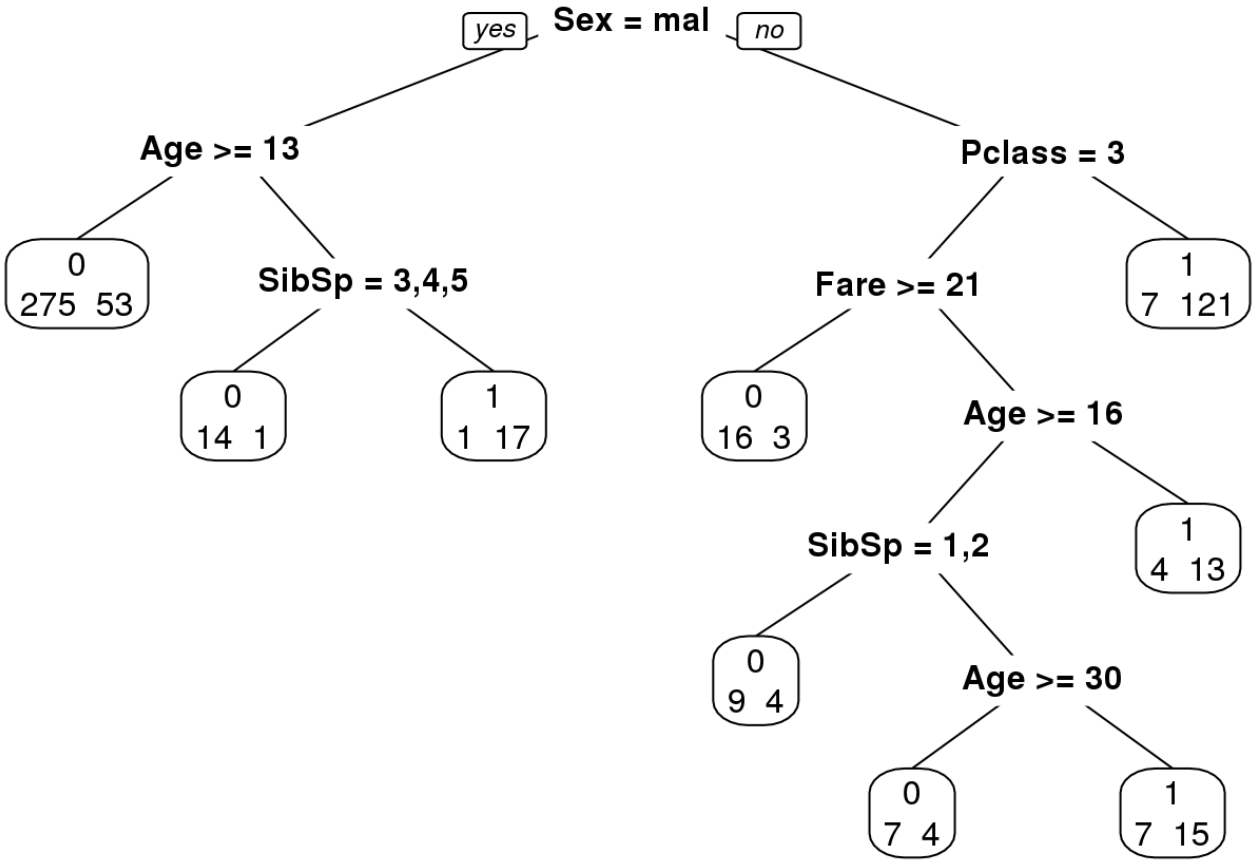
**Question 2:** Plot the tree with the prp() function. Which are the most important variables according to the tree? Does the tree change if we rerun the rpart() function? Why?

```
prp(defaultTree, extra=1)
```




As we can see in the plot, the most important variable (first one selected to split) is the ‘Sex’ of the passenger, followed by the ‘Age’ and the passenger’s class (‘Pclass’). ‘Fare’ and ‘SibSp’ are also important somehow, since they are present in the tree decision making. The variables that are not present (‘Parch’ and ‘Embarked’) are not considered important by the model. Now we rerun the rpart function

```
tree.rerun = rpart(Survived ~ ., trainData)
prp(tree.rerun, extra=1)
```




and see the tree hasn't changed at all. This is because the CART algorithm is completely deterministic, in other words, there is not any kind of randomness in the form the tree is constructed from the data, so provided the same data and parameter values, the constructed decision tree will always be the same.

## **Question 3:** Which are the performance values (accuracy, sensitivity, specificity, etc.) of the learned model on the testing subset?

To see the accuracy of the model in the test data, we use the method predict and then we build the confusion matrix, where we find the desired performance values.

```
defaultCM = confusionMatrix(predict(defaultTree, newdata=testData, type="class"),
testData.target)
defaultCM
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 77 20
##          1  7 37
##
##                Accuracy : 0.8085
##                  95% CI : (0.7338, 0.8699)
##     No Information Rate : 0.5957
##     P-Value [Acc > NIR] : 5.647e-08
##
##                   Kappa : 0.5873
##  Mcnemar's Test P-Value : 0.02092
##
##             Sensitivity : 0.9167
##             Specificity : 0.6491
##          Pos Pred Value : 0.7938
##          Neg Pred Value : 0.8409
##              Prevalence : 0.5957
##          Detection Rate : 0.5461
##    Detection Prevalence : 0.6879
##       Balanced Accuracy : 0.7829
##
##        'Positive' Class : 0
##
```

## **Question 4:** Which are the values for the parameters of this algorithm (minbucket, minsplit, complexity parameter, cost, etc.)?

We check the rpart function documentation.

```
?rpart
```

And we find that the parameters of the model are specified in the control and cost arguments. For the cost, the default value is a vector of ones of length the number of variables:

> cost
> a vector of non-negative costs, one for each variable in the model. Defaults to one for all variables. These are scalings to be applied when considering splits, so the improvement on splitting on a variable is divided by its cost in deciding which split to choose.

For the control argument, the default values are those in the rpart.control

```
?rpart.control
```

And we find the default values to be:

> rpart.control(minsplit = 20, minbucket = round(minsplit/3), cp = 0.01, maxcompete = 4, maxsurrogate = 5, usesurrogate = 2, xval = 10, surrogatestyle = 0, maxdepth = 30, …)

**Question 5:** Try different combinations of values for some of the parameters (decreasing minsplit, minbucket, cp and cost values, for example) and check the performance of each combination on the testing subset. How does this performance change? How do the obtained trees change? Is there any relationship between the parameters values and the shape of the trees?

To measure the change in performance by the different trees built for the different parameter values, we will focus on the Accuracy metric performance. The baseline confusion matrix, that allows us to check changes in performance, is the one shown below (with it's corresponding accuracy).

```
defaultCM$table
```

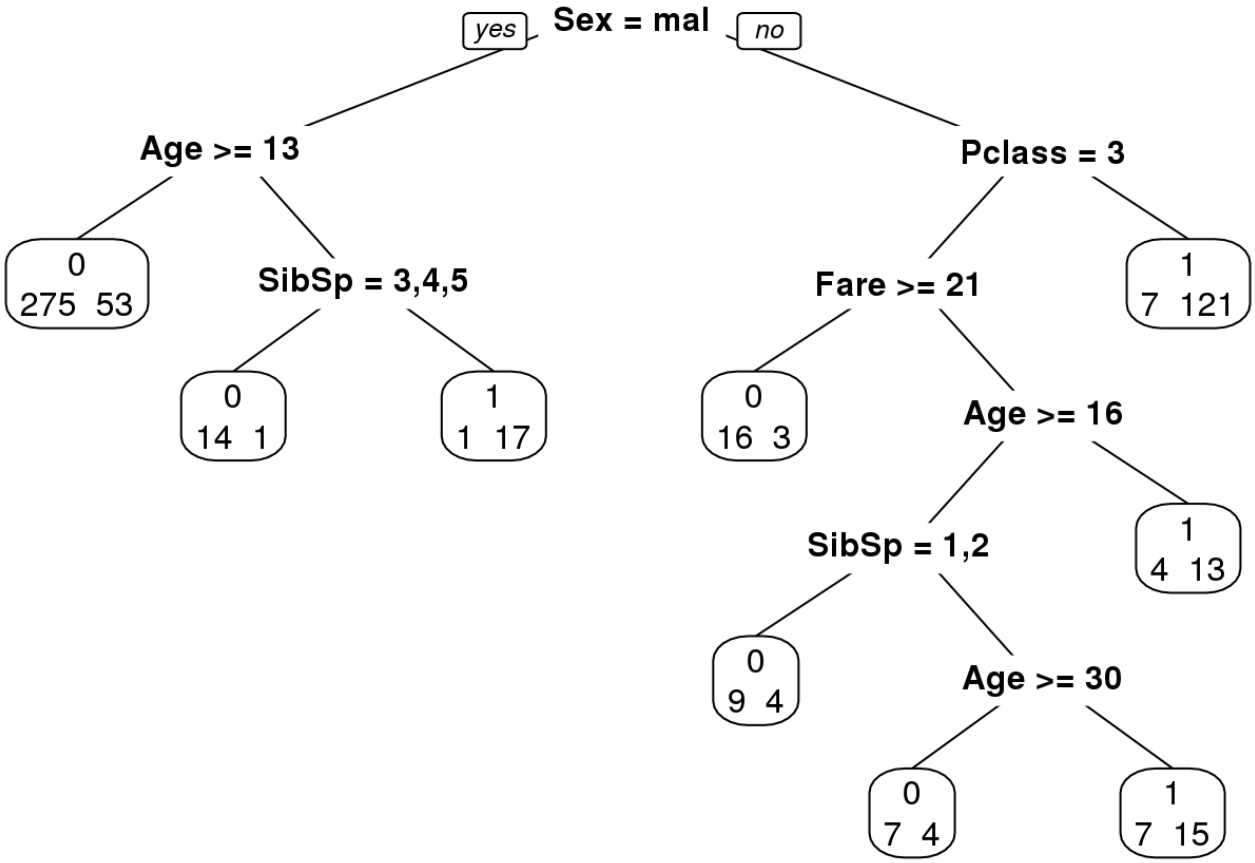
```
##           Reference
## Prediction  0  1
##          0 77 20
##          1  7 37
```

```
defaultCM$overall[1]
```

```
##  Accuracy
## 0.8085106
```

**1.** minsplit = 5 (while maintaining minbucket=7 which is the default value):

```
tunedTree = rpart(Survived ~ ., trainData, control=rpart.control(minsplit=5, minbu
cket=7))
prp(tunedTree, extra=1)
```




```
tunedCM = confusionMatrix(predict(tunedTree, newdata=testData, type="class"), test
Data.target)
tunedCM$table
```

```
##          Reference
## Prediction  0  1
##          0 77 20
##          1  7 37
```

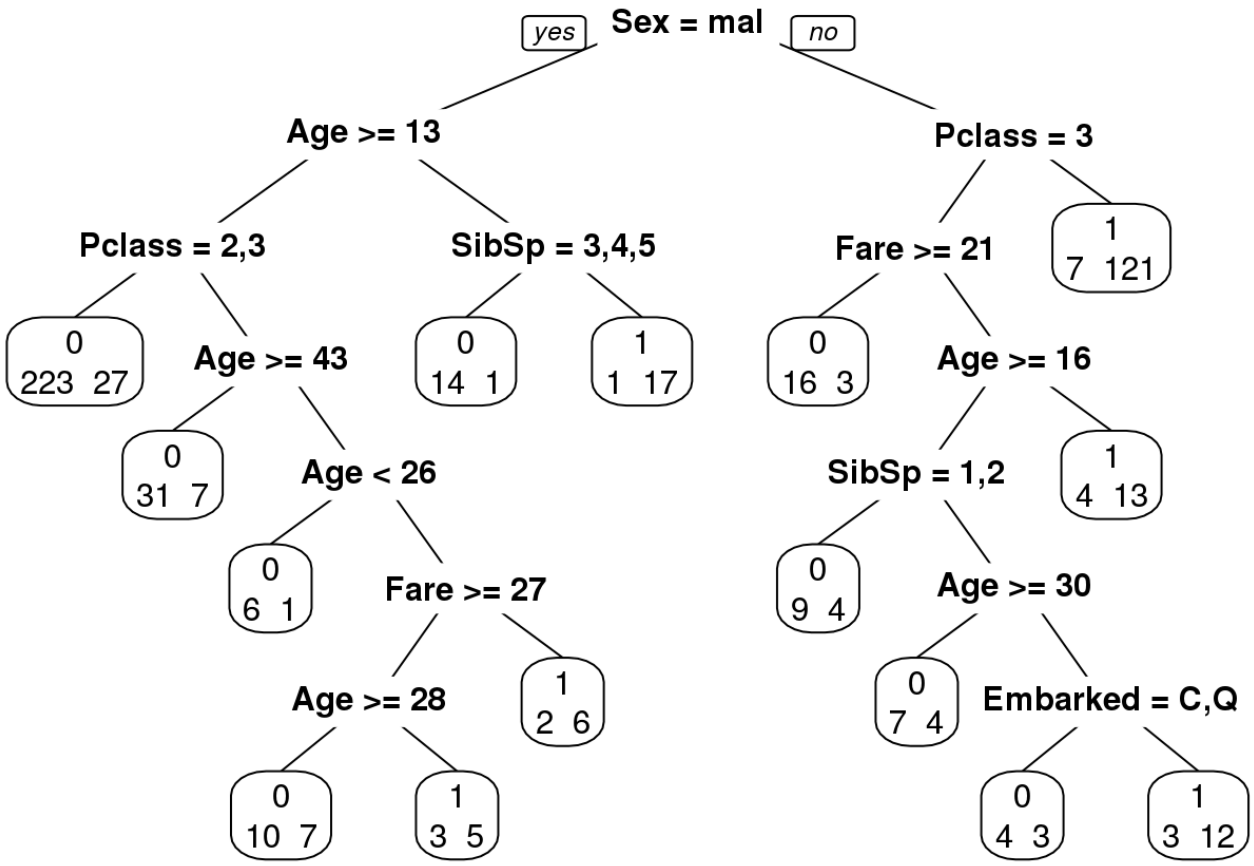
```
tunedCM$overall[1]
```

```
##  Accuracy
## 0.8085106
```

The built tree hasn't changed so it's performance (see the confusion matrix above) hasn't changed either.

**2.** cp = 0.001:

```
tunedTree = rpart(Survived ~ ., trainData, control=rpart.control(cp=0.001))
prp(tunedTree, extra=1)
```




```
tunedCM = confusionMatrix(predict(tunedTree, newdata=testData, type="class"), test
Data.target)
```

Now the tree shape is partially different. This tree is an evolution of the previous one. Conditions have not changed but the tree has grown deeper (it makes sense since the cp parameter is used for pre-prunning).

Change in performance:

```
tunedCM$table
```

```
##           Reference
## Prediction  0  1
##          0 75 20
##          1  9 37
```

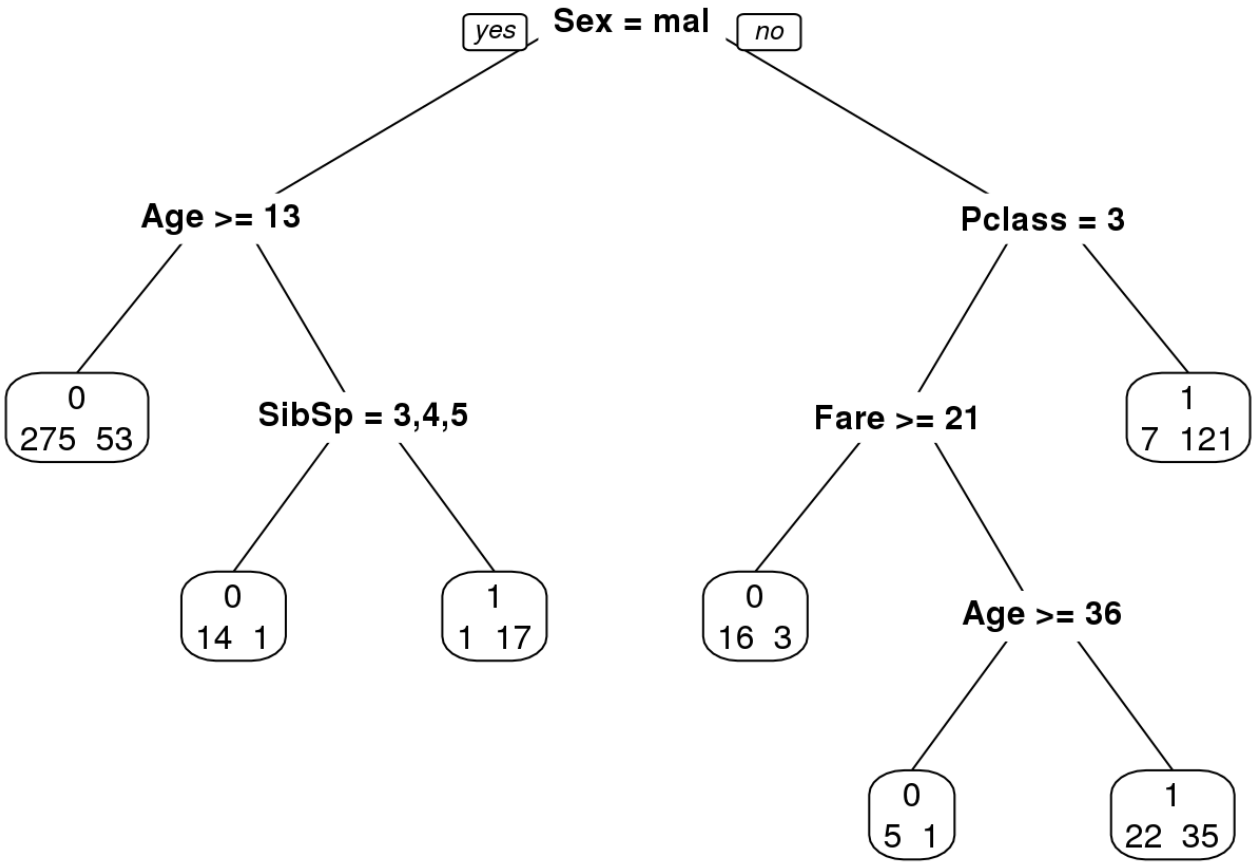
```
tunedCM$overall[1]
```

```
##  Accuracy
## 0.7943262
```

False-positive cases grow while false-negative ones hold and hence performance falls, as we see, for example, in the accuracy.

**3.** minbucket = 6 (while maintaining minsplit=20):

```
tunedTree = rpart(Survived ~ ., trainData, control=rpart.control(minsplit=20, minb
ucket=6))
prp(tunedTree, extra=1)
```




```
tunedCM = confusionMatrix(predict(tunedTree, newdata=testData, type="class"), test
Data.target)
```

There is a change in its shape in the right branch, where after splitting by the condition ‘Fare>=21’ then we check the condition ‘Age>=36’ rather than ‘Age>=16’ (as happened in the previous case). This happens because the condition ‘Age>=16’ gives as result a leaf node with 6 patterns, that previously was discarded because minbucket was 7 (>=6).

Change in performance:

```
tunedCM$table
```

```
##          Reference
## Prediction  0  1
##          0 75 19
##          1  9 38
```

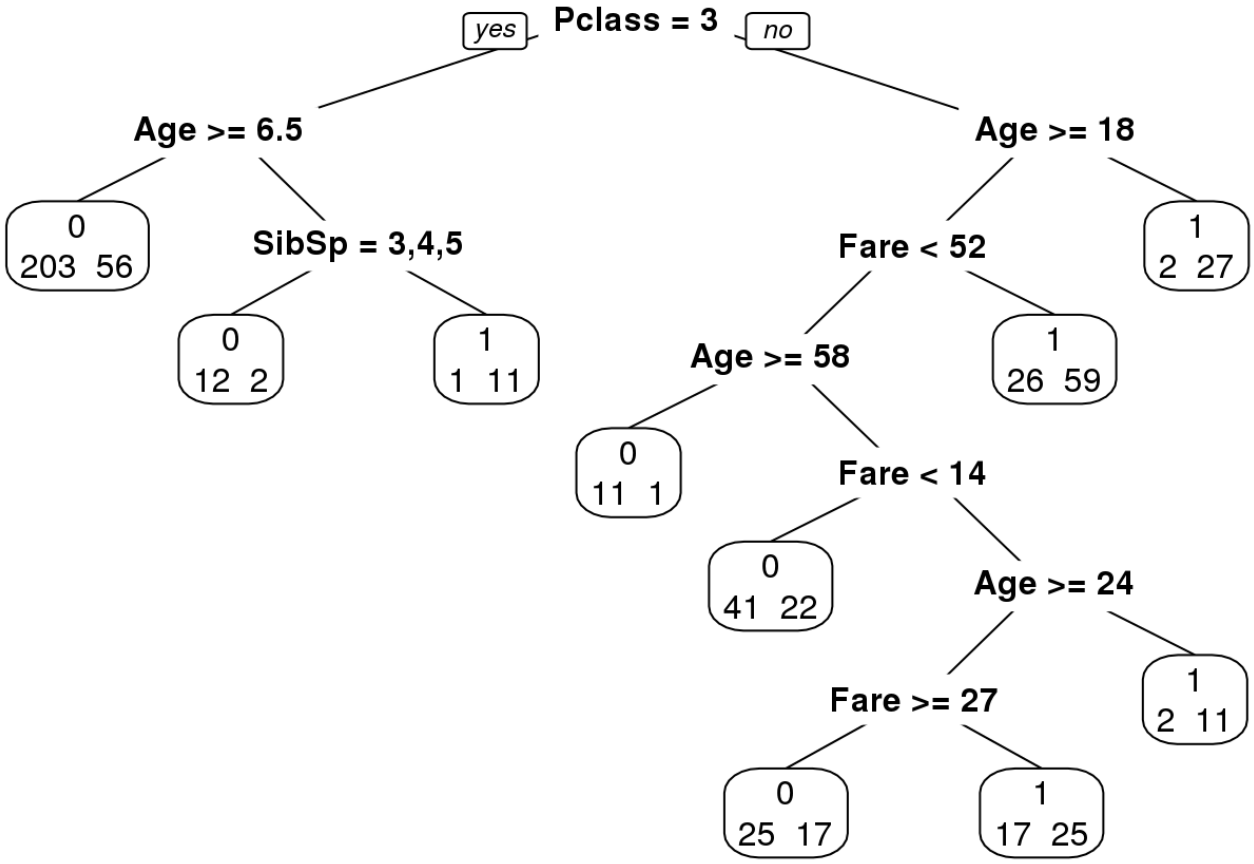
```
tunedCM$overall[1]
```

```
##  Accuracy
## 0.8014184
```

While there are two more false-positives there is one less false-negative so the overall accuracy falls just a bit.

**4.** cost = 100 for variable Sex (2nd component):
This should make the algorithm prefer to split by any variable rather than Sex.

```
tunedTree = rpart(Survived ~ ., trainData, cost=c(1,100,1,1,1,1,1))
prp(tunedTree, extra=1)
```



```
tunedCM = confusionMatrix(predict(tunedTree, newdata=testData, type="class"), test
Data.target)
```

As we see the tree has totally changed since the Sex variable is no longer used to build the tree.

Change in performance:

```
tunedCM$table
```

```
##          Reference
## Prediction  0  1
##          0 68 16
##          1 16 41
```

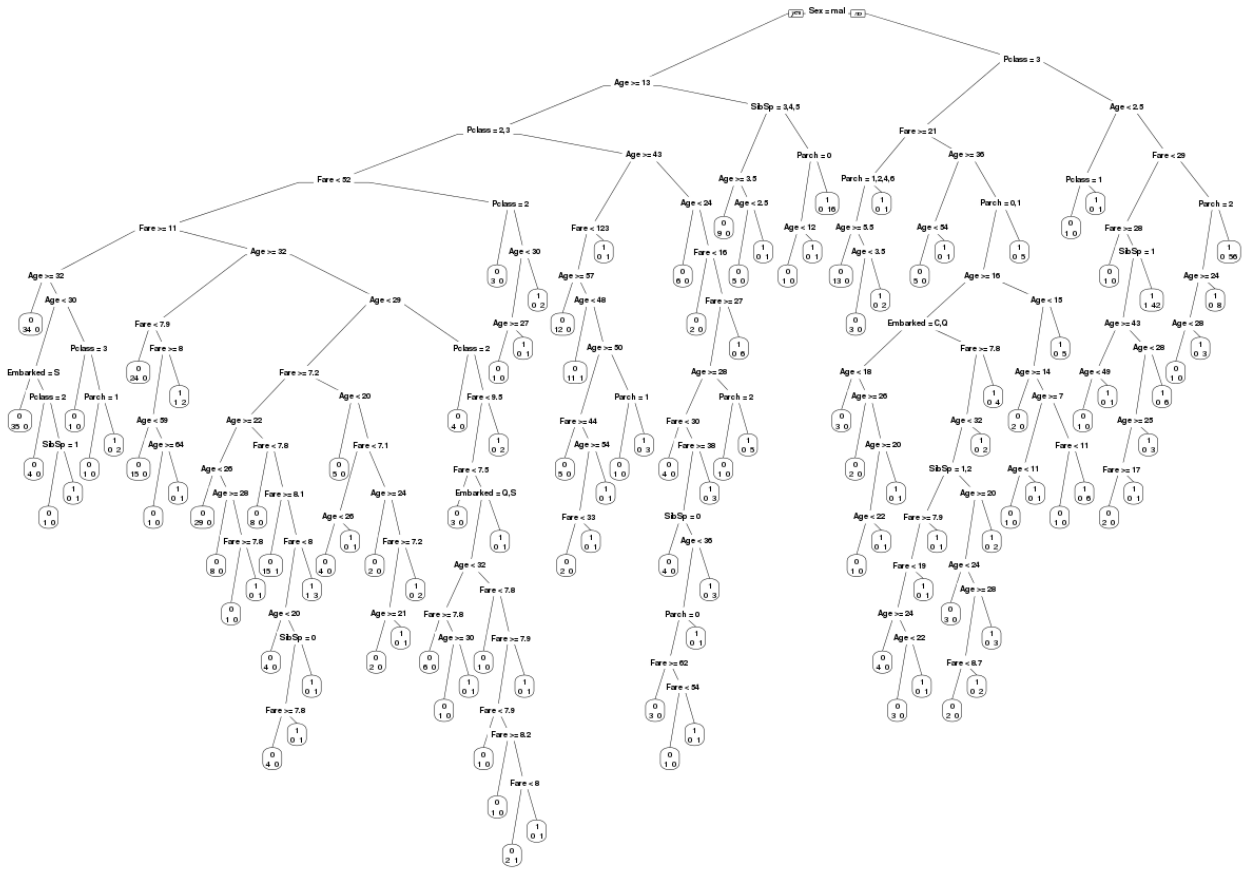
```
tunedCM$overall[1]
```

```
##  Accuracy
## 0.7730496
```

Both false-positive and false-negatives raise, hence the accuracy drops down.

**5.** minsplit=0, minbucket=0, cp=0:
With these values we intend to build the full tree, this is, the tree where each node is pure (only patterns from one class). This might not be achievable because the maximum depth that the algorithm allows us to set is 30, but still we can get a pretty big tree.

```
tunedTree = rpart(Survived ~ ., trainData, control=rpart.control(minsplit=0, minbu
cket=0, cp=0))
prp(tunedTree, extra=1)
```



```
tunedCM = confusionMatrix(predict(tunedTree, newdata=testData, type="class"), test
Data.target)
```

This kind of trees will normally overfit to the training data, since they specialize too much in the examples we show them.

Change in performance:

```
tunedCM$table
```

```
##          Reference
## Prediction  0  1
##          0 67 17
##          1 17 40
```

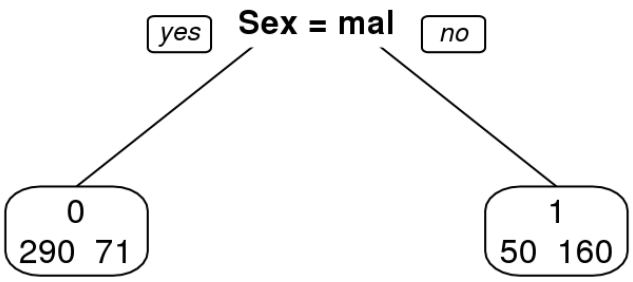
```
tunedCM$overall[1]
```

```
##  Accuracy
## 0.7588652
```

We can notice the tree is overfitting since its performance is even worse that the one from the previous case.

**6.** minsplit=50, minbucket=50, cp=0.1:
Contrarily to letting the tree grow, with these values we will try to get a really simple tree, this is, a tree with only two terminal nodes.

```
tunedTree = rpart(Survived ~ ., trainData, control=rpart.control(minsplit=50, minb
ucket=50, cp=0.1))
prp(tunedTree, extra=1)
```




```
tunedCM = confusionMatrix(predict(tunedTree, newdata=testData, type="class"), test
Data.target)
```

What we expect with these simple trees is to suffer some kind of underfitting, in other words, a lack of capability from the model to explain the data.

Change in performance:

```
tunedCM$table
```

```
##          Reference
## Prediction  0  1
##          0 70 22
##          1 14 35
```

```
tunedCM$overall[1]
```

```
##  Accuracy
## 0.7446809
```

Again, we confirm that the tree is underfitting since it's performance is again a lot worse than previous studied trees.

# Automatically tuning the parameters of a decision tree

For automatically tuning the parameters in the tree we use the train function from caret package. We select the method rpartCost that will try to fit the values for the complexity parameter (cp) and the cost.

```
fitControl = trainControl(method="cv", number=10, search="grid")
treeFit = train(x=trainData.predictors, y=trainData.target, method="rpartCost", tr
Control=fitControl)
```

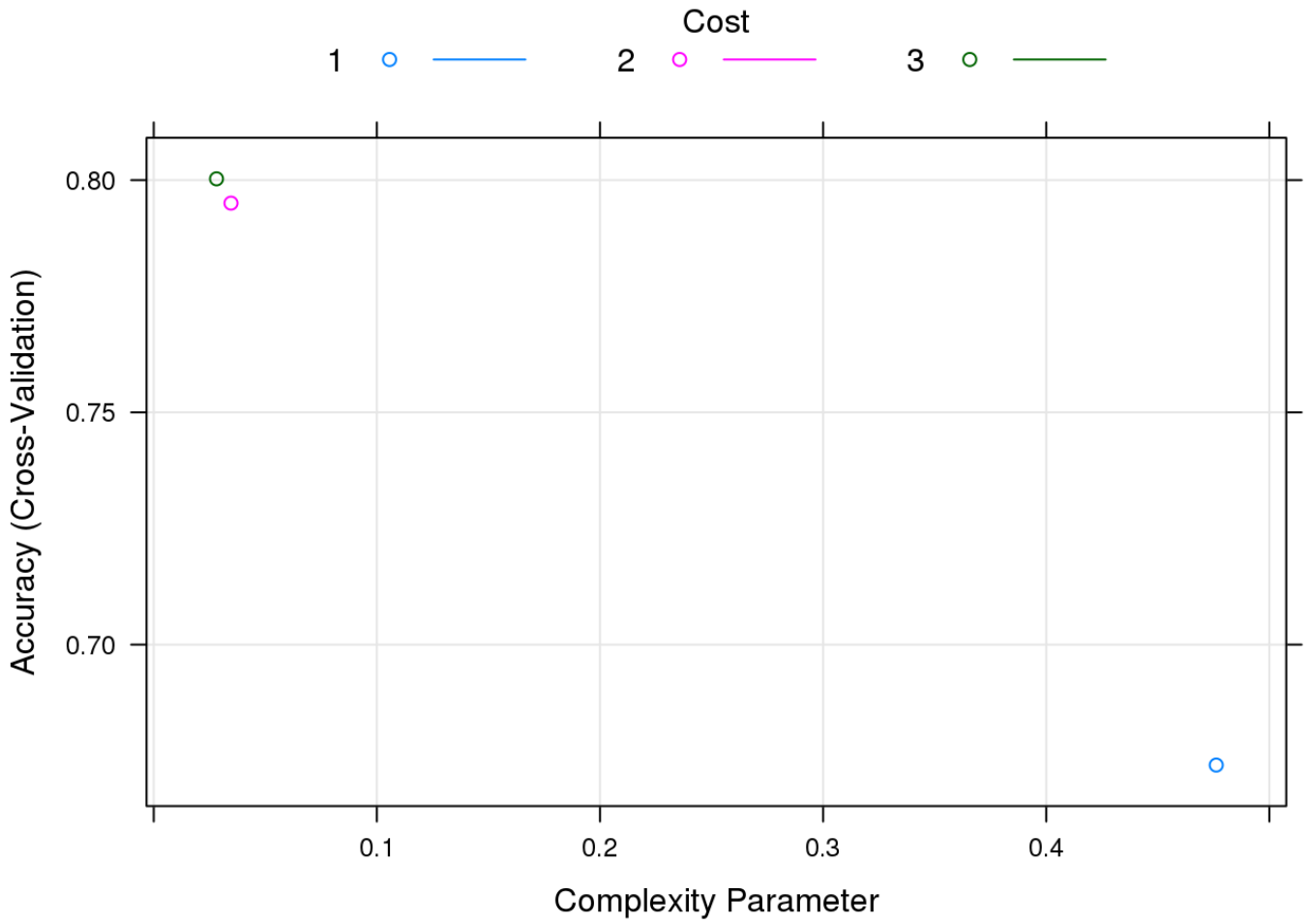
We have found what we believe could be a bug. When calling the train function with the configuration shown above (where by default tuneLength is 3), the function seems to be performing a random search for the parameter values, as only 3 points in the parameter space are being validated

```
treeFit$results
```

```
##   Cost         cp  Accuracy     Kappa AccuracySD    KappaSD
## 3    3 0.02813853 0.8002722 0.5530831 0.03842176 0.09296525
## 2    2 0.03463203 0.7950393 0.5408399 0.03543346 0.08485075
## 1    1 0.47619048 0.6740774 0.2390494 0.08573896 0.25735811
```

and no grid structure is found in the values of the complexity parameter.

```
plot(treeFit)
```




A workaround for it seems to be setting tuneLength>=10, so we will use this value from now on (except when the grid structure is actually passed to the train function).

```
treeFit = train(x=trainData.predictors, y=trainData.target, method="rpartCost", tr
Control=fitControl, tuneLength=10)
```

## **Question 6:** Which are the combinations of parameters values tested by the train() function? Are there any changes in the performance of the algorithm when different combinations of values are used (according to the results of the cross validation)?

We can see the combinations tested by the model below (sorted by Accuracy from higher to lower, and only showing top 5 and bottom 5).

```
head(treeFit$results[order(treeFit$results$Accuracy, decreasing=TRUE), ], 5)
```

```
##    Cost         cp  Accuracy     Kappa AccuracySD   KappaSD
## 11    2 0.05291005 0.7950091 0.5392402 0.06431675 0.1528362
## 20    3 0.05291005 0.7950091 0.5392402 0.06431675 0.1528362
## 29    4 0.05291005 0.7950091 0.5392402 0.06431675 0.1528362
## 38    5 0.05291005 0.7950091 0.5392402 0.06431675 0.1528362
## 47    6 0.05291005 0.7950091 0.5392402 0.06431675 0.1528362
```

```
tail(treeFit$results[order(treeFit$results$Accuracy, decreasing=TRUE), ], 5)
```

```
##    Cost        cp  Accuracy Kappa  AccuracySD KappaSD
## 54    6 0.4232804 0.5954628     0 0.003252191       0
## 63    7 0.4232804 0.5954628     0 0.003252191       0
## 72    8 0.4232804 0.5954628     0 0.003252191       0
## 81    9 0.4232804 0.5954628     0 0.003252191       0
## 90   10 0.4232804 0.5954628     0 0.003252191       0
```

Clearly, the performance of the model (check the Accuracy and Kappa metrics) is different depending on which values are used to train the model. We see this for example looking at the first and last models, where the Accuracy falls from 0.7950091 to 0.5954628

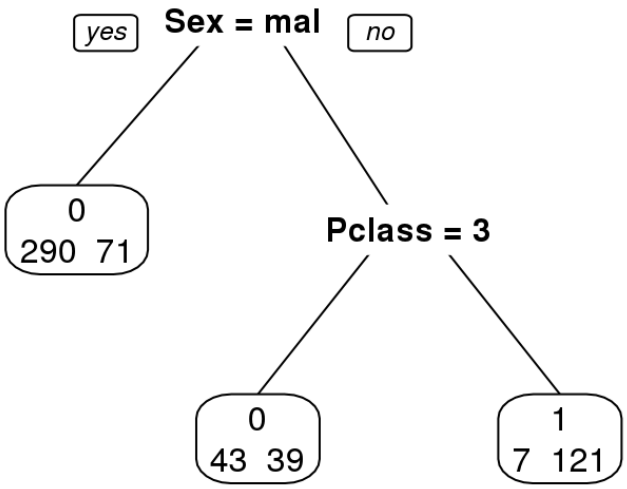
## **Question 7:** Which is the final combination of parameters values used? Which is the shape of the tree trained with this automatic tuning function?

We can check the selected parameters and plot the final model

```
treeFit$bestTune
```

```
##           cp Cost
## 40 0.1587302    5
```

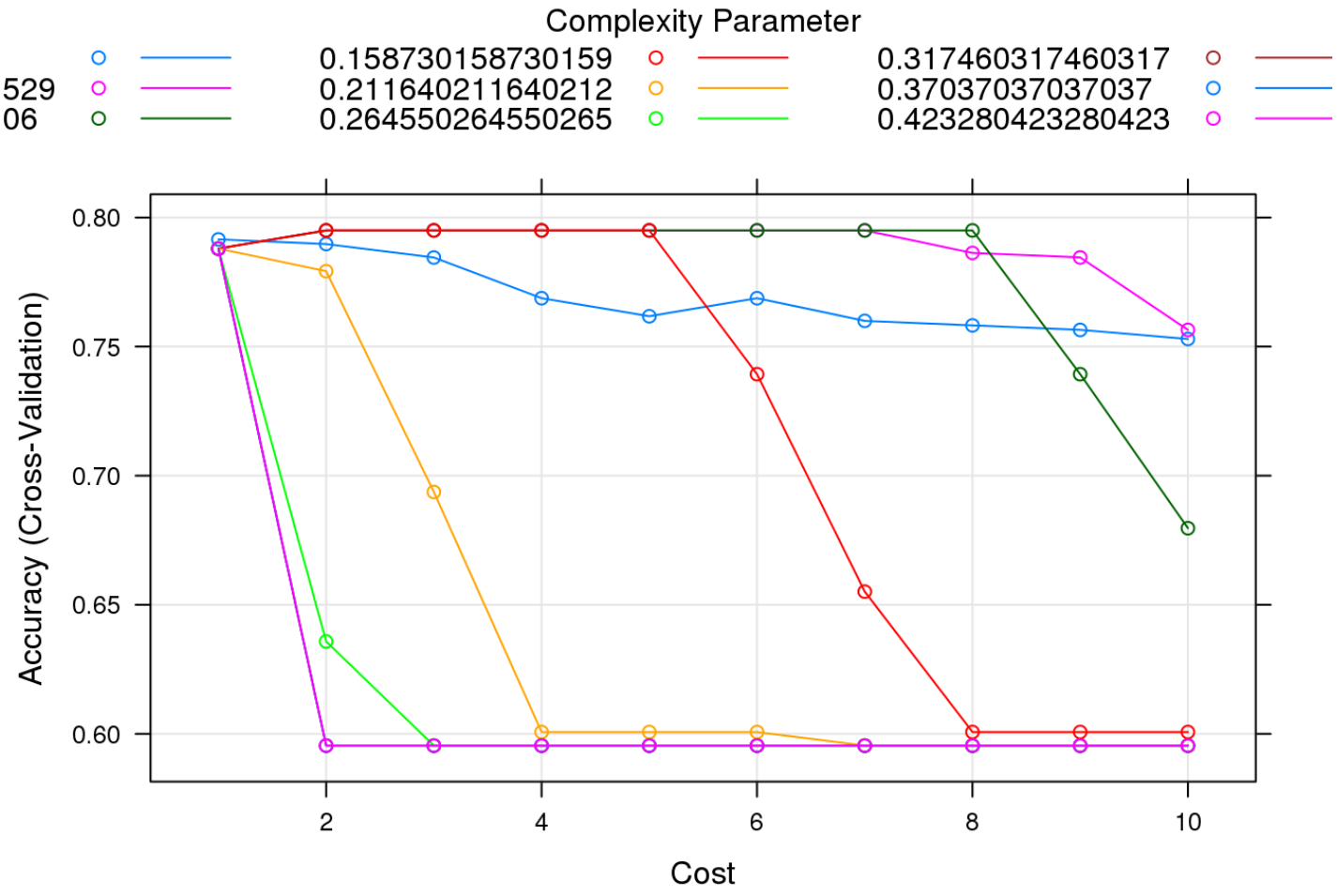
```
prp(treeFit$finalModel, extra=1)
```




As we see this is a very simple model where only 2 of the 7 predictive variables are considered.

## **Question 8:** Plot the result of calling the train() function with the plot() function. What does this plot represent?

```
plot(treeFit)
```




This plot represents the accuracy of the different models used in validation in function of the different values tried for the parameters. As we see the combination chosen by the method as the best (or final) model lies on the top of the plot although it is not the highest one. This is because for the best model selection the function train() not only takes into account the accuracy metric but also other performance metrics.

Now we will rerun the train function with a different set of possible values for the parameters (that we will choose).

```
rpartGrid = expand.grid(.Cost = c(1, 2, 3, 5, 10), .cp=c(0, 0.01, 0.02, 0.04, 0.0
7, 0.10))
treeFit = train(x=trainData.predictors, y=trainData.target, method="rpartCost", tr
Control=fitControl, tuneGrid=rpartGrid)
```

## **Question 9:** Which are the combinations of parameters values tested by the train() function? Are there any changes in the performance of the algorithm when different combinations of values are used (according to the results of the cross validation)?

Again, since we are trying different values for the parameters, the (some) built trees will be different and hence we will find different performances for different parameter combinations. Below we can see the list of the different combinations ordered by accuracy.

```
treeFit$results[order(treeFit$results$Accuracy, decreasing=TRUE), ]
```

```
##    Cost   cp  Accuracy     Kappa AccuracySD   KappaSD
## 3     1 0.02 0.8213854 0.6228662 0.05944493 0.1239301
## 2     1 0.01 0.8179068 0.6115213 0.05782572 0.1260113
## 1     1 0.00 0.8039020 0.5892406 0.06802769 0.1407962
## 8     2 0.02 0.8005142 0.5563576 0.07142996 0.1657810
## 13    3 0.02 0.7987598 0.5522992 0.07001765 0.1624571
## 9     2 0.04 0.7952511 0.5393905 0.05820709 0.1406189
## 14    3 0.04 0.7952511 0.5393905 0.05820709 0.1406189
## 19    5 0.04 0.7952511 0.5393905 0.05820709 0.1406189
## 10    2 0.07 0.7952511 0.5393905 0.05820709 0.1406189
## 15    3 0.07 0.7952511 0.5393905 0.05820709 0.1406189
## 20    5 0.07 0.7952511 0.5393905 0.05820709 0.1406189
## 7     2 0.01 0.7917423 0.5419573 0.07488753 0.1679756
## 12    3 0.01 0.7917423 0.5387725 0.06352171 0.1491125
## 6     2 0.00 0.7899577 0.5469331 0.07330545 0.1594310
## 4     1 0.04 0.7881125 0.5535529 0.07269873 0.1537015
## 5     1 0.07 0.7881125 0.5535529 0.07269873 0.1537015
## 18    5 0.02 0.7847247 0.5132769 0.05889390 0.1440311
## 25   10 0.07 0.7794616 0.5004699 0.05786165 0.1414124
## 17    5 0.01 0.7777072 0.4966136 0.05219331 0.1306322
## 11    3 0.00 0.7759226 0.5080311 0.08007363 0.1737916
## 24   10 0.04 0.7724440 0.4826047 0.05351196 0.1352281
## 23   10 0.02 0.7689353 0.4740118 0.05342154 0.1336281
## 22   10 0.01 0.7636721 0.4608733 0.05247722 0.1313814
## 16    5 0.00 0.7566546 0.4528874 0.07270601 0.1667380
## 21   10 0.00 0.7549002 0.4388887 0.05037280 0.1270147
```

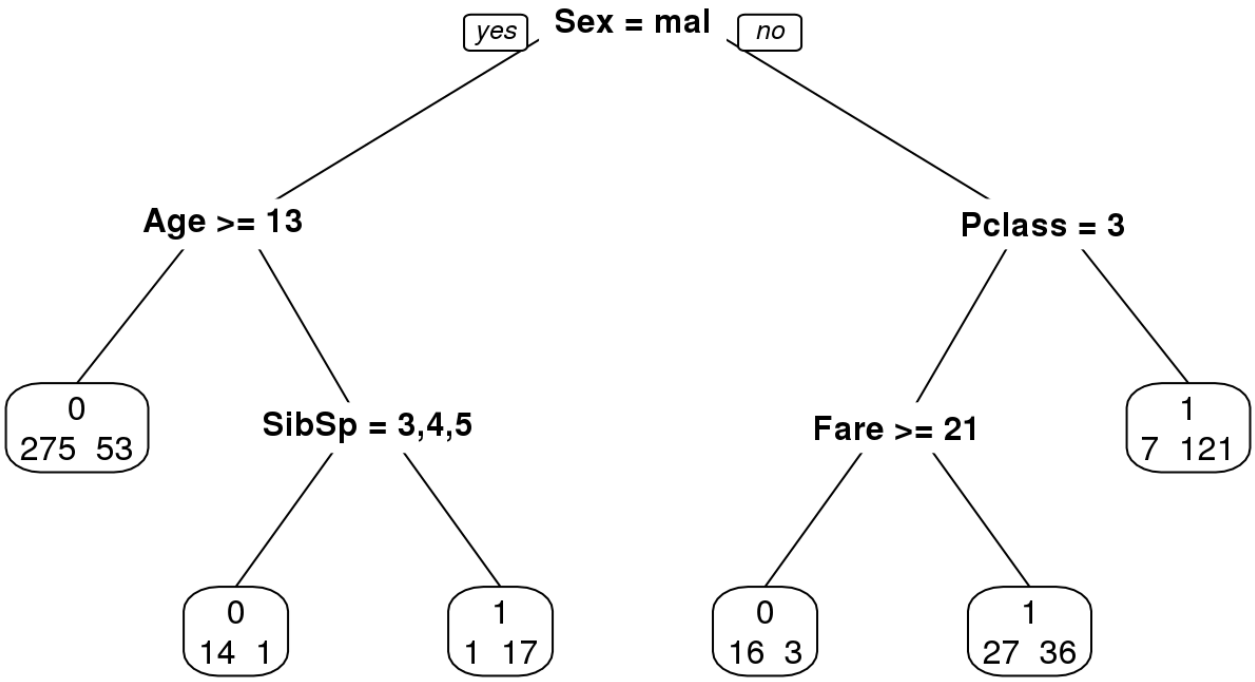
## **Question 10:** Which is the final combination of parameters values used? Which is the shape of the tree trained with this automatic tuning function?

As before we can ask for the parameters of the model best tuned and plot it.

```
treeFit$bestTune
```

```
##     cp Cost
## 3 0.02    1
```

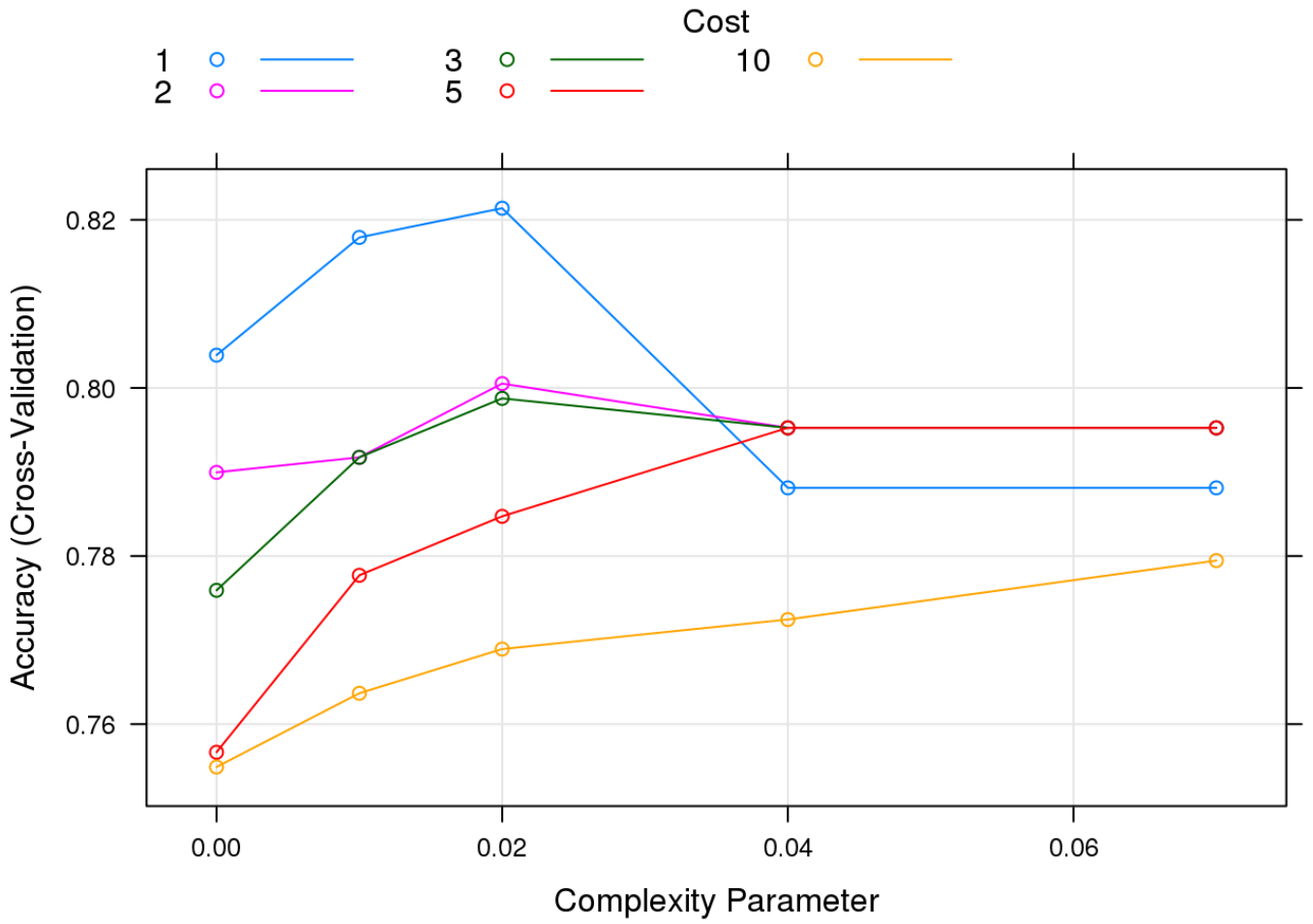
```
bestTree = treeFit$finalModel
prp(bestTree, extra=1)
```




Now the obtained tree is totally different; as can be seen in the plot it is a lot more complex, although some of the conditions at the top remain the same.

## **Question 11:** Plot the result of calling the train() function with the plot() function. What does this plot represent?

```
plot(treeFit)
```




Again this plot shows the Accuracy obtained during cross-validation for the different parameter combinations. As we can see the points represent the combinations of the values we have chosen for the parameters.

# Training a Random Forest (with default and custom parameters values)

First, we train a random forest with the default parameter values and 2000 trees.

```
rfFit= train(x=trainData.predictors, y=trainData.target, method="rf", trControl=fi
tControl, ntree=2000)
```

**Question 12:** Which are the combinations of parameters values tested by the train() function? Are there any changes in the performance of the algorithm when different combinations of values are used (according to the results of the cross validation)?

As in the previous section the validated models are shown below, ordered by accuracy.

```
rfFit$results[order(rfFit$results$Accuracy, decreasing=TRUE), ]
```

```
##   mtry  Accuracy     Kappa AccuracySD    KappaSD
## 1    2 0.8179371 0.6132180 0.05272237 0.11481648
## 2    4 0.8039322 0.5898575 0.04064252 0.08671769
## 3    7 0.7933152 0.5682547 0.04294796 0.09088297
```

One more time we find that the performance of the model changes when different values for the parameter mtry are chosen.

## **Question 13:** Which is the final combination of parameters values used?

The chosen value for the mtry parameter is

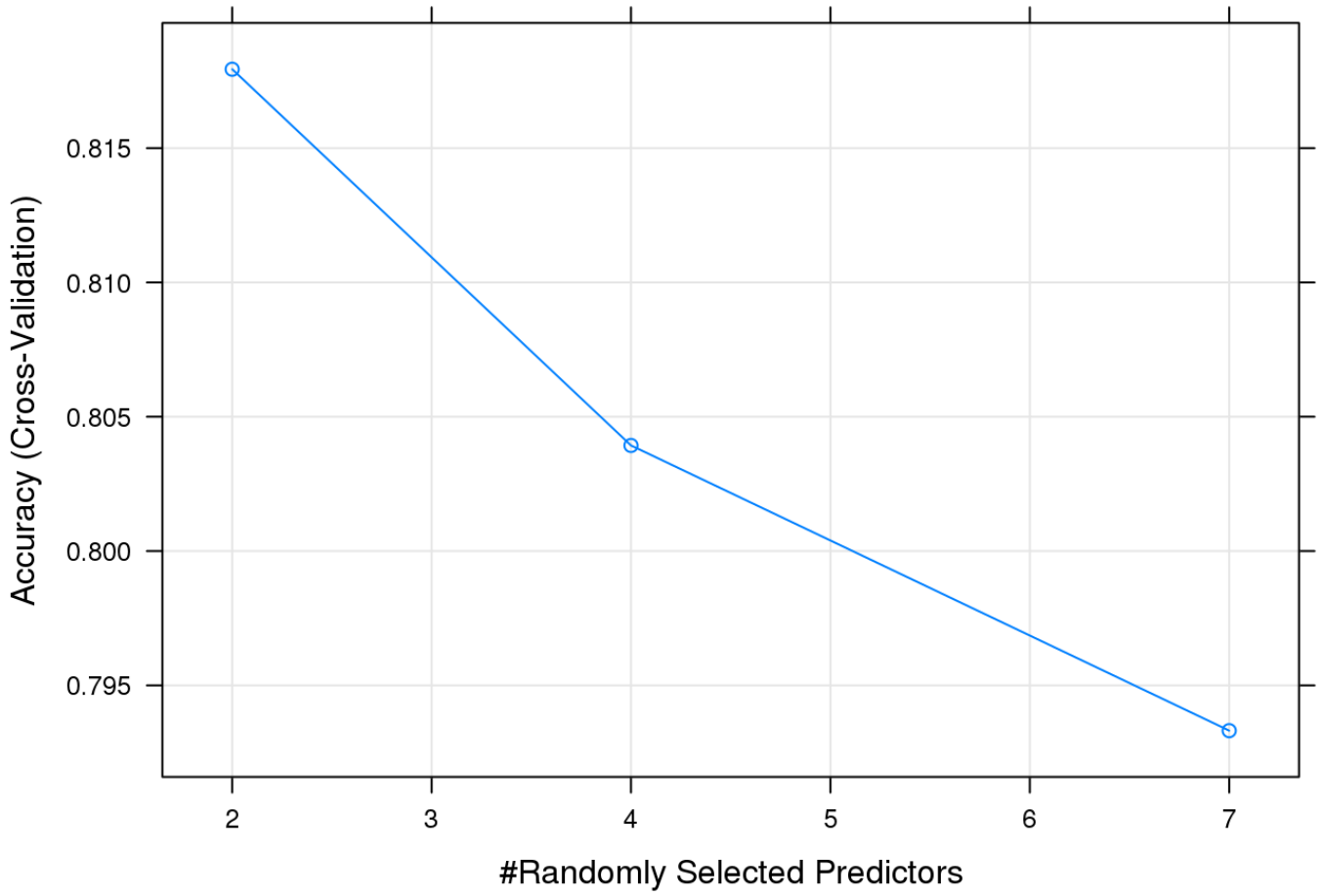
```
rfFit$bestTune
```

```
##   mtry
## 1    2
```

Note that here there is no combination, since only one parameter is being tuned.

## **Question 14:** Plot the result of calling the train() function with the plot() function. What does this plot represent?

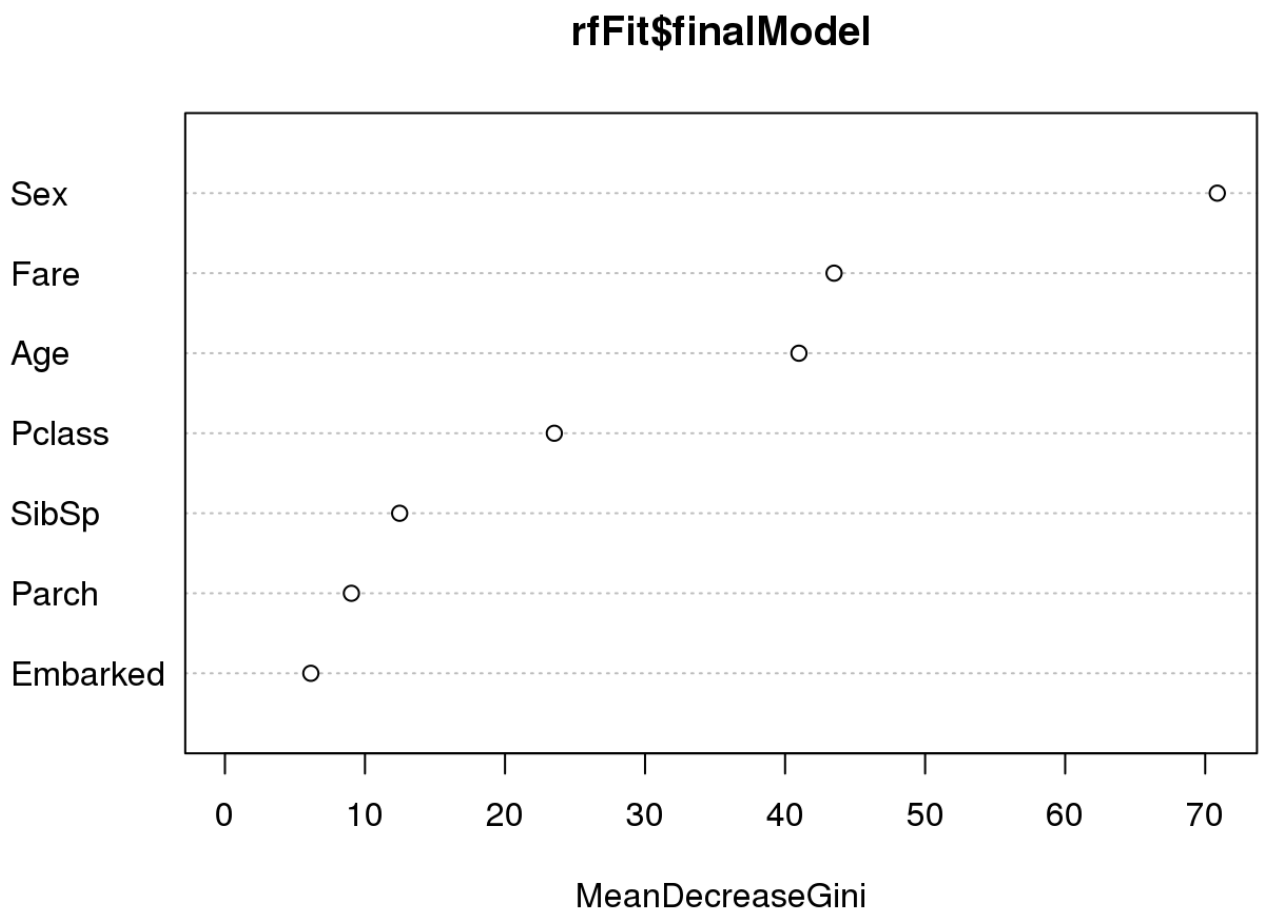
```
plot(rfFit)
```




What we see in the plot is the accuracy achieved by the different models with respect to the selected value for the parameter mtry (randomly selected predictors).

**Question 15:** Plot the importance of each variable for the model with function VarImPlot() from package caret. Which are the most relevant variables according to their mean decrease of the Gini index? Are these variables the ones selected when we built our decision trees?

```
varImpPlot(rfFit$finalModel)
```




```
importance(rfFit$finalModel)[order(importance(rfFit$finalModel), decreasing=TRU
E),]
```

```
##       Sex      Fare       Age    Pclass     SibSp     Parch  Embarked
## 70.862041 43.497702 40.988816 23.521050 12.476470  9.031263  6.135353
```

As we can see, according to the Gini index decreasing the 3 most important variables are Sex(Male), Fare and Age. This differs (a little) from those variables important in the tree construction since Pclass was considered more important than Fare. We also observe that seems to be 4 “clusters” of variable (maybe 3 depending on interpretation):

1. Sex as the most important one, with a lot of difference.
2. Fare and Age as the next important ones.
3. Pclass which may be important or may not.
4. SibSp, Parch and Embarked as not important variables.

Now we proceed to train the random forest with a preselected values for the mtry parameter.

```
rfGrid = expand.grid(.mtry=c(1, 2, 3, 4, 5, 6, 7))
rfFit= train(x=trainData.predictors, y=trainData.target, method="rf", trControl=fi
tControl, ntree=2000, tuneGrid=rfGrid)
```

**Question 16:** Which are the combinations of parameters values tested by the train() function? Are there any changes in the performance of the algorithm when different combinations of values are used (according to the results of the cross validation)?

Using the same code as before.

```
rfFit$bestTune
```

```
##   mtry
## 3    3
```

```
rfFit$results[order(rfFit$results$Accuracy, decreasing=TRUE), ]
```

```
##   mtry  Accuracy     Kappa AccuracySD    KappaSD
## 3    3 0.8091652 0.5987886 0.04892883 0.10305351
## 2    2 0.8056866 0.5885964 0.05083094 0.10686098
## 1    1 0.8056261 0.5799872 0.04551125 0.09656851
## 6    6 0.7986691 0.5785212 0.04190398 0.08676659
## 4    4 0.7968845 0.5760324 0.04522916 0.09261886
## 5    5 0.7968845 0.5762629 0.04815875 0.09688063
## 7    7 0.7934362 0.5689953 0.03966491 0.08210739
```

**Question 17:** Which is the final combination of parameters values used?

The chosen value for the mtry parameter is

```
rfFit$bestTune
```

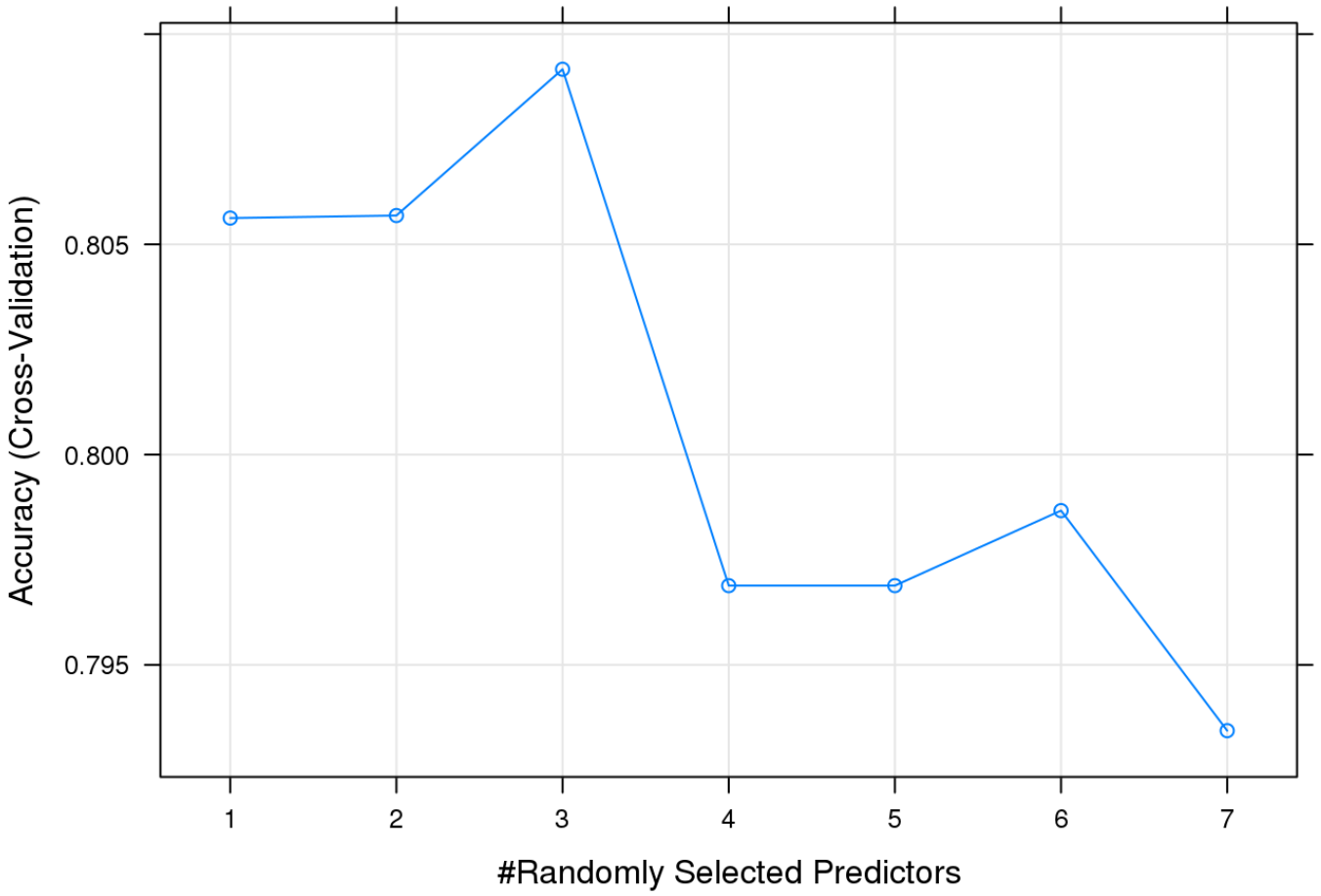
```
##   mtry
## 3    3
```

```
bestRf = rfFit$finalModel
```

Note that here there is no combination, since only one parameter is being tuned.

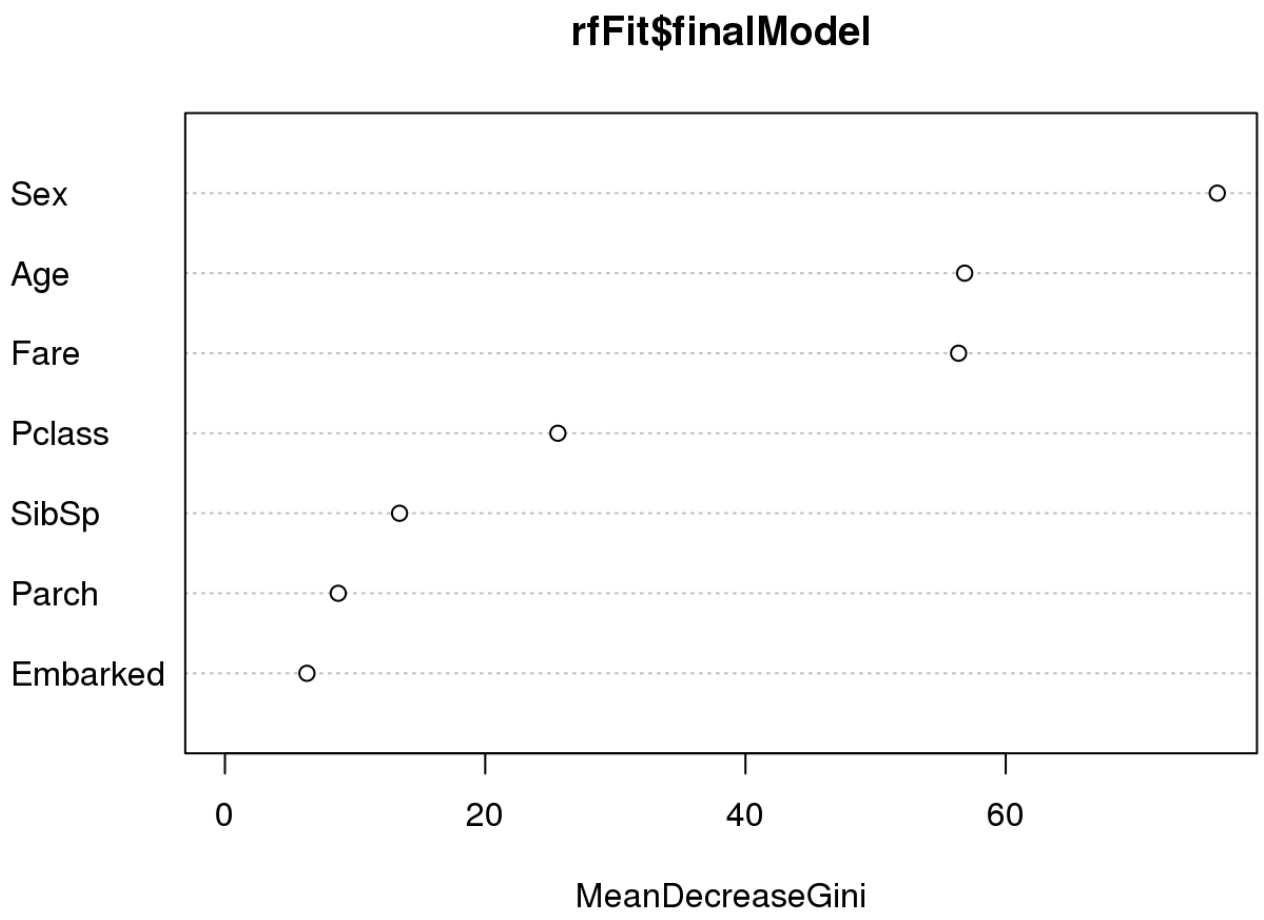
**Question 18:** Plot the result of calling the train() function with the plot() function. What does this plot represent?

```
plot(rfFit)
```




**Question 19:** Plot the importance of each variable for the model with function VarImPlot() from package caret. Which are the most relevant variables according to their mean decrease of the Gini index? Are these variables the ones selected when we built our decision trees?

```
varImpPlot(rfFit$finalModel)
```




```
importance(rfFit$finalModel)[order(importance(rfFit$finalModel), decreasing=TRU
E),]
```

```
##       Sex       Age      Fare    Pclass     SibSp     Parch  Embarked
## 76.256133 56.844621 56.376474 25.586733 13.417991  8.707503  6.303460
```

As before, the more relevant variables are Sex, Fare and Age, but what we find now is that the variable clusters are a bit clearer and that distance between important variables and non-important has increased.

## **Question 20:** Which is the difference in performance with regards to the testing subset between the best decision tree model and the best random forest model?

The results for the best Decision Tree found with automatic parameter fitting:

```
confusionMatrix(predict(bestTree, newdata=testData, type="class"), testData.targe
t)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 74 19
##          1 10 38
##
##                Accuracy : 0.7943
##                  95% CI : (0.7182, 0.8577)
##     No Information Rate : 0.5957
##     P-Value [Acc > NIR] : 4.327e-07
##
##                   Kappa : 0.5619
##  Mcnemar's Test P-Value : 0.1374
##
##             Sensitivity : 0.8810
##             Specificity : 0.6667
##          Pos Pred Value : 0.7957
##          Neg Pred Value : 0.7917
##              Prevalence : 0.5957
##          Detection Rate : 0.5248
##    Detection Prevalence : 0.6596
##       Balanced Accuracy : 0.7738
##
##        'Positive' Class : 0
##
```

Although the parameters have been automatically fitted, the default decision tree achieved better results than this one.

```
confusionMatrix(predict(defaultTree, newdata=testData, type="class"), testData.tar
get)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 77 20
##          1  7 37
##
##                Accuracy : 0.8085
##                  95% CI : (0.7338, 0.8699)
##     No Information Rate : 0.5957
##     P-Value [Acc > NIR] : 5.647e-08
##
##                   Kappa : 0.5873
##  Mcnemar's Test P-Value : 0.02092
##
##             Sensitivity : 0.9167
##             Specificity : 0.6491
##          Pos Pred Value : 0.7938
##          Neg Pred Value : 0.8409
##              Prevalence : 0.5957
##          Detection Rate : 0.5461
##    Detection Prevalence : 0.6879
##       Balanced Accuracy : 0.7829
##
##        'Positive' Class : 0
##
```

The results for the best Random Forest:

```
confusionMatrix(predict(bestRf, newdata=testData, type="class"), testData.target)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 75 17
##          1  9 40
##
##                Accuracy : 0.8156
##                  95% CI : (0.7416, 0.8759)
##     No Information Rate : 0.5957
##     P-Value [Acc > NIR] : 1.91e-08
##
##                   Kappa : 0.6083
##  Mcnemar's Test P-Value : 0.1698
##
##             Sensitivity : 0.8929
##             Specificity : 0.7018
##          Pos Pred Value : 0.8152
##          Neg Pred Value : 0.8163
##              Prevalence : 0.5957
##          Detection Rate : 0.5319
##    Detection Prevalence : 0.6525
##       Balanced Accuracy : 0.7973
##
##        'Positive' Class : 0
##
```

We conclude that we have improved performance with respect to the decision trees. Also a curious observation is that almost all of the created models are more prone to false-negatives than fals-potives so this might be a property of the studied probl

# Conclusions

As it was expected, the **random forests** model (with automatically tuned parameters) achieves **better generalization** results (better performance in test dataset) since what it is doing is “averaging” (actually bagging) the predictions made by 2000 independent trees. This better generalization comes from the **reduction in variance**, since the decision trees are weak learners (more prone to overfitting, that suffer from high variance) and the ensembles approach is based on reducing this variance while maintaining the bias, so that the generalization error also drops.

As a **drawback**, it must be mentioned that random forests **training time is much higher** than that for decision trees (obviously this depends on the number of trained trees, but for 2000 it is noticeable). This can be **solved by parallelizing** the tree’s training, so, when this is an option, random forests are a really good modelling option for classification tasks.