Decision Tree in R | Classification Tree & Code in R with Example

By Daniel Johnson Updated August 28, 2021

What are Decision Trees?

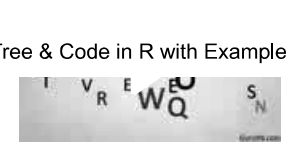
Decision Trees are versatile Machine Learning algorithm that can perform both classification and regression tasks. They are very powerful algorithms, capable of fitting complex datasets. Besides, decision trees are fundamental components of random forests, which are among the most potent Machine Learning algorithms available today.

Training and Visualizing a decision trees

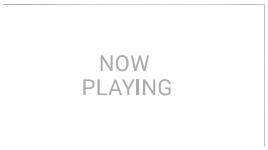
To build your first decision tree in R example, we will proceed as follow in this Decision Tree tutorial:

Step 1) Import the data

If you are curious about the fate of the
purpose of this dataset is to predict wl
collision with the iceberg. The dataset
dataset is ordered by the variable X.



What is Linux Linux Beginner Tutorial



10 Most Common Interview Questions and Answers ????

```
set.seed(678)
path <- 'https://raw.githubusercontent.com/guru99-edu/R-
Programming/master/titanic_data.csv'
titanic <-read.csv(path)
head(titanic)
```

Output:

```
##   X pclass survived
name    sex
## 1 1      1        1                    Allen, Miss. Elisabeth
Walton female
## 2 2      1        1                   Allison, Master. Hudson
Trevor   male
## 3 3      1        0                     Allison, Miss. Helen
Loraine female
## 4 4      1        0            Allison, Mr. Hudson Joshua
Creighton   male
## 5 5      1        0 Allison, Mrs. Hudson J C (Bessie Waldo
Daniels) female
## 6 6      1        1                              Anderson, Mr.
Harry   male
##       age sibsp parch tick
## 1 29.0000     0     0  241
## 2  0.9167     1     2 1137
## 3  2.0000     1     2 1137
## 4 30.0000     1     2 1137
## 5 25.0000     1     2 1137
## 6 48.0000     0     0  199
##                         ho
```

```
## 2 Montreal, PQ / Chesterville, ON
## 3 Montreal, PQ / Chesterville, ON
## 4 Montreal, PQ / Chesterville, ON
## 5 Montreal, PQ / Chesterville, ON
## 6                   New York, NY
```

```
tail(titanic)
```

Output:

```
##         X pclass survived                      name    sex  age
sibsp
## 1304 1304      3        0     Yousseff, Mr. Gerious   male   NA
0
## 1305 1305      3        0      Zabour, Miss. Hileni female 14.5
1
## 1306 1306      3        0     Zabour, Miss. Thamine female   NA
1
## 1307 1307      3        0 Zakarian, Mr. Mapriededer   male 26.5
0
## 1308 1308      3        0       Zakarian, Mr. Ortin   male 27.0
0
## 1309 1309      3        0        Zimmerman, Mr. Leo   male 29.0
0
##      parch ticket    fare
## 1304     0   2627 14.4583
## 1305     0   2665 14.4542
## 1306     0   2665 14.4542
## 1307     0   2656  7.2250
## 1308     0   2670  7.2250
## 1309     0 315082  7.8750
```

From the head and tail output, you can notice the data is not shuffled. This is a big issue! When you will split your data between a train set and test set, you will select **only** the passenger from class 1 and 2 (No passenger from class 3 are in the top 80 percent of the observations), which means the algorithm will never see the features of passenger of class 3. This mistake will lead to poor prediction.

To overcome this issue, you can use the function sample().

```
shuffle_index <- sample(1:nrow(titanic))
head(shuffle_index)
```

Decision tree R code Explanation

- sample(1:nrow(titanic)): Generate a random list of index from 1 to 1309 (i.e. the maximum number of rows).

Output:

```
## [1]  288  874 1078  633  8
```

You will use this index to shuffle the tit

Output:

```
##         X pclass survived
## 288   288      1        0
## 874   874      3        0
## 1078 1078      3        1
## 633   633      3        0
## 887   887      3        1
## 992   992      3        1
##                                                      name
sex age
## 288                                    Sutton, Mr. Frederick
male  61
## 874                  Humblen, Mr. Adolf Mathias Nicolai Olsen
male  42
## 1078                               O'Driscoll, Miss. Bridget
female  NA
## 633  Andersson, Mrs. Anders Johan (Alfrida Konstantia Brogren)
female  39
## 887                                      Jermyn, Miss. Annie
female  NA
## 992                                         Mamee, Mr. Hanna
male  NA
##      sibsp parch ticket    fare cabin embarked
home.dest## 288      0     0  36963 32.3208   D50        S
Haddenfield, NJ
## 874      0     0 348121  7.6500 F G63        S
## 1078     0     0  14311  7.7500              Q
## 633      1     5 347082 31.2750              S Sweden Winnipeg, MN
## 887      0     0  14313  7.7500              Q
## 992      0     0   2677  7.2292              C
```

Step 2) Clean the dataset

The structure of the data shows some
follows

- Create factor variables for pclass and survived
- Drop the NA

```
library(dplyr)
# Drop variables
clean_titanic <- titanic % > %
select(-c(home.dest, cabin, name, X, ticket)) % > %
#Convert to factor level
        mutate(pclass = factor(pclass, levels = c(1, 2, 3), labels =
c('Upper', 'Middle', 'Lower')),
        survived = factor(survived, levels = c(0, 1), labels =
c('No', 'Yes'))) % > %
na.omit()
glimpse(clean_titanic)
```

Code Explanation

- select(-c(home.dest, cabin, name, X, ticket)): Drop unnecessary variables
- pclass = factor(pclass, levels = c(1,2,3), labels= c('Upper', 'Middle', 'Lower')): Add label to the variable pclass. 1 becomes Upper, 2 becomes MIddle and 3 becomes lower
- factor(survived, levels = c(0,1), labels = c('No', 'Yes')): Add label to the variable survived. 1 Becomes No and 2 becomes Yes
- na.omit(): Remove the NA observations

Output:

```
## Observations: 1,045
## Variables: 8
## $ pclass   <fctr> Upper, Lower, Lower, Upper, Middle, Upper,
Middle, U...
## $ survived <fctr> No, No,
No, Y...
## $ sex      <fctr> male, ma
male...
## $ age      <dbl> 61.0, 42.
2.0, ...
## $ sibsp    <int> 0, 0, 1,
```

```
4, 0,...
## $ fare     <dbl> 32.3208, 7.6500, 31.2750, 25.9292, 10.5000,
52.5542, ...
## $ embarked <fctr> S, S, S, S, S, S, S, S, S, C, S, S, S, Q, C, S,
S, C...
```

Step 3) Create train/test set

Before you train your model, you need to perform two steps:

- Create a train and test set: You train the model on the train set and test the prediction on the test set (i.e. unseen data)
- Install rpart.plot from the console

The common practice is to split the data 80/20, 80 percent of the data serves to train the model, and 20 percent to make predictions. You need to create two separate data frames. You don't want to touch the test set until you finish building your model. You can create a function name create_train_test() that takes three arguments.

```
create_train_test(df, size = 0.8, train = TRUE)
arguments:
-df: Dataset used to train the model.
-size: Size of the split. By default, 0.8. Numerical value
-train: If set to `TRUE`, the function creates the train set,
otherwise the test set. Default value sets to `TRUE`. Boolean
value.You need to add a Boolean parameter because R does not allow to
return two data frames simultaneously.
```

```
create_train_test <- function
    n_row = nrow(data)
    total_row = size * n_row
    train_sample < - 1: total
    if (train == TRUE) {
        return (data[train_sa
    } else {
        return (data[-train_s
```

Code Explanation

- function(data, size=0.8, train = TRUE): Add the arguments in the function
- n_row = nrow(data): Count number of rows in the dataset
- total_row = size*n_row: Return the nth row to construct the train set
- train_sample <- 1:total_row: Select the first row to the nth rows
- if (train ==TRUE){ } else { }: If condition sets to true, return the train set, else the test set.

You can test your function and check the dimension.

```
data_train <- create_train_test(clean_titanic, 0.8, train = TRUE)
data_test <- create_train_test(clean_titanic, 0.8, train = FALSE)
dim(data_train)
```

Output:

```
## [1] 836   8
```

```
dim(data_test)
```

Output:

```
## [1] 209   8
```

The train dataset has 1046 rows while

You use the function prop.table() coml
process is correct.

```
prop.table(table(data_train$survived))
```

Output:

```
##
##        No       Yes
## 0.5944976 0.4055024
```

```
prop.table(table(data_test$survived))
```

Output:

```
##
##        No       Yes
## 0.5789474 0.4210526
```

In both dataset, the amount of survivors is the same, about 40 percent.

Install rpart.plot

rpart.plot is not available from conda libraries. You can install it from the console:

```
install.packages("rpart.plot")
```

Step 4) Build the model

You are ready to build the model. The

```
rpart(formula, data=, method=
arguments:
```

```
- data: Specifies the data frame- method:
- "class" for a classification tree
- "anova" for a regression tree
```

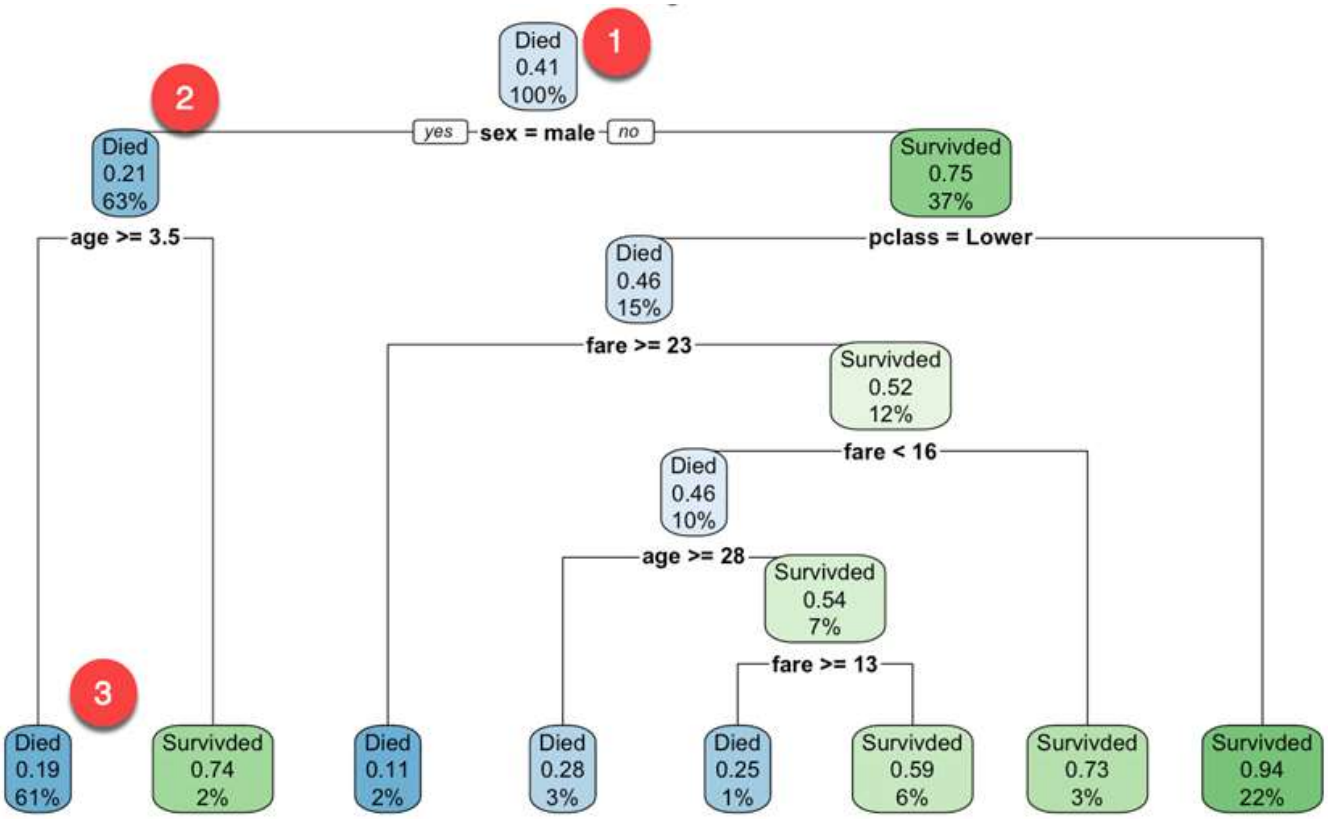
You use the class method because you predict a class.

```
library(rpart)
library(rpart.plot)
fit <- rpart(survived~., data = data_train, method = 'class')
rpart.plot(fit, extra = 106
```

Code Explanation

- rpart(): Function to fit the model. The arguments are:
    - survived ~.: Formula of the Decision Trees
    - data = data_train: Dataset
    - method = 'class': Fit a binary model
- rpart.plot(fit, extra= 106): Plot the tree. The extra features are set to 101 to display the probability of the 2nd class (useful for binary responses). You can refer to the vignette for more information about the other choices.

Output:



You start at the root node (depth 0 over 3, the top of the graph):

1. At the top, it is the overall probability of survival. It shows the proportion of passenger that survived the crash. 41 percent of passenger survived.
2. This node asks whether the gender of the passenger is male. If yes, then you go down to the root's left child node (depth 2). 63 percent are males with a survival probability of 21 percent.
3. In the second node, you ask if the male passenger is above 3.5 years old. If yes, then the chance of survival is 19 percent.
4. You keep on going like that to understand what features impact the likelihood of survival.

Note that, one of the many qualities of Decision Trees is that they require very little data preparation. In particular, they don't require feature scaling or centering.

By default, rpart() function uses the **Gi**
the Gini coefficient, the more different

Step 5) Make a prediction

You can predict your test dataset. To m

```
predict(fitted_model, df, type = 'class')
arguments:
- fitted_model: This is the object stored after model estimation.
- df: Data frame used to make the prediction
- type: Type of prediction
    - 'class': for classification
    - 'prob': to compute the probability of each class
    - 'vector': Predict the mean response at the node level
```

You want to predict which passengers are more likely to survive after the collision from the test set. It means, you will know among those 209 passengers, which one will survive or not.

```
predict_unseen <-predict(fit, data_test, type = 'class')
```

Code Explanation

- predict(fit, data_test, type = 'class'): Predict the class (0/1) of the test set

Testing the passenger who didn't make it and those who did.

```
table_mat <- table(data_test$survived, predict_unseen)
table_mat
```

Code Explanation

- table(data_test$survived, predict
  passengers are classified as survi
  decision tree classification in R

Output:

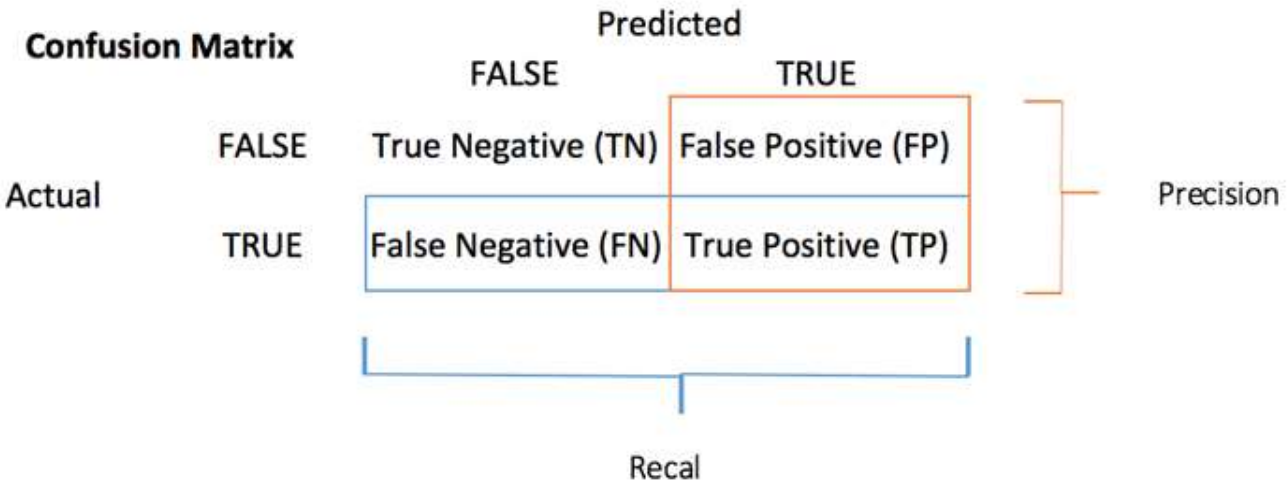
```
##    No  106  15
##    Yes  30  58
```

The model correctly predicted 106 dead passengers but classified 15 survivors as dead. By analogy, the model misclassified 30 passengers as survivors while they turned out to be dead.

Step 6) Measure performance

You can compute an accuracy measure for classification task with the **confusion matrix**:

The **confusion matrix** is a better choice to evaluate the classification performance. The general idea is to count the number of times True instances are classified are False.



Each row in a confusion matrix represents an actual target, while each column represents a predicted target. The first row of this matrix considers dead passengers (the False class): 106 were correctly classified as dead (**True negative**), while the remaining one was wrongly classified as a survivor (**False positive**). The second row considers the survivors, the positive class were 58 (**True positi**

You can compute the **accuracy test** fro

$$accuracy =$$

```
accuracy_Test <- sum(diag(table_mat)) / sum(table_mat)
```

Code Explanation

- sum(diag(table_mat)): Sum of the diagonal
- sum(table_mat): Sum of the matrix.

You can print the accuracy of the test set:

```
print(paste('Accuracy for test', accuracy_Test))
```

Output:

```
## [1] "Accuracy for test 0.784688995215311"
```

You have a score of 78 percent for the test set. You can replicate the same exercise with the training dataset.

Step 7) Tune the hyper-parameters

Decision tree in R has various parameters that control aspects of the fit. In rpart decision tree library, you can control the parameters using the rpart.control() function. In the following code, you introduce the parameters you will tune. You can refer to the vignette for other parameters.

```
rpart.control(minsplit = 20,
= 30)
Arguments:
-minsplit: Set the minimum nu
the algorithm perform a split
-minbucket:  Set the minimum
i.e. the leaf
-maxdepth: Set the maximum de
```

We will proceed as follow:

- Construct function to return accuracy
- Tune the maximum depth
- Tune the minimum number of sample a node must have before it can split
- Tune the minimum number of sample a leaf node must have

You can write a function to display the accuracy. You simply wrap the code you used before:

1. predict: predict_unseen <- predict(fit, data_test, type = 'class')
2. Produce table: table_mat <- table(data_test$survived, predict_unseen)
3. Compute accuracy: accuracy_Test <- sum(diag(table_mat))/sum(table_mat)

```
accuracy_tune <- function(fit) {
    predict_unseen <- predict(fit, data_test, type = 'class')
    table_mat <- table(data_test$survived, predict_unseen)
    accuracy_Test <- sum(diag(table_mat)) / sum(table_mat)
    accuracy_Test
}
```

You can try to tune the parameters and see if you can improve the model over the default value. As a reminder, you need to get an accuracy higher than 0.78

```
control <- rpart.control(minsplit = 4,
    minbucket = round(5 / 3),
    maxdepth = 3,
    cp = 0)
tune_fit <- rpart(survived~., data = data_train, method = 'class',
control = control)
accuracy_tune(tune_fit)
```


Output:

```
## [1] 0.7990431
```

With the following parameter:

```
minsplit = 4
minbucket= round(5/3)
maxdepth = 3cp=0
```

You get a higher performance than the previous model. Congratulation!

Summary

We can summarize the functions to train a decision tree algorithm in R

Library	Objective	Function	Class	Parameters	Details
rpart	Train classification tree in R	rpart()	class	formula, df, method	
rpart	Train regression tree	rpart()	anova	formula, df, method	
rpart	Plot the trees	rpart.plot()		fitted model	
base	predict	predict()	class	fitted model, type	
base	predict	predict()	prob	fitted model,	
base	predict	predict()			

Library	Objective	Function	Class	Parameters	Details
rpart	Control parameters	rpart.control()		minsplit	Set the minimum number of observations in the node before the algorithm perform a split
				minbucket	Set the minimum number of observations in the final note i.e. the leaf
				maxdepth	Set the maximum depth of any node of the final tree. The root node is treated a depth 0
rpart	Train model with control parameter	rpart()		formula, df, method, control	

Note : Train the model on a training data and test the performance on an unseen dataset, i.e. test set.

You Might Like:

- Code Coverage Tutorial: Branch, Statement, Decision, FSM
- 15 BEST Code Review Tools for C
- TensorFlow Binary Classification:
- CNN Image Classification in Tens
- PyTorch Tutorial: Regression, Ima