



DECISION TREES

E. Fersini



INTRODUCTION

On April 15, 1912, during the maiden voyage, the Titanic sank had a collision with an iceberg, killing 1502 out of 2224 passengers and crew.

During this lab, we will try to analysis what sorts of people were likely to survive. We will induce a decision tree on the dataset to predict which passengers survived the tragedy.

INTRODUCTION

- Dataset

Variable	Definition	NOTE
survival	Survival	0 = No, 1 = Yes
pclass	Ticket class	1 = 1st, 2 = 2nd, 3 = 3rd
sex	Sex	
Age	Age in years	
sibsp	# of siblings / spouses aboard the Titanic	
parch	# of parents / children aboard the Titanic	
ticket	Ticket number	
fare	Passenger fare	
cabin	Cabin number	
embarked	Port of Embarkation	C = Cherbourg, Q = Queenstown, S = Southampton

IMPORT THE DATASET

- We can use `read.csv()` to load data into a data frame.
 - In order to treat the **blank** string as **NA**, we can specify that `na.strings` equals either "NA" or an empty string

```
> dataset = read.csv("dataset.csv", na.strings=c("NA", ""))
```

- Check the loaded data with the `str()` function:

```
str(dataset)
```

IMPORT THE DATASET

'data.frame': 891 obs. of 12 variables:

\$ PassengerId: int 1 2 3 4 5 6 7 8 9 10 ...

\$ Survived : int 0 1 1 1 0 0 0 1 1 ...

This is our target variable

\$ Pclass : int 3 1 3 1 3 3 1 3 3 2 ...

\$ Name : Factor w/ 891 levels "Abbing, Mr. Anthony",...: 109 191 358 277 16 559 520 629 417 581 ...

\$ Sex : Factor w/ 2 levels "female","male": 2 1 1 1 2 2 2 1 1 ...

\$ Age : num 22 38 26 35 35 NA 54 2 27 14 ...

\$ SibSp : int 1 1 0 1 0 0 0 3 0 1 ...

\$ Parch : int 0 0 0 0 0 0 0 1 2 0 ...

\$ Ticket : Factor w/ 681 levels "110152","110413",...: 524 597 670 50 473 276 86 396 345 133 ...

\$ Fare : num 7.25 71.28 7.92 53.1 8.05 ...

\$ Cabin : Factor w/ 147 levels "A10","A14","A16",...: NA 82 NA 56 NA NA 130 NA NA NA ...

\$ Embarked : Factor w/ 3 levels "C","Q","S": 3 1 3 3 3 2 3 3 3 1 ...

DATA TRANSFORMATION

- Since nominal, ordinal, interval, and ratio variable are treated differently, we have to convert a nominal variable from a character into a factor.
- To transform the variable from the **int** numeric type to the **factor** categorical type, you can cast factor:

```
> dataset$Survived = factor(dataset$Survived)
```


```
> dataset$Pclass = factor(dataset$Pclass)
```

- Print out the variable with the str() function and again, you can see that **Pclass** and **Survived** are now transformed into the **factor**

```
str(dataset)
```

DATASET EXPLORATION

- Bar plot of passenger survival
- Bar plot of gender
- Bar plot of ages
- Use barplot to:
 - identify which gender (`$Sex`) is more likely to perish during shipwrecks
 - examine whether the class (`$Pclass`) factor of each passenger may affect the survival rate



```
> barplot(table(dataset$Survived), main="Passenger Survival",  
names= c("Perished", "Survived"))
```

```
> barplot(table(dataset$Sex), main="Passenger Gender")
```

```
> hist(dataset$Age, main="Passenger Age", xlab = "Age")
```

```
> counts = table( dataset$Survived, dataset$Sex)
```

```
> barplot(counts, col=c("darkblue","red"), legend = c("Perished",  
"Survived"), main = "Passenger Survival by Sex")
```


TRAIN AND TEST

The dataset can be split into two groups:

- TRAINING SET:
 - The **training set** should be used to **build** our **machine learning models**. For the training set, we provide the outcome (also known as the “ground truth”) for each passenger, i.e. the prediction model will be based on “features” like passengers’ gender and class.
- TEST SET
 - The **test set** should be used to see **how well your model performs on unseen data**. For the test set, there is no ground truth. For each passenger in the test set, use the model we trained to predict whether or not they survived the sinking of the Titanic.
 - However, in our case we will have a ground truth to understand if our model performs well. But this is just for evaluation purposes!!

TRAIN AND TEST

1. Let's create a function called `split.data()` to split our dataset in **train** and **test**. We should have 3 **input parameters**:
 - **data**: input dataset.
 - **p**: proportion of generated subset from the input dataset.
 - **s**: random seed.

```
> split.data = function(data, p = 0.7, s = 1){  
  set.seed(s)  
  index = sample(1:dim(data)[1])  
  train = data[index[1:floor(dim(data)[1] * p)], ]  
  test = data[index[(ceiling(dim(data)[1] * p)) + 1]:dim(data)[1]], ]  
  return(list(train=train, test=test)) }
```

TRAIN AND TEST

- Now we can use our function to effectively split the dataset in train and test:

```
> allset= split.data(dataset, p = 0.7)
```

```
> trainset= allset$train
```

```
> testset= allset$test
```

DATASET EXPLORATION

- We can use the `table()` function on the attribute `trainset$Survived` to count the occurrence of each value in it.
 - We see that in the training set, 238 passengers survived, while 385 died.

```
> table(trainset$Survived)
```

- If you want to see a distribution you can use the `prop.table()` function

```
> prop.table(table(trainset$Survived))
```

- 38% of passengers survived the disaster in the training set.

BASELINE MODEL

- We are you ready to make our first “dummy” prediction, which is usually called **baseline model**:
 - Since most people died in our training set, perhaps it's a good start to assume that everyone in the test set did too.
- Let's add a trivial “everyone dies” prediction to the test set
 - Since there was no “Prediction” column in the dataframe, it will create one for us and repeat our “0” prediction 267times, the number of test instances we have. Recall that we work with factors!

```
> testset$Prediction = rep(0, 267)
> testset$Prediction = factor(testset$Prediction)
```

CONFUSION MATRIX

- We can now estimate how good is our baseline model, i.e. we can compare if our prediction is correct with respect to the ground truth
 1. Create a **confusion matrix** in a very naive way

```
> confusion.matrix = table(testset$Survived, testset$Prediction)
```

		Predicted class	
		Yes	No
Actual class	Yes	TP: True positive	FN: False negative
	No	FP: False positive	TN: True negative

ACCURACY

- We can now estimate how good is our baseline model, i.e. we can compare if our prediction is correct with respect to the ground truth
 2. Estimate **accuracy**

```
> sum(diag(confusion.matrix))/sum(confusion.matrix)
```

0.6142322



Not so bad considering the “dummy” prediction!



LET'S MAKE A BETTER BASELINE

- According to the plot that we have created at the beginning of the lesson, we know which input variable can better explain our target.
 - Let's use this information to create a better baseline

LET'S MAKE A BETTER BASELINE

- We observed during our exploration task that there is one variable that describe well the survival

```
> prop.table(table(trainset$Sex, trainset$Survived),1)
```

	0	1
female	0.2669683	0.7330317
male	0.8109453	0.1890547

- The majority of females aboard survived, and a very low percentage of males did. We can update our baseline model prediction accordingly:

LET'S MAKE A BETTER BASELINE

- Let's update our prediction on the test set

```
> testset$Prediction = 0  
> testset$Prediction[testset$Sex == 'female'] = 1  
> testset$Prediction = factor(testset$Prediction)
```

- Let's estimate Accuracy on the “smarter” baseline

```
> confusion.matrix = table(testset$Survived, testset$Prediction)  
> sum(diag(confusion.matrix))/sum(confusion.matrix)
```

0.7977528

TRAIN A DECISION TREE

- Load the rpart package for creating a decision tree

```
> library(rpart)
```

- Use the rpart() function to **build** a classification tree model:

```
> decisionTree = rpart(Survived ~ ., data=trainset, method="class")
```

- Remarks:**
- symbol `~` stands for “is modeled as”
 - We have explicitly directed the routine to treat Survived as a categorical variable by using **method='class'**
 - `*` indicates that the node is terminal.

TYPES OF DECISION TREES

rpart

- Type of learning:
 - ID3, as an "Iterative Dichotomiser," is for binary classification only
 - **CART**, or "Classification And Regression Trees," is a family of algorithms (including, but not limited to, binary classification tree learning).
 - With `rpart()`, you can specify `method='class'` or `method='anova'`, but `rpart` can infer this from the type of dependent variable (i.e., factor or numeric).
- Loss functions used for split selection.
 - ID3, selects its splits based on Information Gain, which is the reduction in entropy between the parent node and (weighted sum of) children nodes.
 - **CART**, selects its splits to achieve the subsets that minimize Gini Index

$$I_G(p) = 1 - \sum_{i=1}^J p_i^2$$

$p_{i|s}$ is the fraction of items labeled with class i in the set.

TRAIN A DECISION TREE

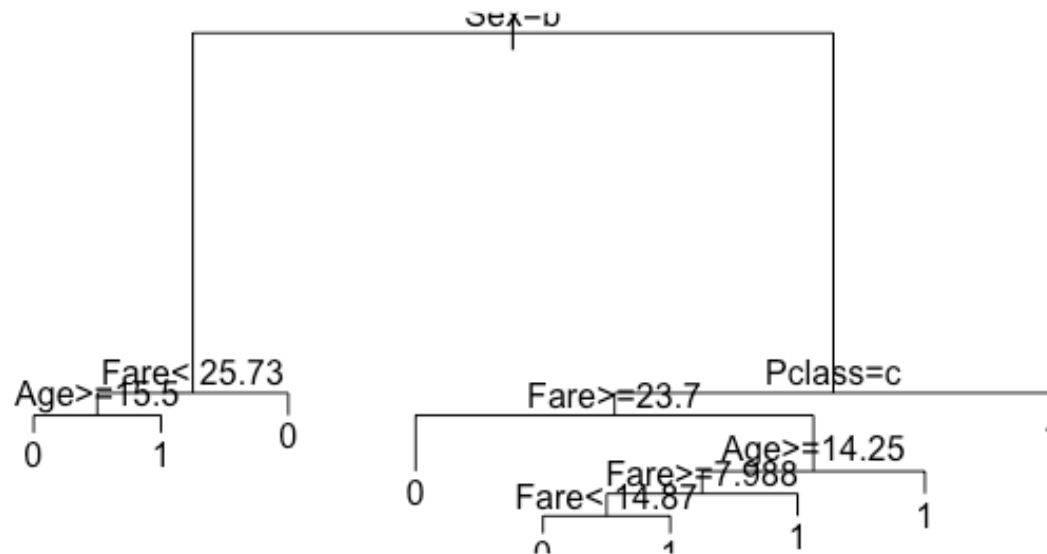
- If we do not specify which attribute should be used to induce our decision tree, the result will be useless
 - Also text variables will be considered
- We should use only a sub set of input features!

```
> decisionTree = rpart(Survived ~ Pclass + Sex + Age + SibSp + Parch  
+ Fare + Embarked, data=trainset, method="class")
```

VISUALIZE A DECISION TREE

- Very naive visualization

```
> plot(decisionTree)  
> text(decisionTree)
```

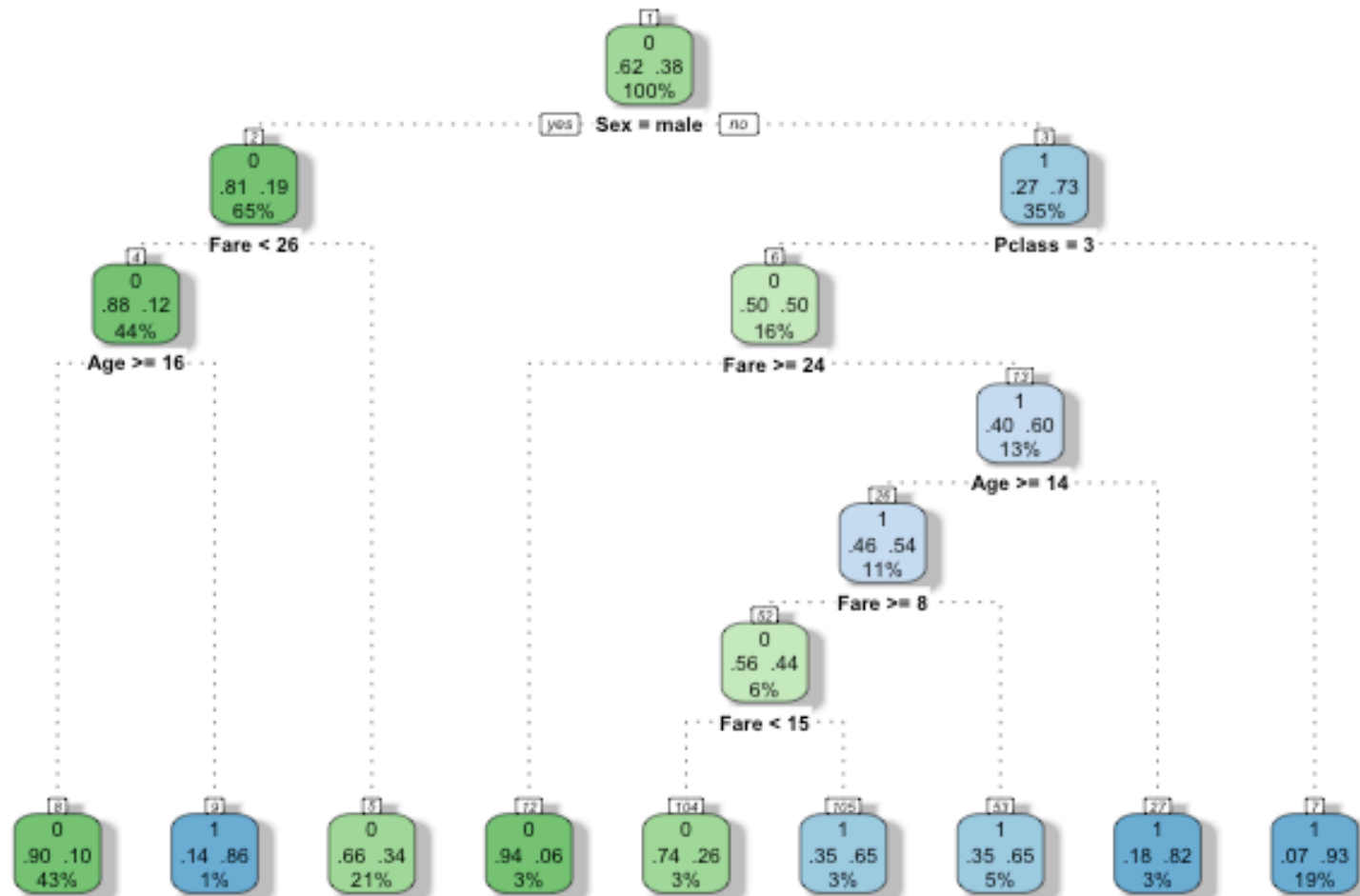


VISUALIZE A DECISION TREE

- Better visualization

```
> library(rattle)
> library(rpart.plot)
> library(RColorBrewer)
> fancyRpartPlot(decisionTree)
```

VISUALIZE A DECISION TREE



INFERENCE

- We can now use our decision tree model, already trained, to make a prediction on our test set

```
> testset$Prediction <- predict(decisionTree, testset, type = "class")
```

- Let's estimate Accuracy on our decision tree model (on the test set)

```
> confusion.matrix = table(testset$Survived, testset$Prediction)  
> sum(diag(confusion.matrix))/sum(confusion.matrix)
```

0.835206

OTHER USEFUL INFORMATION

- The decision tree that we've built contains a rich set of information

```
> summary(decisionTree)
```

- List complexity parameter

```
> printcp(decisionTree)
```

- Plot complexity parameter

```
> plotcp(decisionTree)
```

COMPLEXITY PARAMETER

- The **complexity parameter** (cp) is the minimum improvement in the model needed at each node. It's based on the cost complexity of the model defined as

$$\sum_{\text{Terminal Nodes}} \text{Misclass}_i + \lambda * (\text{Splits})$$

- For the given tree, add up the misclassification at every terminal node.
- Then multiply the number of splits time a penalty term (lambda) and add it to the total misclassification.
 - The lambda is determined through cross-validation and not reported in R.
 - λ can take value between 0 and ∞ and defines the 'cost' of adding another variable to the model
 - The cp we see using `printcp()` is the scaled version of lambda over the misclassification rate of the overall data.
 - The cp value is a stopping parameter. It helps speed up the search for splits because it can identify splits that don't meet this criteria and prune them before going too far.

OTHER USEFUL INFORMATION

- With the `printcp()` function we can understand better our decision tree model.

```
> printcp(decisionTree)
```

- We can find the value of a **complexity parameter** that serves as a penalty to control the size of the tree.
 - In short, the greater the CP value, the fewer the number of splits there are.
 - *rel error* represents the average deviance of the current tree divided by the average deviance of the null tree.
 - *xerror* value represents the relative error estimated on the training set.
 - *xstd* stands for the standard error of the relative error.

OTHER USEFUL INFORMATION

- In practice:

Classification tree:

```
rpart(formula = Survived ~ Pclass + Sex + Age + SibSp + Parch +  
      Fare + Embarked, data = trainset, method = "class")
```

Variables actually used in tree construction:

```
[1] Age    Fare    Pclass Sex
```

Root node error: 238/623 = 0.38202

n= 623

	CP	nsplit	rel error	xerror	xstd
1	0.432773	0	1.00000	1.00000	0.050956
2	0.033613	1	0.56723	0.56723	0.043207
3	0.012605	3	0.50000	0.51261	0.041618
4	0.010504	6	0.46218	0.57983	0.043550
5	0.010000	8	0.44118	0.57983	0.043550

- CP is the amount by which splitting that node improved the relative error. So in our example, splitting the original root node dropped the relative error from 1.0 to 0.43
 - so the CP of the root node is 0.57

PRUNING DECISION TREE

- In order to avoid overfitting, it is better to prune our trees

```
> prunedDecisionTree = prune(decisionTree, cp= decisionTree.cp)
```

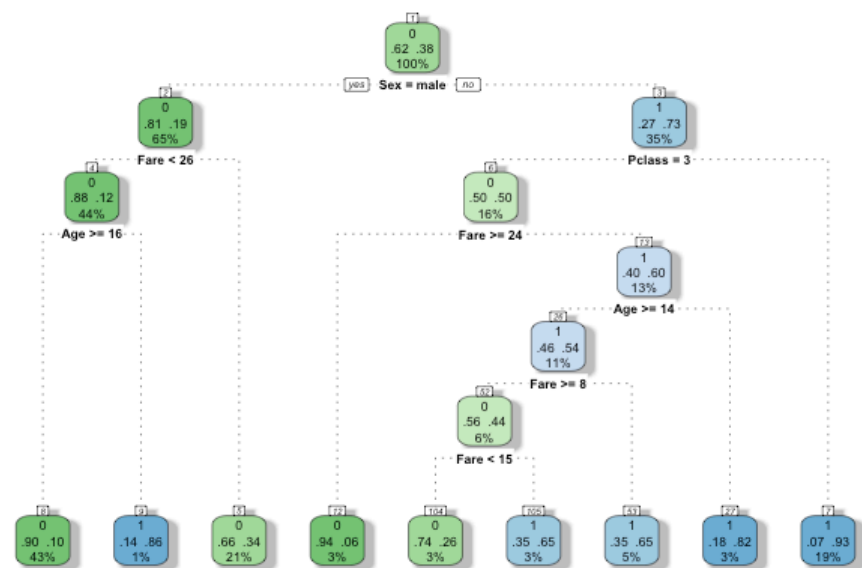
```
> fancyRpartPlot(prunedDecisionTree)
```

- If we want, we can specify at which CP level we should prune our tree

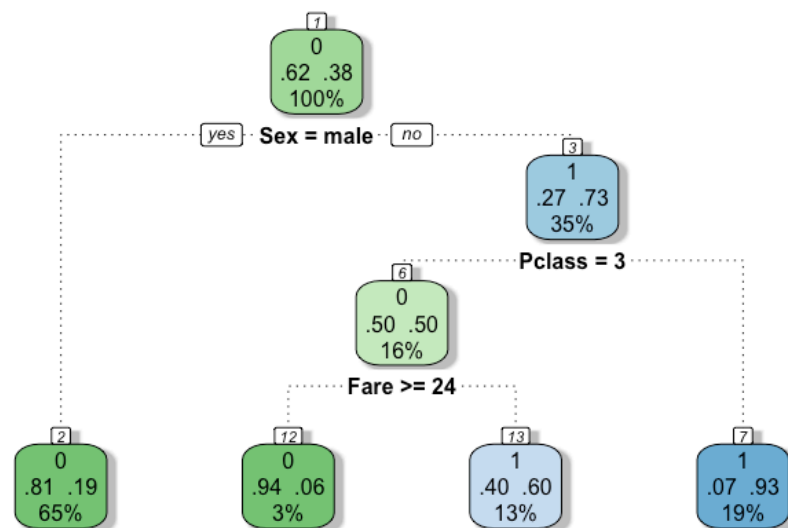
```
> myPruned = prune(decisionTree, cp=.011)
```

PRUNING DECISION TREE

Original Tree



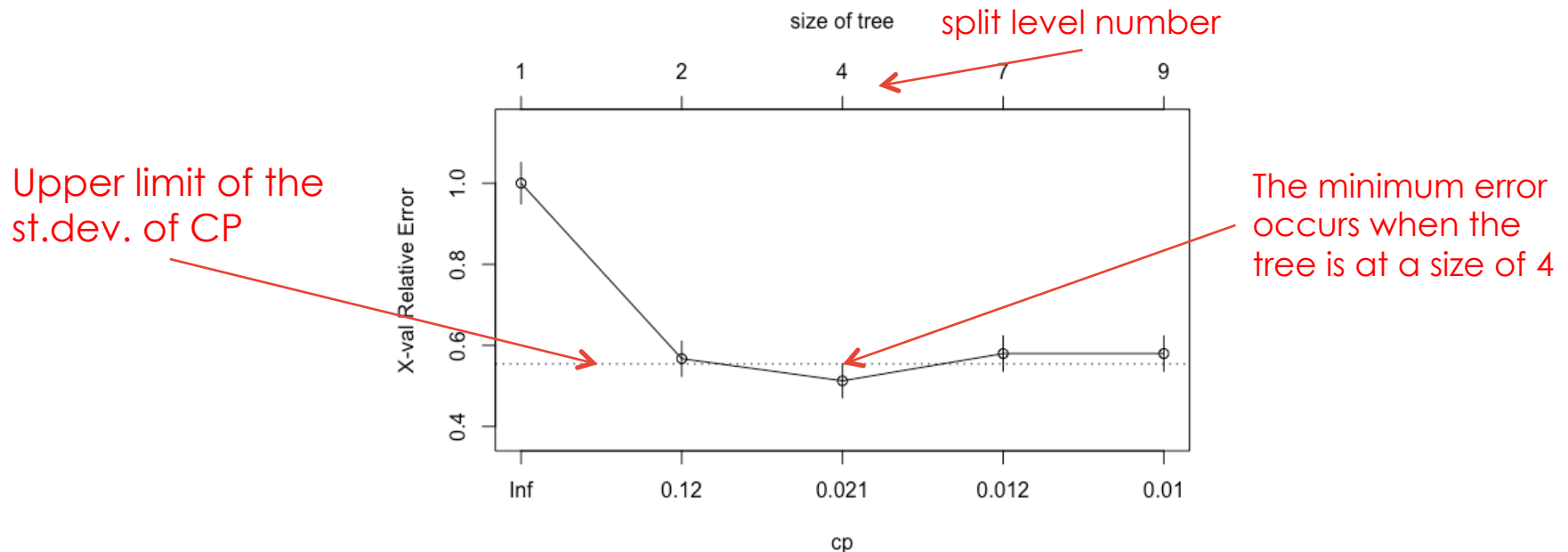
Pruning at CP=0.11



OTHER USEFUL INFORMATION

- With the `plotcp()` function we can visualize the complexity parameter of the decision tree.

```
> plotcp(decisionTree)
```



JUST TO KNOW

- `printcp()` and `plotcp()` are slightly different:
 - `printcp()` gives the minimal `cp` for which the pruning happens.
 - `plotcp()` plots against the geometric mean
- For more information about the `rpart`, `printcp`, and `summary` functions, please use the help function:
 - > **?rpart**
 - > **?printcp**
 - > **?summary.rpart**

ASSIGNMENT

- Now let's use a different splitting criteria
- You can specify the splitting criteria
 - The default is the Gini Index
 - You can try with the "Information gain"
- **Do you obtain the same tree?**

```
> decisionTreeIG = rpart(Survived ~ Pclass + Sex + Age + SibSp +  
Parch + Fare + Embarked, data=trainset, method="class", parms =  
list(split = 'information'))
```