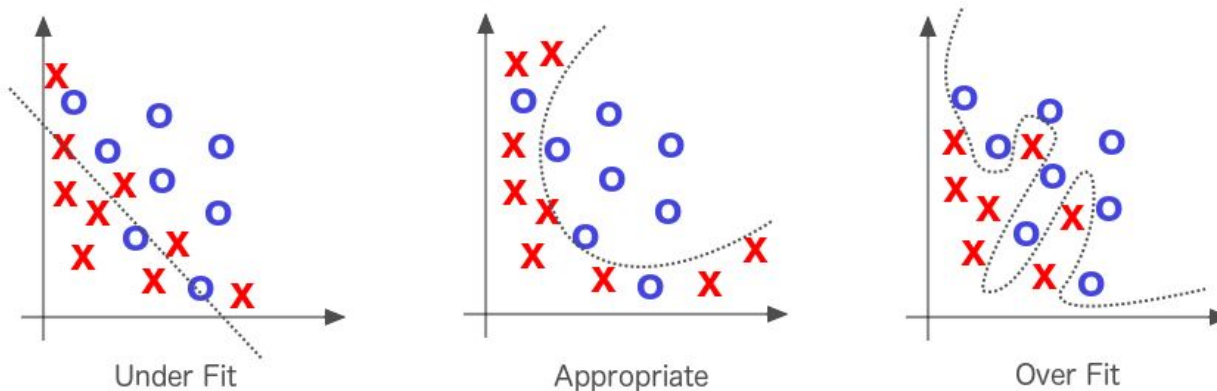


Overfitting

(what you just did)

Learning irrelevant details of the dataset: Overfitting

We say that a model is overfitting if its predictive abilities (output) depend too much on the data which was used for learning the parameters.

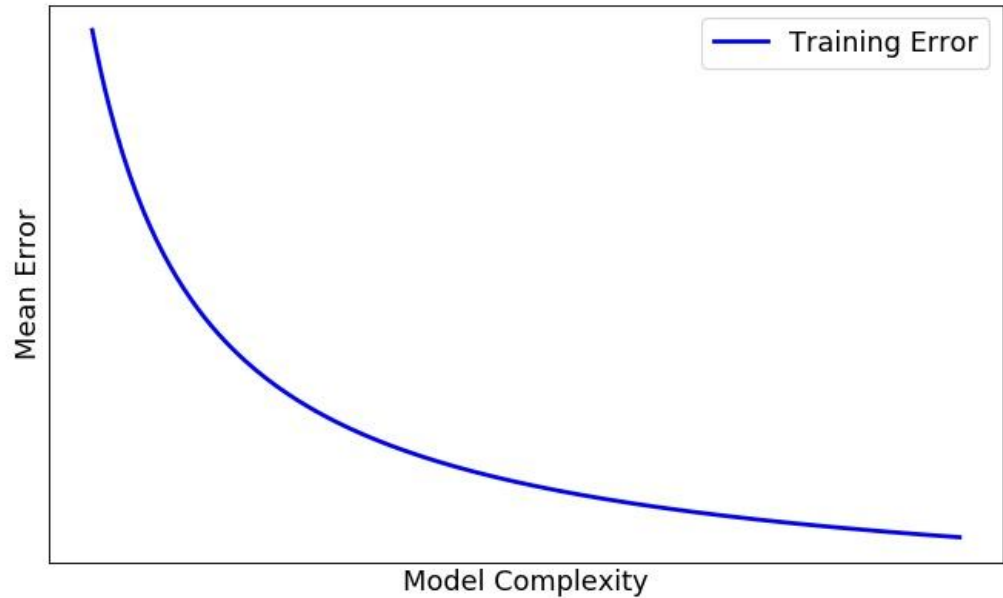


Think about adding one new point to the graph, how well will the model perform?

The more complex the model is, the more likely it is to overfit. Overfitting is a huge issue as the model will seem to perform great when training it and then perform poorly when applied.

Overfitting: training error

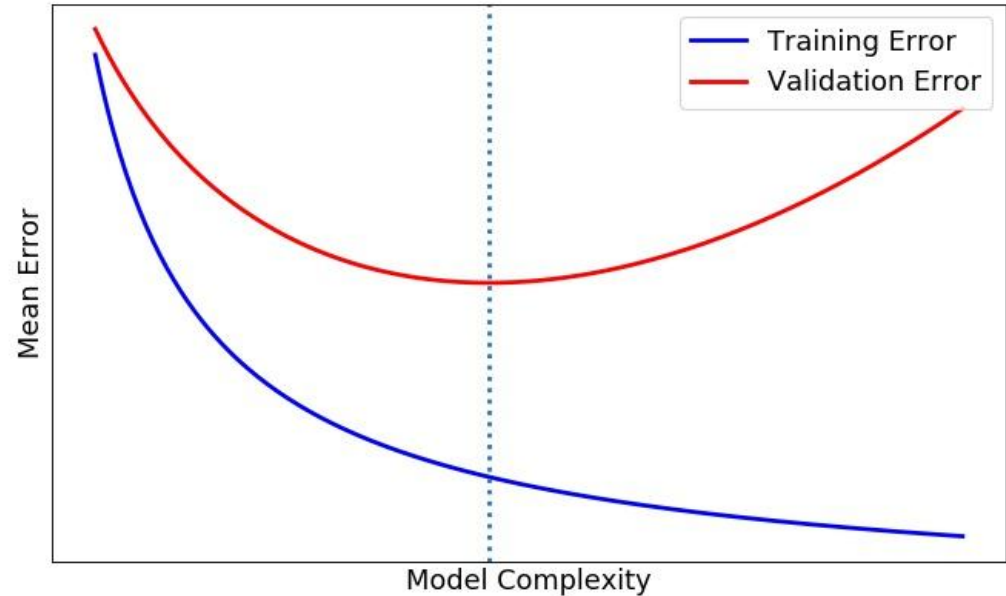
As you have just experienced, **making the model more complex** (e.g. adding more variables, interaction terms etc), **decreases the “mean error” of the model** [blue line, training error].



Overfitting: test error

However, as we have seen in the competition, a low mean error might mean that our model overfits the data: it might start learning irrelevant details of the data and the mean error goes towards 0 [blue line].

This means that, while the model will seem great on the data we have, **it will perform poorly on new data we have not seen before** [red line, validation error]. This dataset is called the validation set.



As we increase model complexity:

↑
Error on data we have seen (training) and data we have not seen (validation) decreases. Making the model more complex gives us a better model.

↑
At some point the model starts to overfit. The error on data we have seen (training) continues to decrease but error on unseen data (validation) starts increasing

Preventing overfit: Train - Validation

To prevent overfitting and make sure we have a good estimate how our classifier will perform, we divide our dataset (at random) into two sets:

Training set: Part of the dataset that we use to **train the model**. For example to learn the coefficients of our logistic regression.

Validation set: Part of the dataset that we use to decide on the type of model and depth of the tree

Creating train and validation sets

```
# First, we split the data into train, test and validation set  
set.seed(100) # set seed for random sample splitting  
  
# 80% of observations will go into the training set  
split <- 0.8
```

Creating train and validation sets

```
# First, we split the data into train, test and validation set
set.seed(100) # set seed for random sample splitting

# 80% of observations will go into the training set
split <- 0.8

# size of train set
train_size <- floor(splitsplit * nrow(titanic))
```

Creating train and validation sets

```
# First, we split the data into train, test and validation set
set.seed(100) # set seed for random sample splitting

# 80% of observations will go into the training set
split <- 0.8

# size of train set
train_size <- floor(first_split * nrow(titanic))

# id to bootstrap train set
train_id <- sample(seq_len(nrow(titanic)), train_size, replace = FALSE)
```


Creating train and validation sets

```
# First, we split the data into train, test and validation set
set.seed(100) # set seed for random sample splitting

# 80% of observations will go into the training set
split <- 0.8

# size of train set
train_size <- floor(first_split * nrow(titanic))

# id to bootstrap train set
train_id <- sample(seq_len(nrow(titanic)), train_size, replace = FALSE)

# train set
train_h <- titanic[train_id,]
```

Creating train and validation sets

```
# First, we split the data into train, test and validation set
set.seed(100) # set seed for random sample splitting

# 80% of observations will go into the training set
split <- 0.8

# size of train set
train_size <- floor(first_split * nrow(titanic))

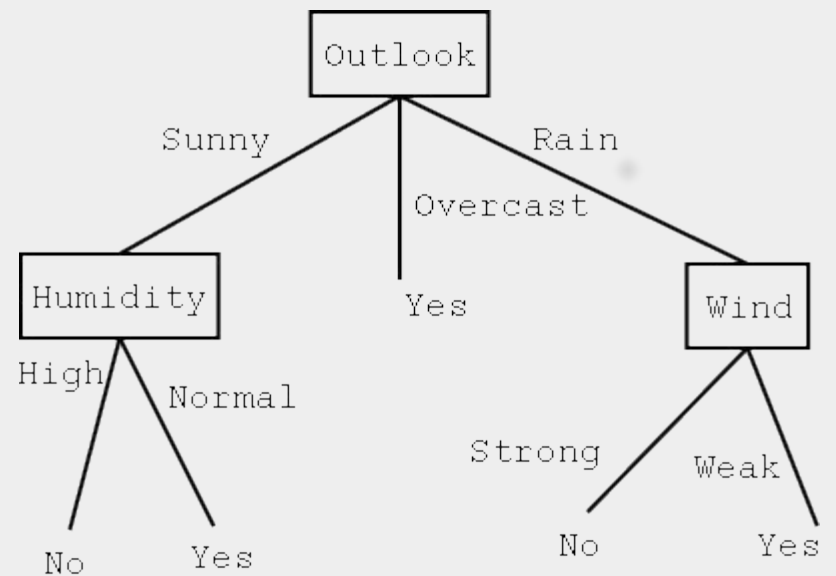
# id to bootstrap train set
train_id <- sample(seq_len(nrow(titanic)), train_size, replace = FALSE)

# train set
train_h <- titanic[train_id,]

# validation set
val_h <- titanic[-train_id,]
```

Decision tree

E.g. are people going to play tennis tonight?



Let's build a simple tree using sex, Pclass1, 2, and 3

$$1 - \left(\frac{302}{498}\right)^2 - \left(\frac{196}{498}\right)^2 = 0.477$$

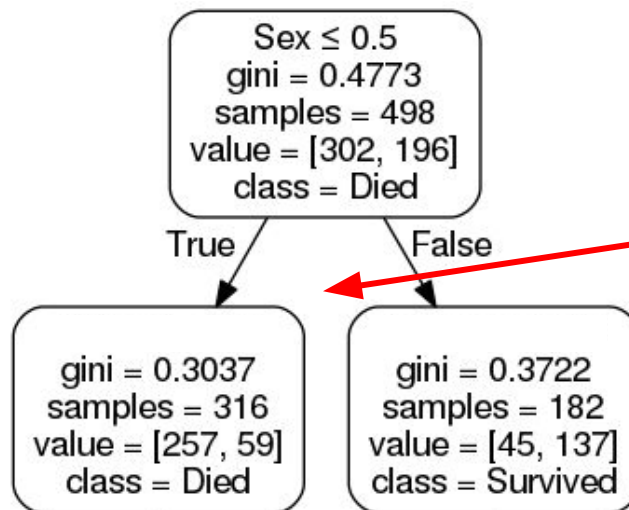
Sex ≤ 0.5
gini = 0.4773
samples = 498
value = [302, 196]
class = Died

→ We thus first decide to split the dataset by sex

We start with the initial node with [302, 196] split giving us a gini of 0.477. We then consider each of the possible variables we could split the dataset by:

- by Sex: sample=(316, 182), gini=(0.304, 0.372), wavg=0.329
 - Which gives us two nodes, one with 316 people (men) with a gini of 0.304 (257 of them died out of 316) and 182 people (women) with a gini of 0.372 (45 of them died out of 182). This gives us a weighted average of $316/(316+182) * 0.304 + 182/(316+182) * 0.372$
- by PClass1: sample=(369, 129), gini=(0.416, 0.439), wavg=0.422
- by PClass2: sample=(377, 121), gini=(0.467, 0.497), wavg=0.474
- by PClass3: sample=(250, 248), gini=(0.490, 0.336), wavg=0.413

Let's build a simple tree using sex, Pclass1, 2, and 3



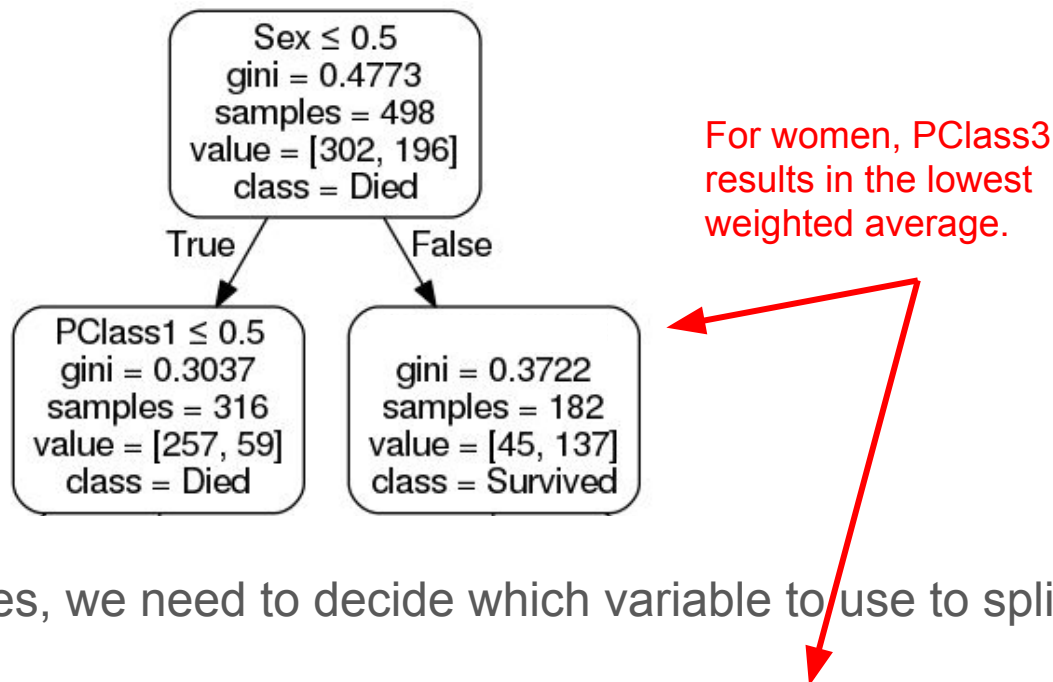
For men, PClass1 results in the lowest weighted average. We therefore decide to split the men now by PClass1

Then, for each of the two nodes, we need to decide which variable to use to split.

For the men:

- by PClass1: sample=(244, 72), gini=(0.203, 0.490), wavg=0.269
- by PClass2: sample=(247, 69), gini=(0.328, 0.205), wavg=0.301
- by PClass3: sample=(141, 175), gini=(0.400, 0.202), wavg=0.291

Let's build a simple tree using sex, Pclass1, 2, and 3

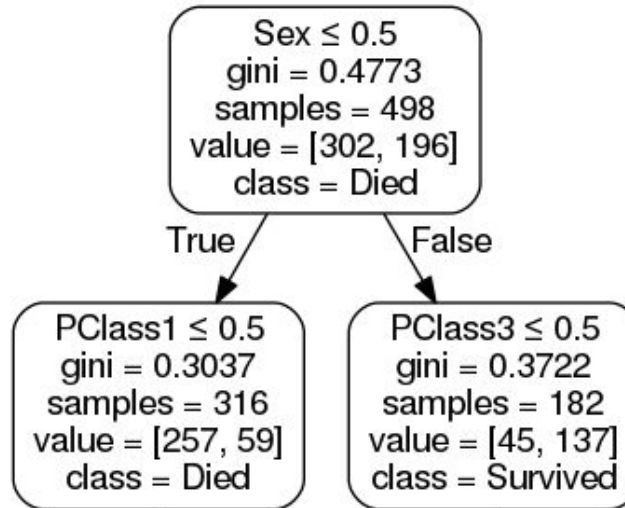


Then, for each of the two nodes, we need to decide which variable to use to split.

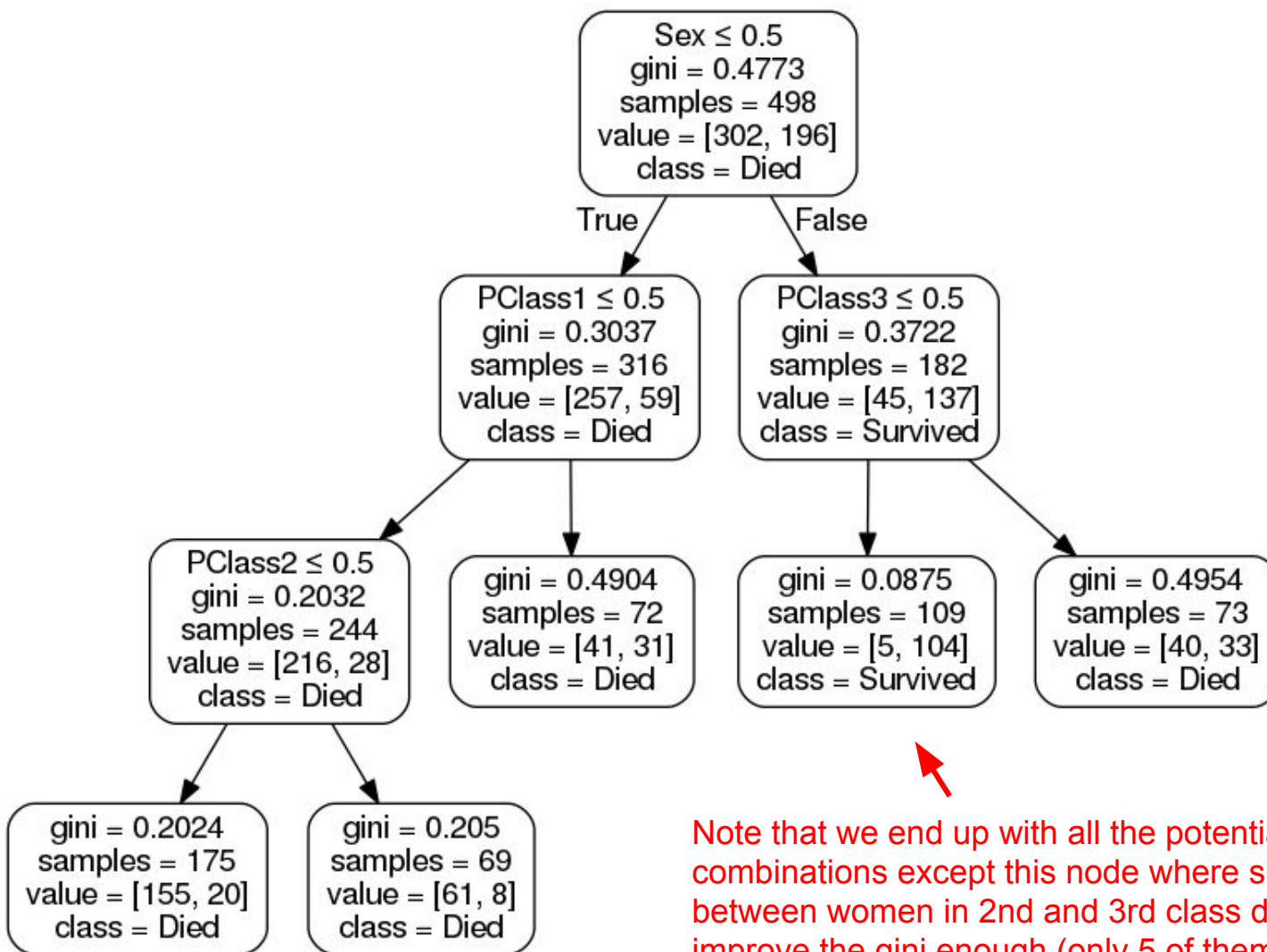
For the women:

- by PClass1: sample=(125, 57), gini=(0.456, 0.034), wavg=0.324
- by PClass2: sample=(130, 52), gini=(0.432, 0.142), wavg=0.349
- by PClass3: sample=(109, 73), gini=(0.088, 0.495), wavg=0.251

And we can keep going



Until we have a full tree




Note that we end up with all the potential combinations except this node where splitting between women in 2nd and 3rd class does not improve the gini enough (only 5 of them died)

Building trees in *R*

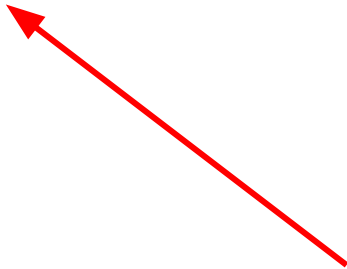
```
library(rpart) # library for tree models
clf1 <- rpart(Survived ~ Fare + Age + Pclass + Sex,
              data = train_h, maxdepth = 4, cp = 0.01)
```

Load the library `rpart`,
specify the parameters
of your tree (`max_depth`
and `cp`) and train it



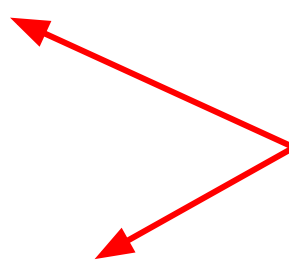
```
clf2 <- rpart(Survived ~ Fare + Age + Pclass + Sex,
              data = train_h, maxdepth = 4, cp = 0.03)
```

You can then train other
models specifying
different combinations of
parameters



Evaluating trees in *R*

```
performance <- confusionMatrix(predict(clf1, newdata = val_h,  
                                     type = "class"),  
                               val_h[, "Survived"])  
  
cat(sprintf('Accuracy: %0.3f', performance$overall[1]))  
> Accuracy: 0.771  
  
clf2 <- rpart(Survived ~ Fare + Age + Pclass + Sex,  
              data = train_h, maxdepth = 4, cp = 0.03)  
  
performance <- confusionMatrix(predict(clf2, newdata = val_h,  
                                     type = "class"),  
                               val_h[, "Survived"])  
  
cat(sprintf('Accuracy: %0.3f', performance$overall[1]))  
> Accuracy: 0.720
```



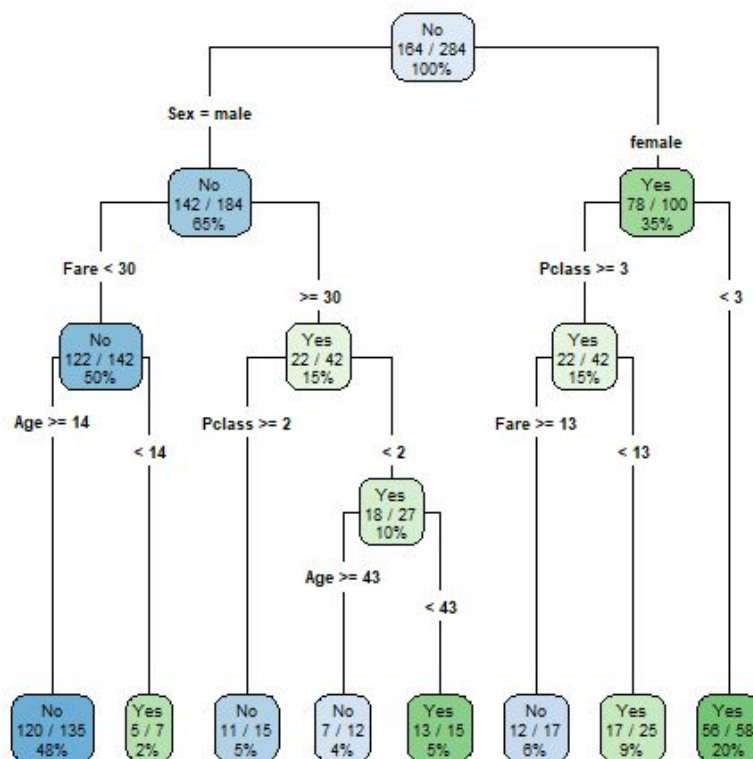
You then check the accuracy of the models you trained on the validation set using the confusion matrix from package caret

Then, select the parameters with best scores on validation. In this example it would be model 1. It has highest accuracy!

Visualize trees in R

```
library(rpart.plot)
rpart.plot(clf1, type = 4, extra = 102)
```

You can visualize your tree with function `rpart.plot` from package `rpart.plot`



Validate the performances (on unseen data) of the tree we selected

```
best_model <- clf1
performance <- confusionMatrix(predict(best_model, newdata = test_h,
                                     type = "class"),
                              test_h[, "Survived"])
cat(sprintf('Accuracy on the test set: %.3f', performance$overall[1]))
> Accuracy on the test set: 0.808
```

Once you are satisfied with the parameters, train your model one last time and test it on the test set. Here the accuracy on the test set is slightly higher than the one we have on the validation.

Yves-Alexandre de Montjoye
Imperial College London
Computational Privacy Group
deMontjoye@imperial.ac.uk