Titanic: Getting Started With R

In this lab, you must predict the fate of the passengers aboard the RMS Titanic, which famously sank in the Atlantic ocean during its maiden voyage to New York City after colliding with an iceberg.

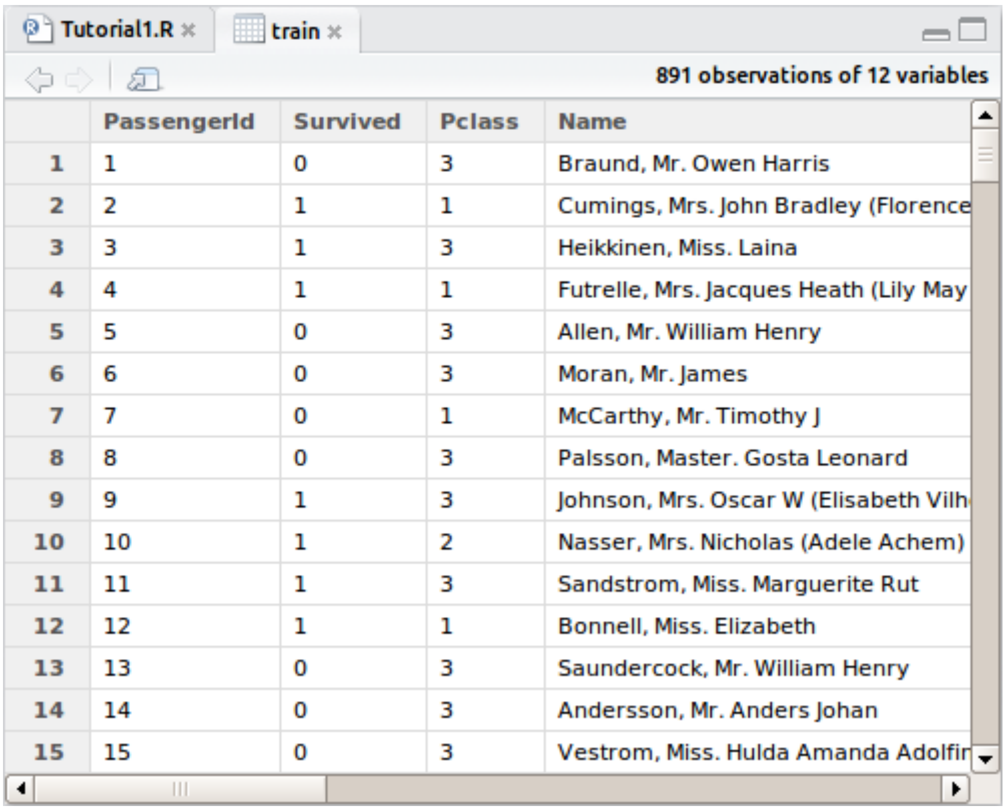
While there could hardly be a more chaotic event than frightened people scrambling to escape a sinking ship, the disaster is famous for saving "women and children first". With an inadequate number of lifeboats available only a fraction of the passengers survived, and through this series of lessons, we'll try to predict who they were.

You are given two datasets (available from the lab folder):

- a training set, complete with the outcome (or target variable) for a group of passengers as well as a collection of other parameters such as their age, gender, etc. This is the dataset on which you must train your predictive model.
- a test set, for which you must predict the now unknown target variable based on the other passenger attributes that are provided for both datasets.

The first thing you will want to do is set your working directory. This changes the default location for all file input and output that you will do in the current session. RStudio makes this easy, simply click Session -> Set Working Directory -> Choose Directory… and navigate to where you saved your train and test sets. You should notice it has entered the command you would have typed to do this manually in the console.

Okay, let's load the data and have a look at it. In the top-right pane, hit 'Import Dataset' and select train.csv. There shouldn't be any need to adjust the defaults for this dataset, so simply click 'Import'. For some datasets, headings or delimiters may not be automatically detected and this window would let you adjust the way it is imported. You will again see the manual command to import the dataset in the console, a new object in your environment pane, as well as a preview of the dataset in the script pane.



891 observations of 12 variables

	PassengerId	Survived	Pclass	Name
1	1	0	3	Braund, Mr. Owen Harris
2	2	1	1	Cumings, Mrs. John Bradley (Florence
3	3	1	3	Heikkinen, Miss. Laina
4	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May
5	5	0	3	Allen, Mr. William Henry
6	6	0	3	Moran, Mr. James
7	7	0	1	McCarthy, Mr. Timothy J
8	8	0	3	Palsson, Master. Gosta Leonard
9	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilh
10	10	1	2	Nasser, Mrs. Nicholas (Adele Achem)
11	11	1	3	Sandstrom, Miss. Marguerite Rut
12	12	1	1	Bonnell, Miss. Elizabeth
13	13	0	3	Saundercock, Mr. William Henry
14	14	0	3	Andersson, Mr. Anders Johan
15	15	0	3	Vestrom, Miss. Hulda Amanda Adolfir

You may recognize the preview as being similar to a spreadsheet, the main difference is that you can only interact with it through the R programming language. You will see columns of data that match all the variables we saw on the Kaggle download page earlier. Import the test.csv dataset in the same manner. Go ahead and take a look through some of the information in both datasets before moving on. At any time during this tutorial you can preview what's changed in your dataset by clicking the object in the explorer and the preview will open up again. It won't refresh as you make changes though.

Copy both import commands into your script. It is also good practice to comment you code; you do this by adding the hash/pound symbol `#` to the beginning of any line. The purpose of code comments is simply to inform a reader, or yourself, of what the section of code is doing. In this case you may wish to add `# Set working directory and import datafiles` to the top of your file. You may also wish to put some additional information at the top such as your name, the date, or the overall purpose of the script.

The structure that our data is stored in is called a dataframe in R. You should notice in the object explorer the two dataframes' dimensions. We see that there are 891 observations (rows) in the training set with 12 variables each. The test set is smaller, with only 418 passengers' fate to predict, and only 11 variables since the 'Survived' column is missing. It wouldn't be much of a challenge if it weren't right? That's what we're here to predict.

Let's take a quick look at the structure of the dataframe, ie the types of variables that were loaded. We will use the `str` command for this:

```
> str(train)
'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 0 1 1 ...
 $ Pclass     : int  3 1 3 1 3 3 1 3 3 2 ...
 $ Name       : Factor w/ 891 levels "Abbing, Mr. Anthony",..: 109 191 358
277 16 559 520 629 416 581 ...
 $ Sex        : Factor w/ 2 levels "female","male": 2 1 1 1 2 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",..: 525 596 662 50
473 276 86 396 345 133 ...
 $ Fare       : num  7.25 71.28 7.92 53.1 8.05 ...
 $ Cabin      : Factor w/ 148 levels "","A10","A14",..: 1 83 1 57 1 1 131 1
1 1 ...
 $ Embarked   : Factor w/ 4 levels "","C","Q","S": 4 2 4 4 4 3 4 4 4 2 ...
```

Let’s talk about data types we see here. An ‘int’ is an integer which can only store whole numbers, a ‘num’ is a numeric variable which is able to hold decimals, and a ‘factor’ is like a category. By default, R will import all text strings as factors, and that is okay here, we can convert them back to text later if we want to manipulate them, but if the dataset has a lot of text that we know we will want to work with, we could have imported the file with

```
train <- read.csv("train.csv", stringsAsFactors=FALSE)
```

In this case, the passengers name, their ticket number and cabin have all been imported as factors. We can see that the name variable has 891 levels, that means that no two passengers share the same factor level as that’s the total number of rows. For the other two, there are fewer levels, probably because there are missing values in there. For now, let’s leave the import command as it was, as the only factor we’ll be working with for the near-term is the gender variable, which correctly was imported as a category.

To access columns of a dataframe, there are several options, but if you want to isolate a single column of the dataframe, use the dollar sign operator. Try this is in the console: `train$Survived`. You should see a vector of the fates of the passengers in the training set. You can feed this vector to a function too. Let’s try `table(train$Survived)`

```
> table(train$Survived)
  0   1
549 342
```

The table command is one of the most basic summary statistics functions in R, it runs through the vector you gave it and simply counts the occurrence of each value in it. We see that in the training set, 342 passengers survived, while 549 died. How about a proportion? Well, we can send the output of one function into another. So now give `prop.table()` the output of the table function as input:

```
> prop.table(table(train$Survived))
        0         1
0.6161616 0.3838384
```

Okay, that's a bit more readable. 38% of passengers survived the disaster in the training set. This of course means that most people aboard perished. So are you ready to make your first prediction? Since most people died in our training set, perhaps it's a good start to assume that everyone in the test set did too? A bit morbid perhaps, but let's break the ice (so to speak) and send in a prediction.

Some more R syntax to keep moving. The assignment operator is `<-` and is used to store the right hand side value to the left hand side. For instance, `x <- 3` will store the value of 3 to the variable x. In some limited cases, such as passing argument values to a function signature, the equals sign is used (you'll see this in later lessons).

Okay, so let's add our 'everyone dies' prediction to the test set dataframe. To do this we'll need to use a new command, 'rep' that simply repeats something by the number of times we tell it to:

```
test$Survived <- rep(0, 418)
```

Since there was no 'Survived' column in the dataframe, it will create one for us and repeat our '0' prediction 418 times, the number of rows we have. If this column already existed, it would overwrite it with the new values, so be careful! While not entirely necessary for this simple model, putting the prediction next to the existing data will help keep things in order later, so it's a good habit to get into for more complicated predictions. If you preview the test set dataframe now, you will find our new column at the end.

We need a PassengerId as well as our Survived predictions. So let's extract those two columns from the test dataframe, store them in a new container, and then send it to an output file:

```
submit <- data.frame(PassengerId = test$PassengerId, Survived =
test$Survived)
write.csv(submit, file = "theyallperish.csv", row.names = FALSE)
```

The data.frame command has created a new dataframe with the headings consistent with those from the test set, go ahead and take a look by previewing it. The write.csv command has sent that dataframe out to a CSV file, and importantly excluded the row numbers.

The disaster was famous for saving “women and children first”, so let’s take a look at the Sex and Age variables to see if any patterns are evident. We’ll start with the gender of the passengers. After reloading the data into R, take a look at the summary of this variable:

```
> summary(train$Sex)
female   male
   314    577
```

So we see that the majority of passengers were male. Now let’s expand the proportion table command we used last time to do a two-way comparison on the number of males and females that survived:

```
> prop.table(table(train$Sex, train$Survived))
                  0          1
  female 0.09090909 0.26150393
  male   0.52525253 0.12233446
```

Well that’s not very clean, the proportion table command by default takes each entry in the table and divides by the total number of passengers. What we want to see is the row-wise proportion, ie, the proportion of each sex that survived, as separate groups. So we need to tell the command to give us proportions in the 1st dimension which stands for the rows (using ‘2’ instead would give you column proportions):

```
> prop.table(table(train$Sex, train$Survived),1)
                 0         1
  female 0.2579618 0.7420382
  male   0.8110919 0.1889081
```

Okay, that’s better. We now can see that the majority of females aboard survived, and a very low percentage of males did. In our last prediction we said they all met Davy Jones, so changing our prediction for this new insight should give us a big gain on the leaderboard! Let's update our old prediction and introduce some more R syntax:

```
test$Survived <- 0
test$Survived[test$Sex == 'female'] <- 1
```

Here we have begun with adding the 'everyone dies' prediction column as before, except that we'll ditch the rep command and just assign the zero to the whole column, it has the same effect. We then altered that same column with 1's for the subset of passengers where the variable 'Sex' is equal to 'female'.

We just used two new pieces of R syntax, the equality operator, ==, and the square bracket operator. The square brackets create a subset of the total dataframe, and apply our assignment of '1' to only those rows that meet the criteria specified. The double equals sign no longer works as an assignment here, now it is a boolean test to see if they are already equal.

Nice! We're getting there, but let's start digging into the age variable now:

```
> summary(train$Age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   0.42   20.12   28.00   29.70   38.00   80.00     177
```

It is possible for values to be missing in data analytics, and this can cause a variety of problems out in the real world that can be quite difficult to deal with at times. For now we could assume that the 177 missing values are the average age of the rest of the passengers, ie. late twenties.

Our last few tables were on categorical variables, ie. they only had a few values. Now we have a continuous variable which makes drawing proportion tables almost useless, as there may only be one or two passengers for each age! So, let's create a new variable, Child, to indicate whether the passenger is below the age of 18:

```
train$Child <- 0
train$Child[train$Age < 18] <- 1
```

As with our prediction column, we have now created a new column in the training set dataframe indicating whether the passenger was a child or not. Beginning with the assumption that they were an adult, and then overwriting the value for passengers below the age of 18. To do this we used the less than operator, which is another boolean test, similar to the equality check used in our last predictions. If you click on the train object in the explorer, you will see that any passengers with an age of NA have been assigned a zero, this is because the NA will fail any boolean test. This is what we wanted though, since we had decided to use the average age, which was an adult.

Now we want to create a table with both gender and age to see the survival proportions for different subsets. Unfortunately our proportion table isn't equipped for this, so we're going to have to use a new R command, `aggregate`. First let's try to find the number of survivors for the different subsets:

```
> aggregate(Survived ~ Child + Sex, data=train, FUN=sum)
  Child    Sex Survived
1     0 female      195
2     1 female       38
3     0   male       86
4     1   male       23
```

The `aggregate` command takes a formula with the target variable on the left hand side of the tilde symbol and the variables to subset over on the right. We then tell it which dataframe to look at with the data argument, and finally what function to apply to these subsets. The command above subsets the whole dataframe over the different possible combinations of the age and gender variables and applies the `sum` function to the Survived vector for each of these subsets. As our target variable is coded as a 1 for survived, and 0 for not, the result of summing is the number of survivors. But we don't know the total number of people in each subset; let's find out:

```
> aggregate(Survived ~ Child + Sex, data=train, FUN=length)
  Child    Sex Survived
1     0 female      259
2     1 female       55
3     0   male      519
4     1   male       58
```

This simply looked at the length of the Survived vector for each subset and output the result, the fact that any of them were 0's or 1's was irrelevant for the length function. Now we have the totals for each group of passengers, but really, we would like to know the proportions again. To do this is a little more complicated. We need to create a function that takes the subset vector as input and applies both the sum and length commands to it, and then does the division to give us a proportion. Here is the syntax:

```
> aggregate(Survived ~ Child + Sex, data=train, FUN=function(x)
{sum(x)/length(x)})
  Child    Sex  Survived
```

```
1      0 female 0.7528958
2      1 female 0.6909091
3      0   male 0.1657033
4      1   male 0.3965517
```

Well, it still appears that if a passenger is female most survive, and if they were male most don't, regardless of whether they were a child or not. So we haven't got anything to change our predictions on here. Let's take a look at a couple of other potentially interesting variables to see if we can find anything more: the class that they were riding in, and what they paid for their ticket.

While the class variable is limited to a manageable 3 values, the fare is again a continuous variable that needs to be reduced to something that can be easily tabulated. Let's bin the fares into less than $10, between $10 and $20, $20 to $30 and more than $30 and store it to a new variable:

```
train$Fare2 <- '30+'
train$Fare2[train$Fare < 30 & train$Fare >= 20] <- '20-30'
train$Fare2[train$Fare < 20 & train$Fare >= 10] <- '10-20'
train$Fare2[train$Fare < 10] <- '<10'
```

Now let's run a longer aggregate function to see if there's anything interesting to work with here:

```
> aggregate(Survived ~ Fare2 + Pclass + Sex, data=train, FUN=function(x)
{sum(x)/length(x)})
   Fare2 Pclass    Sex  Survived
1  20-30      1 female 0.8333333
2    30+      1 female 0.9772727
3  10-20      2 female 0.9142857
4  20-30      2 female 0.9000000
5    30+      2 female 1.0000000
6    <10      3 female 0.5937500
7  10-20      3 female 0.5813953
8  20-30      3 female 0.3333333 **
9    30+      3 female 0.1250000 **
10   <10      1   male 0.0000000
11 20-30      1   male 0.4000000
12   30+      1   male 0.3837209
13   <10      2   male 0.0000000
```

```
14 10-20      2   male 0.1587302
15 20-30      2   male 0.1600000
16   30+      2   male 0.2142857
17   <10      3   male 0.1115385
18 10-20      3   male 0.2368421
19 20-30      3   male 0.1250000
20   30+      3   male 0.2400000
```

While the majority of males, regardless of class or fare still don't do so well, we notice that most of the class 3 women who paid more than $20 for their ticket actually also miss out on a lifeboat, I've indicated these with asterisks, but R won't know what you're looking for, so they won't show up in the console.

It's a little hard to imagine why someone in third class with an expensive ticket would be worse off in the accident, but perhaps those more expensive cabins were located close to the iceberg impact site, or further from exit stairs? Whatever the cause, let's make a new prediction based on the new insights.

```
test$Survived <- 0
test$Survived[test$Sex == 'female'] <- 1
test$Survived[test$Sex == 'female' & test$Pclass == 3 & test$Fare >= 20] <-
0
```

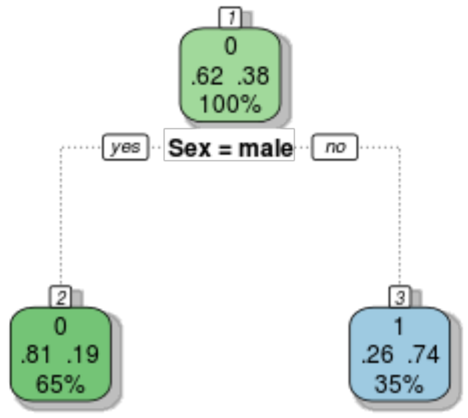
Most of the above code should be familiar to you by now. The only exception would be that there are multiple boolean checks all stringed together for the last adjustment. For more complicated boolean logic, you can combine the logical AND operator `&` with the logical OR operator `|`.

Decision Trees

We have sliced and diced the data to try and find subsets of the passengers that were more, or less, likely to survive the disaster. To find more fine-grained subsets with predictive ability would require a lot of time to adjust our bin sizes and look at the interaction of many different variables. Luckily there is a simple and elegant algorithm that can do this work for us. Today we're going to use machine learning to build decision trees to do the heavy lifting for us.

Decision trees have a number of advantages. They are what's known as a glass-box model, after the model has found the patterns in the data you can see exactly what decisions will be made for unseen data that you want to predict. They are also intuitive and can be read by people with little experience in machine learning after a brief explanation. Finally, they are the basis for some of the most powerful and popular machine learning algorithms.

I won’t get into the mathematics here, but conceptually, the algorithm starts with all of the data at the root node (drawn at the top) and scans all of the variables for the best one to split on. The way it measures this is to make the split on the variable that results in the most pure nodes below it, ie with either the most 1’s or the most 0’s in the resulting buckets. But let’s look at something more familiar to get the idea. Here we draw a decision tree for only the gender variable, and some familiar numbers jump out:



Let’s decode the numbers shown on this new representation of our original manual gender-based model. The root node, at the top, shows our tutorial one insights, 62% of passengers die, while 38% survive. The number above these proportions indicates the way that the node is voting (recall we decided at this top level that everyone would die, or be coded as zero) and the number below indicates the proportion of the population that resides in this node, or bucket (here at the top level it is everyone, 100%).

So far, so good. Now let’s travel down the tree branches to the next nodes down the tree. If the passenger was a male, indicated by the boolean choice below the node, you move left, and if female, right. The survival proportions exactly match those we found in tutorial two through our proportion tables. If the passenger was male, only 19% survive, so the bucket votes that everyone here (65% of passengers) perish, while the female bucket votes in the opposite manner, most of them survive as we saw before. In fact, the above decision tree is an exact representation of our gender model from last lesson.

The final nodes at the bottom of the decision tree are known as terminal nodes, or sometimes as leaf nodes. After all the boolean choices have been made for a given passenger, they will end up in one of the leaf nodes, and the majority vote of all passengers in that bucket determine how we will predict for new passengers with unknown fates.

We can grow this tree until every passenger is classified and all the nodes are marked with either 0% or 100% chance of survival… All that chopping and comparing of subsets is taken care of for us in the blink of an eye!

Decision trees do have some drawbacks though, they are greedy. They make the decision on the current node which appear to be the best at the time, but are unable to change their minds as they grow new nodes.

Perhaps a better, more pure, tree would have been grown if the gender split occurred later? It is really hard to tell, there are a huge number of decisions that could be made, and exploring every possible version of a tree is extremely computationally expensive. This is why the greedy algorithm is used.

As an example, imagine a cashier in a make-believe world with a currency including 25c, 15c and 1c coins. The cashier must make change for 30c using the smallest number of coins possible. A greedy algorithm would start with the coin that leaves the smallest amount of change left to pay:

- Greedy: 25 + 1 + 1 + 1 + 1 + 1 = 30c, with 6 coins
- Optimal: 15 + 15 = 30c, with 2 coins

Clearly the greedy cashier algorithm failed to find the best solution here, and the same is true with decision trees. Though they usually do a great job given their speed and the other advantages we already mentioned, the optimal solution is not guaranteed. Decision trees are also prone to overfitting which requires us to use caution with how deep we grow them as we'll see later.

So, let's get started with our first real algorithm! Now we start to open up the power of R: its packages. R is extremely extensible, you'd be hard pressed to find a package that doesn't automatically do what you need. There's thousands of options out there written by people who needed the functionality and published their work. You can easily add these packages within R with just a couple of commands.

The one we'll need for this lesson comes with R. It's called `rpart` for 'Recursive Partitioning and Regression Trees' and uses the CART decision tree algorithm. While `rpart` comes with base R, you still need to import the functionality each time you want to use it. Go ahead:

```
library(rpart)
```

Now let's build our first model. Let's take a quick review of the possible variables we could look at. Last time we used aggregate and proportion tables to compare gender, age, class and fare. But we never did investigate SibSp, Parch or Embarked. The remaining variables of passenger name, ticket number and cabin number are all unique identifiers for now; they don't give any new subsets that would be interesting for a decision tree. So let's build a tree off everything else.

The format of the `rpart` command works similarly to the `aggregate` function we used in tutorial 2. You feed it the equation, headed up by the variable of interest and followed by the variables used for prediction. You then point it at the data, and for now, follow with the type of prediction you want to run (see `?rpart` for more info). If you wanted to predict a continuous variable, such as age, you may use `method=”anova”`. This would run generate decimal quantities for you. But here, we just want a one or a zero, so `method=”class”` is appropriate:

```
fit <- rpart(Survived ~ Pclass + Sex + Age + SibSp + Parch + Fare +
Embarked, data=train, method="class")
```

Let’s examine the tree. There are a lot of ways to do this, and the built-in version requires running

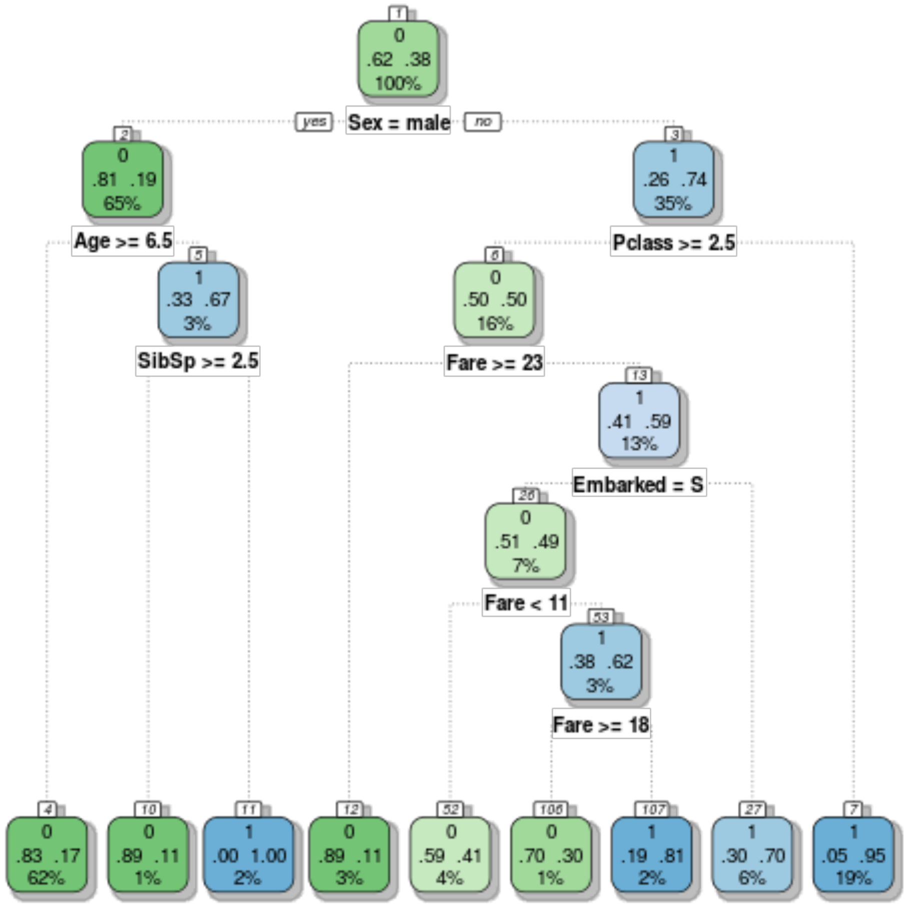
```
plot(fit)
text(fit)
```

Hmm, not very pretty or insightful. To get some more informative graphics, you will need to install some external packages. As I mentioned, tons of world-class developers donate their time and energy to the R project by contributing powerful packages to CRAN, free of charge. You can install them from within R using `install.packages()`, and load them as before with `library()`. Here are the ones we need for some better graphics for `rpart`:

```
install.packages('rattle')
install.packages('rpart.plot')
install.packages('RColorBrewer')
library(rattle)
library(rpart.plot)
library(RColorBrewer)
```

Let’s try rendering this tree a bit nicer with `fancyRpartPlot` (of course).

```
fancyRpartPlot(fit)
```



Okay, now we've got somewhere readable. The decisions that have been found go a lot deeper than what we saw last time when we looked for them manually. Decisions have been found for the SipSp variable, as well as the port of embarkation one that we didn't even look at. And on the male side, the kids younger than 6 years old have a better chance of survival, even if there weren't too many aboard. That resonates with the famous naval law we mentioned earlier. It all looks very promising.

To make a prediction from this tree doesn't require all the subsetting and overwriting we did last lesson, it's actually a lot easier.

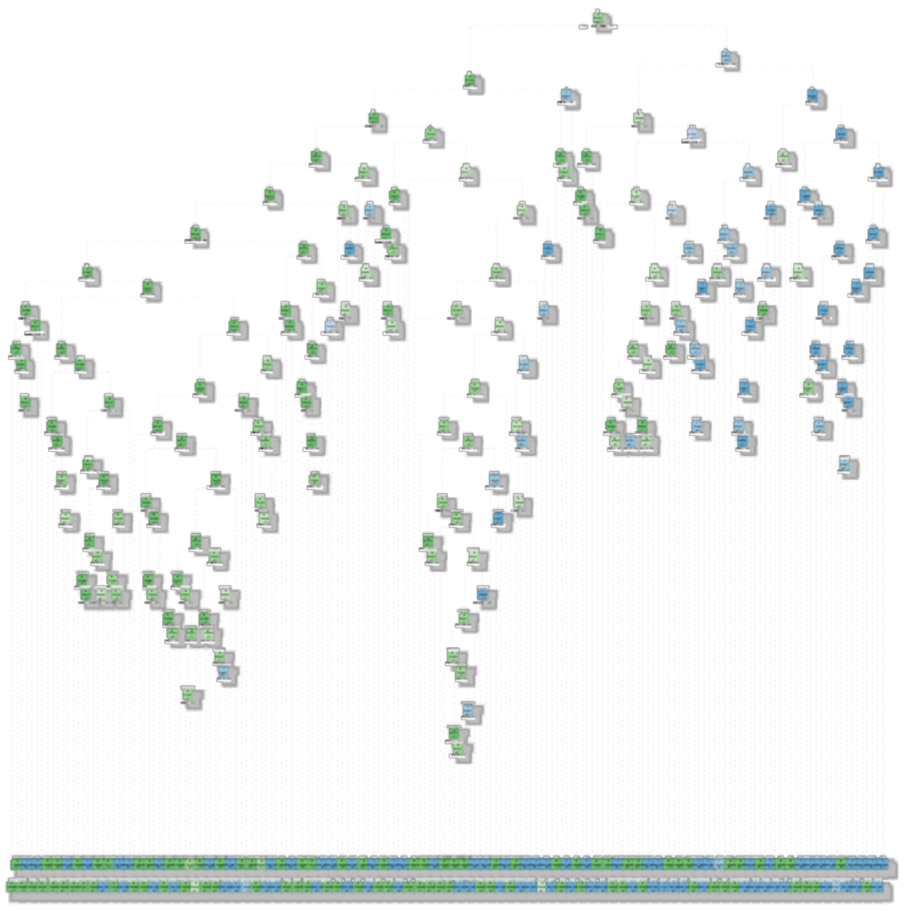
```
Prediction <- predict(fit, test, type = "class")
submit <- data.frame(PassengerId = test$PassengerId, Survived = Prediction)
write.csv(submit, file = "myfirstdtree.csv", row.names = FALSE)
```

Here we have called rpart's predict function. Here we point the function to the model's fit object, which contains all of the decisions we see above, and tell it to work its magic on the test dataframe. No need to tell it which variables we originally used in the model-building phase, it automatically looks for them and will certainly let you know if something is wrong. Finally we tell it to again use the class method (for ones and zeros output) and as before write the output to a dataframe and submission file.

The `rpart` package automatically caps the depth that the tree grows by using a metric called complexity which stops the resulting model from getting too out of hand. But we already saw that a more complex model than what we made ourselves did a bit better, so why not go all out and override the defaults? Let's do it.

You can find the default limits by typing `?rpart.control`. The first one we want to unleash is the `cp` parameter, this is the metric that stops splits that aren't deemed important enough. The other one we want to open up is `minsplit` which governs how many passengers must sit in a bucket before even looking for a split. Let's max both out and reduce `cp` to zero and `minsplit` to 2 (no split would obviously be possible for a single passenger in a bucket):

```
fit <- rpart(Survived ~ Pclass + Sex + Age + SibSp + Parch + Fare +
Embarked, data=train,
             method="class", control=rpart.control(minsplit=2, cp=0))
fancyRpartPlot(fit)
```



Okay, I can't even see what's going on here, but with that much subsetting and mining for tiny nuggets of truth, how could we go wrong!

Even our simple gender model did better! What went wrong? Welcome to overfitting.

Overfitting is technically defined as a model that performs better on a training set than another simpler model, but does worse on unseen data, as we saw here. We went too far and grew our decision tree out to encompass massively complex rules that may not generalize to unknown passengers. Perhaps that 34 year old female in third class who paid $20.17 for a ticket from Southampton with a sister and mother aboard may have been a bit of a rare case after all.

The point of this exercise was that you must use caution with decision trees. While this particular tree may have been 100% accurate on the data that you trained it on, even a trivial tree with only one rule could beat it on unseen data. You just overfit big time!

Use caution with decision trees, and any other algorithm actually, or you can find yourself making rules from the noise you've mistaken for signal!

Before moving on, I encourage you to have a play with the various control parameters we saw in the rpart.control help file. Perhaps you can find a tree that does a little better by either growing it out further, or reigning it in. You can also manually trim trees in R with these commands:

```
fit <- rpart(Survived ~ Pclass + Sex + Age + SibSp + Parch + Fare +
Embarked, data=train,
             method="class", control=rpart.control( your controls ))
new.fit <- prp(fit,snip=TRUE)$obj
fancyRpartPlot(new.fit)
```

An interactive version of the decision tree will appear in the plot tab where you simply click on the nodes that you want to kill. Once you're satisfied with the tree, hit 'quit' and it will be stored to the new.fit object. Try to look for overly complex decisions being made, and kill the nodes that appear to go to far.

Feature Engineering

Feature engineering is so important to how your model performs, that even a simple model with great features can outperform a complicated algorithm with poor ones. In fact, feature engineering has been described as "easily the most important factor" in determining the success or failure of your predictive model. Feature engineering really boils down to the human element in machine learning. How much you understand the data, with your human intuition and creativity, can make the difference.

So what is feature engineering? It can mean many things to different problems, but in Titanic problem it could mean chopping, and combining different attributes to squeeze a little bit more value from them. In general, an engineered feature may be easier for a machine learning algorithm to digest and make rules from than the variables it was derived from.

The initial suspects for gaining more machine learning mojo from are the three text fields that we never sent into our decision trees last time. While the ticket number, cabin, and name were all unique to each passenger; perhaps parts of those text strings could be extracted to build a new predictive attribute. Let's start with the name field. If we take a glance at the first passenger's name we see the following:

```
> train$Name[1]
[1] Braund, Mr. Owen Harris
891 Levels: Abbing, Mr. Anthony Abbott, Mr. Rossmore Edward ... Zimmerman,
Mr. Leo
```

Previously we have only accessed passenger groups by subsetting, now we access an individual by using the row number, 1, as an index instead. Okay, no one else on the boat had that name, that's pretty much certain, but what else might they have shared? Well, I'm sure there were plenty of Mr's aboard. Perhaps the persons title might give us a little more insight.

If we scroll through the dataset we see many more titles including Miss, Mrs, Master, and even the Countess! The title 'Master' is a bit outdated now, but back in these days, it was reserved for unmarried boys. Additionally, the nobility such as our Countess would probably act differently to the lowly proletariat too. There seems to be a fair few possibilities of patterns in this that may dig deeper than the combinations of age, gender, etc that we looked at before.

In order to extract these titles to make new variables, we'll need to perform the same actions on both the training and testing set, so that the features are available for growing our decision trees, and making predictions on the unseen testing data. An easy way to perform the same processes on both datasets at the same time is to merge them. In R we can use `rbind`, which stands for row bind, so long as both dataframes have the same columns as each other. Since we obviously lack the Survived column in our test set, let's create one full of missing values (NAs) and then row bind the two datasets together:

```
test$Survived <- NA
combi <- rbind(train, test)
```

Now we have a new dataframe called combi with all the same rows as the original two datasets, stacked in the order in which we specified: train first, and test second.

If you look back at the output of our inquiry on Owen, his name is still encoded as a factor. As we mentioned earlier, strings are automatically imported as factors in R, even if it doesn't make sense. So we need to cast this column back into a text string. To do this we use `as.character`. Let's do this and then take another look at Owen:

```
> combi$Name <- as.character(combi$Name)
> combi$Name[1]
[1] "Braund, Mr. Owen Harris"
```

Excellent, no more levels, now it's just pure text. In order to break apart a string, we need some hooks to tell the program to look for. Nicely, we see that there is a comma right after the person's last name, and a full stop after their title. We can easily use the function `strsplit`, which stands for string split, to break apart our original name over these two symbols. Let's try it out on Mr. Braund:

```
> strsplit(combi$Name[1], split='[,.]')
[[1]]
[1] "Braund"       " Mr"          " Owen Harris"
```

Okay, good. Here we have sent strsplit the cell of interest, and given it some symbols to chose from when splitting the string up, either a comma or period. Those symbols in the square brackets are called regular expressions, though this is a very simple one, and if you plan on working with a lot of text I would certainly recommend getting used to using them!

We see that the title has been broken out on its own, though there's a strange space before it begins because the comma occurred at the end of the surname. But how do we get to that title piece and clear out the rest of the stuff we don't want? An index `[[1]]` is printed before the text portions. Let's try to dig into this new type of container by appending all those square brackets to the original command:

```
> strsplit(combi$Name[1], split='[,.]')[[1]]
[1] "Braund"       " Mr"          " Owen Harris"
```

Getting there! String split uses a doubly stacked matrix because it can never be sure that a given regex will have the same number of pieces. If there were more commas or periods in the name, it would create more segments, so it hides them a level deeper to maintain the rectangular types of containers that we are used to in things like spreadsheets, or now dataframes! Let's go a level deeper into the indexing mess and extract the title. It's the second item in this nested list, so let's dig in to index number 2 of this new container:

```
> strsplit(combi$Name[1], split='[,.]')[[1]][2]
[1] " Mr"
```

Great. We have isolated the title we wanted at last. But how to *apply* this transformation to every row of the combined train/test dataframe? Luckily, R has some extremely useful functions that apply more complication functions one row at a time. As we had to dig into this container to get the title, simply trying to run `combi$Title <- strsplit(combi$Name, split='[,.]')[[1]][2]` over the whole name vector would result in all of our rows having the same value of Mr., so we need to work a bit harder. Unsurprisingly applying a function to a lot of cells in a dataframe or vector uses the `apply` suite of functions of R:

```
combi$Title <- sapply(combi$Name, FUN=function(x) {strsplit(x,
split='[,.]')[[1]][2]})
```

R's apply functions all work in slightly different ways, but `sapply` will work great here. We feed `sapply` our vector of names and our function that we just came up with. It runs through the rows of the vector of names, and sends each name to the function. The results of all these string splits are all combined up into a vector as output from the `sapply` function, which we then store to a new column in our original dataframe, called Title.

Finally, we may wish to strip off those spaces from the beginning of the titles. Here we can just substitute the first occurrence of a space with nothing. We can use `sub` for this (`gsub` would replace all spaces, poor 'the Countess' would look strange then though):

```
combi$Title <- sub(' ', '', combi$Title)
```

Alright, we now have a nice new column of titles, let's have a look at it:

```
> table(combi$Title)
        Capt           Col          Don          Dona            Dr
Jonkheer         Lady
           1             4            1             1             8
1            1
       Major        Master         Miss          Mlle           Mme
Mr           Mrs
           2            61          260             2             1
757          197
```

```
         Ms           Rev          Sir the Countess
          2             8            1            1
```

Hmm, there are a few very rare titles in here that won't give our model much to work with, so let's combine a few of the most unusual ones. We'll begin with the French. Mademoiselle and Madame are pretty similar (so long as you don't mind offending) so let's combine them into a single category:

```
combi$Title[combi$Title %in% c('Mme', 'Mlle')] <- 'Mlle'
```

What have we done here? The `%in%` operator checks to see if a value is part of the vector we're comparing it to. So here we are combining two titles, 'Mme' and 'Mlle', into a new temporary vector using the `c()` operator and seeing if any of the existing titles in the entire Title column match either of them. We then replace any match with 'Mlle'.

Let's keep looking for redundancy. It seems the very rich are a bit of a problem for our set here too. For the men, we have a handful of titles that only one or two have been blessed with: Captain, Don, Major and Sir. All of these are either military titles, or rich fellas who were born with vast tracts of land.

For the ladies, we have Dona, Lady, Jonkheer (*see comments below), and of course our Countess. All of these are again the rich folks, and may have acted somewhat similarly due to their noble birth. Let's combine these two groups and reduce the number of factor levels to something that a decision tree might make sense of:

```
combi$Title[combi$Title %in% c('Capt', 'Don', 'Major', 'Sir')] <- 'Sir'
combi$Title[combi$Title %in% c('Dona', 'Lady', 'the Countess', 'Jonkheer')]
<- 'Lady'
```

Our final step is to change the variable type back to a factor, as these are essentially categories that we have created:

```
combi$Title <- factor(combi$Title)
```

Alright. We're done with the passenger's title now. What else can we think up? Well, there's those two variables SibSb and Parch that indicate the number of family members the passenger is travelling with. Seems reasonable to assume that a large family might have trouble tracking down little Johnny as they all scramble to get off the sinking ship, so let's combine the two variables into a new one, FamilySize:

```
combi$FamilySize <- combi$SibSp + combi$Parch + 1
```

Pretty simple! We just add the number of siblings, spouses, parents and children the passenger had with them, and plus one for their own existence of course, and have a new variable indicating the size of the family they travelled with.

Anything more? Well we just thought about a large family having issues getting to lifeboats together, but maybe *specific* families had more trouble than others? We could try to extract the Surname of the passengers and group them to find families, but a common last name such as Johnson might have a few extra non-related people aboard. In fact there are three Johnsons in a family with size 3, and another three probably unrelated Johnsons all travelling solo.

Combining the Surname with the family size though should remedy this concern. No two family-Johnson's should have the same FamilySize variable on such a small ship. So let's first extract the passengers' last names. This should be a pretty simple change from the title extraction code we ran earlier, now we just want the first part of the `strsplit` output:

```
combi$Surname <- sapply(combi$Name, FUN=function(x) {strsplit(x,
split='[,.]')[[1]][1]})
```

We then want to append the FamilySize variable to the front of it, but as we saw with factors, string operations need strings. So let's convert the FamilySize variable temporarily to a string and combine it with the Surname to get our new FamilyID variable:

```
combi$FamilyID <- paste(as.character(combi$FamilySize), combi$Surname,
sep="")
```

We used the function `paste` to bring two strings together, and told it to separate them with nothing through the sep argument. This was stored to a new column called FamilyID. But those three single Johnsons would all have the same Family ID. Given we were originally hypothesising that large families might have trouble sticking together in the panic, let's knock out any family size of two or less and call it a "small" family. This would fix the Johnson problem too.

```
combi$FamilyID[combi$FamilySize <= 2] <- 'Small'
```

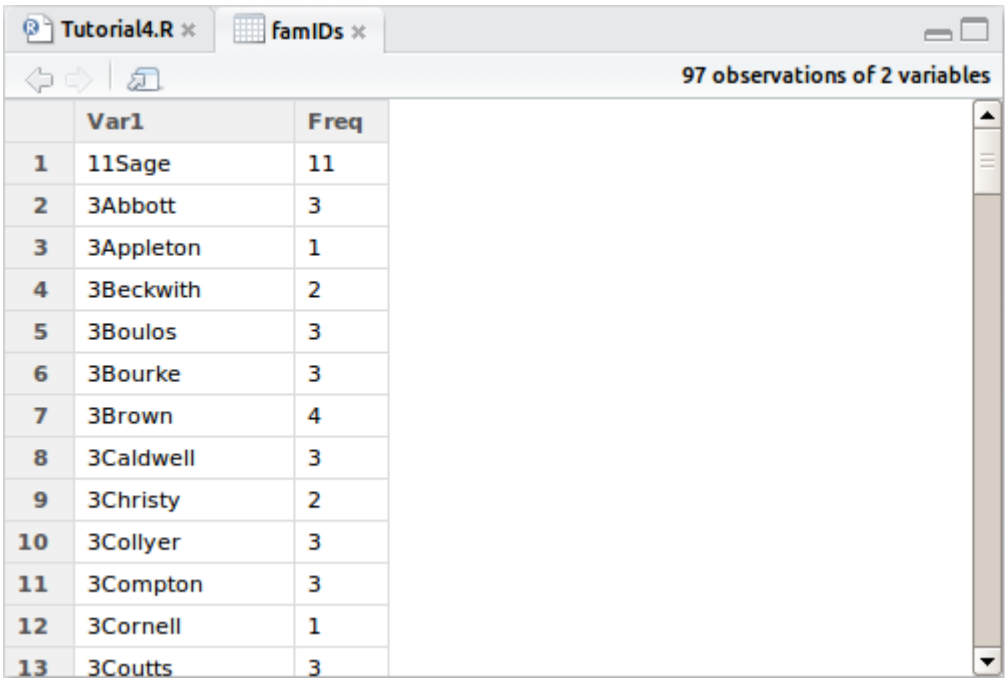
Let's see how we did for identifying these family groups:

```
> table(combi$FamilyID)
          11Sage           3Abbott         3Appleton         3Beckwith
3Boulos
              11                 3                 1                 2
3
         3Bourke            3Brown         3Caldwell         3Christy
3Collyer
               3                 4                 3                 2
3
        3Compton          3Cornell           3Coutts          3Crosby
3Danbom
               3                 1                 3                 3
3 . . .
```

Hmm, a few seemed to have slipped through the cracks here. There's plenty of FamilyIDs with only one or two members, even though we wanted only family sizes of 3 or more. Perhaps some families had different last names, but whatever the case, all these one or two people groups is what we sought to avoid with the three person cut-off. Let's begin to clean this up:

```
famIDs <- data.frame(table(combi$FamilyID))
```

Now we have stored the table above to a dataframe. Yep, you can store most tables to a dataframe if you want to, so let's take a look at it by clicking on it in the explorer:



97 observations of 2 variables

	Var1	Freq
1	11Sage	11
2	3Abbott	3
3	3Appleton	1
4	3Beckwith	2
5	3Boulos	3
6	3Bourke	3
7	3Brown	4
8	3Caldwell	3
9	3Christy	2
10	3Collyer	3
11	3Compton	3
12	3Cornell	1
13	3Coutts	3

Here we see again all those naughty families that didn't work well with our assumptions, so let's subset this dataframe to show only those unexpectedly small FamilyID groups.

```
famIDs <- famIDs[famIDs$Freq <= 2,]
```

We then need to overwrite any family IDs in our dataset for groups that were not correctly identified and finally convert it to a factor:

```
combi$FamilyID[combi$FamilyID %in% famIDs$Var1] <- 'Small'
combi$FamilyID <- factor(combi$FamilyID)
```

We are now ready to split the test and training sets back into their original states, carrying our fancy new engineered variables with them. The nicest part of what we just did is how the factors are treated in R. Behind the scenes, factors are basically stored as integers, but masked with their text names for us to look at. If you create the above factors on the isolated test and train sets separately, there is no guarantee that both groups exist in both sets.

For instance, the family '3Johnson' previously discussed does not exist in the test set. We know that all three of them survive from the training set data. If we had built our factors in isolation, there would be no factor '3Johnson' for the test set. This would upset any machine learning model because the factors between the training set used to build the model and the test set it is asked to predict for are not consistent. ie. R will throw errors at you if you try.

Because we built the factors on a single dataframe, and then split it apart after we built them, R will give all factor levels to both new dataframes, even if the factor doesn't exist in one. It will still have the factor level, but no actual observations of it in the set. Neat trick right? Let me assure you that manually updating factor levels is a pain.

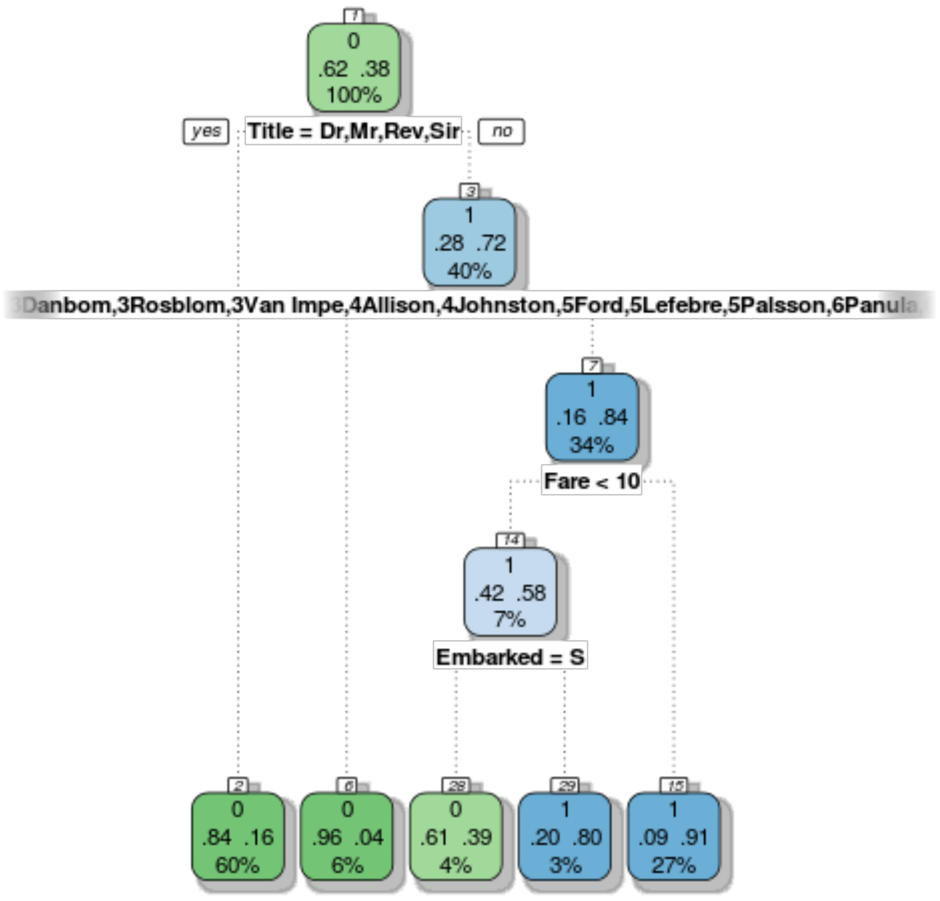
So let's break them apart and do some predictions on our new fancy engineered variables:

```
train <- combi[1:891,]
test <- combi[892:1309,]
```

Here we introduce yet another subsetting method in R; there are many depending on how you want to chop up your data. We have isolated certain ranges of rows of the combi dataset based on the sizes of the original train and test sets. The comma after that with no numbers following it indicates that we want to take ALL columns with this subset and store it to the assigned dataframe. This gives us back our original number of rows, as well as all our new variables including the consistent factor levels.

Time to do our predictions! We have a bunch of new variables, so let's send them to a new decision tree. Last time the default complexity worked out pretty well, so let's just grow a tree with the vanilla controls and see what it can do:

```
fit <- rpart(Survived ~ Pclass + Sex + Age + SibSp + Parch + Fare +
Embarked + Title + FamilySize + FamilyID,
             data=train, method="class")
```



Interestingly our new variables are basically governing our tree. Here's another drawback with decision trees that I didn't mention last time: they are biased to favour factors with many levels. Look at how our 61-level FamilyID factor is so prominent here, and the tree picked out all the families that are biased one way more than the others. This way the decision node can chop and change the data into the best way possible combination for purity of the following nodes.

But all that aside, you know should know how to create a submission from a decision tree, so let's see how it performed!

Go ahead and try and create some more engineered variables! As before, I also really encourage you to play around with the complexity parameters and maybe try trimming some deeper trees to see if it helps or hinders your rank. You may even consider excluding some variables from the tree to see if that changes anything too.

In most cases though, the title or gender variables will govern the first decision due to the greedy nature of decision trees. The bias towards many-levelled factors won't go away either, and the overfitting problem can be difficult to gauge without actually sending in submissions, but good judgement can help.

We can overcome these limitations by building an ensemble of decision trees with the powerful Random Forest algorithm.

Random Forests

Seems fitting to start with a definition,

en·sem·ble

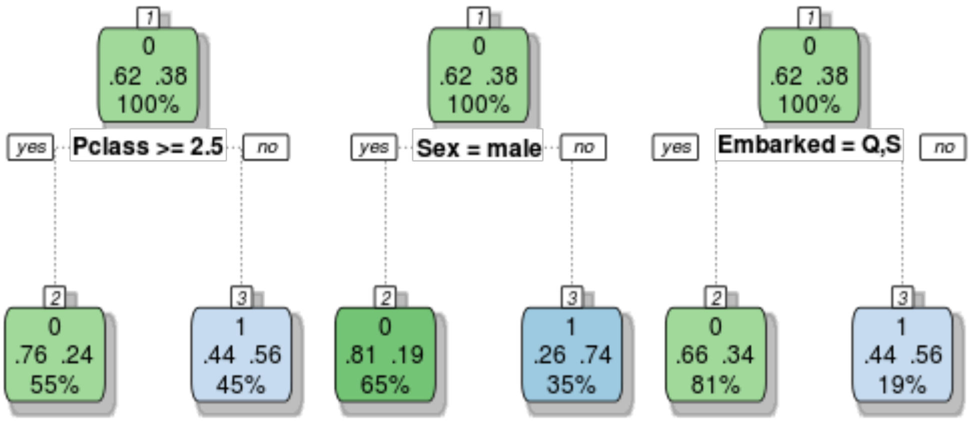
A unit or group of complementary parts that contribute to a single effect, especially:

1. A coordinated outfit or costume.
2. A coordinated set of furniture.
3. A group of musicians, singers, dancers, or actors who perform together

While I won't be teaching about how to best coordinate your work attire or living room, I think the musician metaphor works here. In an ensemble of talented instrumentalists, the issues one might have with an off-note are overpowered by the others in the group.

The same goes for machine learning. Take a large collection of individually imperfect models, and their one-off mistakes are probably not going to be made by the rest of them. If we average the results of all these models, we can sometimes find a superior model from their combination than any of the individual parts. That's how ensemble models work, they grow a lot of different models, and let their outcomes be averaged or voted across the group.

We are now well aware of the overfitting problems with decision trees. But if we grow a whole lot of them and have them vote on the outcome, we can get passed this limitation. Let's build a very small ensemble of three simple decision trees to illustrate:



Each of these trees make their classification decisions based on different variables. So let's imagine a female passenger from Southampton who rode in first class. Tree one and two would vote that she survived, but tree three votes that she perishes. If we take a vote, it's 2 to 1 in favour of her survival, so we would classify this passenger as a survivor.

Random Forest models grow trees much deeper than the decision stumps above, in fact the default behaviour is to grow each tree out as far as possible, like the overfitting tree we made in lesson three. But since the formulas for building a single decision tree are the same every time, some source of randomness is required to make these trees different from one another. Random Forests do this in two ways.

The first trick is to use bagging, for bootstrap aggregating. Bagging takes a randomized sample of the rows in your training set, with replacement. This is easy to simulate in R using the sample function. Let's say we wanted to perform bagging on a training set with 10 rows.

```
> sample(1:10, replace = TRUE)
 [1]  3  1  9  1  7 10 10  2  2  9
```

In this simulation, we would still have 10 rows to work with, but rows 1, 2, 9 and 10 are each repeated twice, while rows 4, 5, 6 and 8 are excluded. If you run this command again, you will get a different sample of rows each time. On average, around 37% of the rows will be left out of the bootstrapped sample. With these repeated and omitted rows, each decision tree grown with bagging would evolve slightly differently. If you have very strong features such as gender in our example though, that variable will probably still dominate the first decision in most of your trees.

The second source of randomness gets past this limitation though. Instead of looking at the entire pool of available variables, Random Forests take only a subset of them, typically the square root of the number available. In our case we have 10 variables, so using a subset of three variables would be reasonable. The selection of available variables is changed for each and every node in the decision trees. This way, many of the trees won't even have the gender variable available at the first split, and might not even see it until several nodes deep.

Through these two sources of randomness, the ensemble contains a collection of totally unique trees which all make their classifications differently. As with our simple example, each tree is called to make a classification for a given passenger, the votes are tallied (with perhaps many hundreds, or thousands of trees) and the majority decision is chosen. Since each tree is grown out fully, they each overfit, but in different ways. Thus the mistakes one makes will be averaged out over them all.

R's Random Forest algorithm has a few restrictions that we did not have with our decision trees. The big one has been the elephant in the room until now, we have to clean up the missing values in our dataset.

`rpart` has a great advantage in that it can use surrogate variables when it encounters an NA value. In our dataset there are a lot of age values missing. If any of our decision trees split on age, the tree would search for another variable that split in a similar way to age, and use them instead. Random Forests cannot do this, so we need to find a way to manually replace these values.

A method we implicitly used in part 2 when we defined the adult/child age buckets was to assume that all missing values were the mean or median of the remaining data. Since then we've learned a lot of new skills though, so let's use a decision tree to fill in those values instead. Let's pick up where we left off last lesson, and take a look at the combined dataframe's age variable to see what we're up against:

```
> summary(combi$Age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   0.17   21.00   28.00   29.88   39.00   80.00     263
```

263 values out of 1309 were missing this whole time, that's a whopping 20%! A few new pieces of syntax to use. Instead of subsetting by boolean logic, we can use the R function `is.na()`, and it's reciprocal `!is.na()` (the bang symbol represents 'not'). This subsets on whether a value is missing or not. We now also want to use the `method="anova"` version of our decision tree, as we are not trying to predict a category any more, but a continuous variable. So let's grow a tree on the subset of the data with the age values available, and then replace those that are missing:

```
Agefit <- rpart(Age ~ Pclass + Sex + SibSp + Parch + Fare + Embarked +
Title + FamilySize,
                data=combi[!is.na(combi$Age),], method="anova")
combi$Age[is.na(combi$Age)] <- predict(Agefit, combi[is.na(combi$Age),])
```

I left off the family size and family IDs here as I didn't think they'd have much impact on predicting age. You can go ahead and inspect the summary again, all those NA values are gone.

Let's take a look at the summary of the entire dataset now to see if there are any other problem variables that we hadn't noticed before:

```
summary(combi)
```

Two jump out as a problem, though no where near as bad as Age, Embarked and Fare both are lacking values in two different ways.

```
> summary(combi$Embarked)
      C   Q   S
  2 270 123 914
```

Embarked has a blank for two passengers. While a blank wouldn't be a problem for our model like an NA would be, since we're cleaning anyhow, let's get rid of it. Because it's so few observations and such a large majority boarded in Southampton, let's just replace those two with 'S'. First we need to find out who they are though! We can use `which` for this:

```
> which(combi$Embarked == '')
[1]  62 830
```

This gives us the indexes of the blank fields. Then we simply replace those two, and encode it as a factor:

```
combi$Embarked[c(62,830)] = "S"
combi$Embarked <- factor(combi$Embarked)
```

The other naughty variable was Fare, let's take a look:

```
> summary(combi$Fare)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  0.000   7.896  14.450  33.300  31.280 512.300       1
```

It's only one passenger with a NA, so let's find out which one it is and replace it with the median fare:

```
> which(is.na(combi$Fare))
[1] 1044
> combi$Fare[1044] <- median(combi$Fare, na.rm=TRUE)
```

Okay. Our dataframe is now cleared of NAs. Now on to restriction number two: Random Forests in R can only digest factors with up to 32 levels. Our FamilyID variable had almost double that. We could take two paths forward here, either change these levels to their underlying integers (using the `unclass()` function) and having the tree treat them as continuous variables, or manually reduce the number of levels to keep it under the threshold.

Let’s take the second approach. To do this we’ll copy the FamilyID column to a new variable, FamilyID2, and then convert it from a factor back into a character string with `as.character()`. We can then increase our cut-off to be a “Small” family from 2 to 3 people. Then we just convert it back to a factor and we’re done:

```
combi$FamilyID2 <- combi$FamilyID
combi$FamilyID2 <- as.character(combi$FamilyID2)
combi$FamilyID2[combi$FamilySize <= 3] <- 'Small'
combi$FamilyID2 <- factor(combi$FamilyID2)
```

Okay, we’re down to 22 levels so we’re good to split the test and train sets back up as we did last lesson and grow a Random Forest. Install and load the package `randomForest`:

```
install.packages('randomForest')
library(randomForest)
```

Because the process has the two sources of randomness that we discussed earlier, it is a good idea to set the random seed in R before you begin. This makes your results reproducible next time you load the code up, otherwise you can get different classifications for each run.

```
set.seed(415)
```

The number inside isn’t important, you just need to ensure you use the same seed number each time so that the same random numbers are generated inside the Random Forest function.

Now we’re ready to run our model. The syntax is similar to decision trees, but there’s a few extra options.

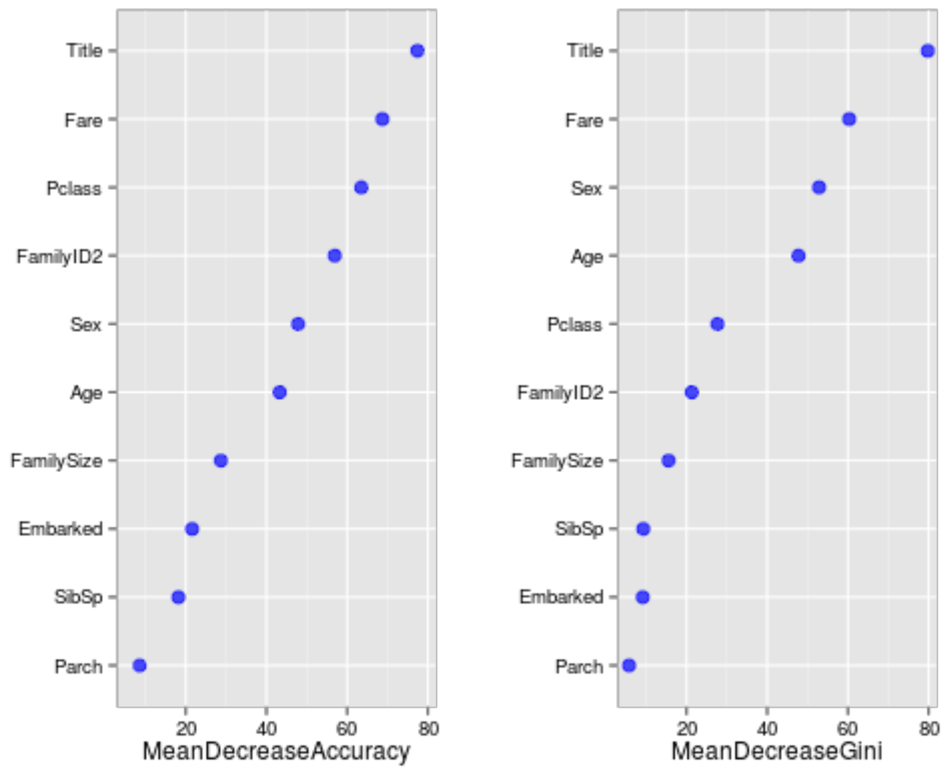
```
fit <- randomForest(as.factor(Survived) ~ Pclass + Sex + Age + SibSp +
Parch + Fare + Embarked + Title + FamilySize +
                    FamilyID2, data=train, importance=TRUE, ntree=2000)
```

Instead of specifying `method=”class”` as with `rpart`, we force the model to predict our classification by temporarily changing our target variable to a factor with only two levels using `as.factor()`. The `importance=TRUE` argument allows us to inspect variable importance as we’ll see, and the `ntree` argument specifies how many trees we want to grow.

If you were working with a larger dataset you may want to reduce the number of trees, at least for initial exploration, or restrict the complexity of each tree using `nodesize` as well as reduce the number of rows sampled with `sampsize`. You can also override the default number of variables to choose from with `mtry`, but the default is the square root of the total number available and that should work just fine. Since we only have a small dataset to play with, we can grow a large number of trees and not worry too much about their complexity, it will still run pretty fast.

So let’s look at what variables were important:

```
varImpPlot(fit)
```



Remember with bagging how roughly 37% of our rows would be left out? Well Random Forests doesn’t just waste those “out-of-bag” (OOB) observations, it uses them to see how well each tree performs on unseen data. It’s almost like a bonus test set to determine your model’s performance on the fly.

There’s two types of importance measures shown above. The accuracy one tests to see how worse the model performs without each variable, so a high decrease in accuracy would be expected for very predictive variables. The Gini one digs into the mathematics behind decision trees, but essentially measures how pure the nodes are at the end of the tree. Again it tests to see the result if each variable is taken out and a high score means the variable was important.

Unsurprisingly, our Title variable was at the top for both measures. We should be pretty happy to see that the remaining engineered variables are doing quite nicely too. Anyhow, enough delay, let's see how it did!

The prediction function works similarly to decision trees, and we can build our submission file in exactly the same way. It will take a bit longer though, as all 2000 trees need to make their classifications and then discuss who's right:

```
Prediction <- predict(fit, test)
submit <- data.frame(PassengerId = test$PassengerId, Survived = Prediction)
write.csv(submit, file = "firstforest.csv", row.names = FALSE)
```

It's relatively poor performance does go to show that on smaller datasets, sometimes a fancier model won't beat a simple one.

But let's not give up yet. There's more than one ensemble model. Let's try a forest of conditional inference trees. They make their decisions in slightly different ways, using a statistical test rather than a purity measure, but the basic construction of each tree is fairly similar.

So go ahead and install and load the `party` package.

```
install.packages('party')
library(party)
```

We again set the seed for consistent results and build a model in a similar way to our Random Forest:

```
set.seed(415)
fit <- cforest(as.factor(Survived) ~ Pclass + Sex + Age + SibSp + Parch +
Fare + Embarked + Title + FamilySize + FamilyID,
                 data = train, controls=cforest_unbiased(ntree=2000, mtry=3))
```

Conditional inference trees are able to handle factors with more levels than Random Forests can, so let's go back to out original version of FamilyID. You may have also noticed a few new arguments. Now we have to specify the number of trees inside a more complicated command, as arguments are passed to `cforest` differently. We also have to manually set the number of variables to sample at each node as the default of 5 is pretty high for our dataset. Okay, let's make another prediction:

```
Prediction <- predict(fit, test, OOB=TRUE, type = "response")
```

The prediction function requires a few extra nudges for conditional inference forests as you see.

There may be a few more insights to wring from this dataset yet though. We never did look at the ticket or cabin numbers, so take a crack at extracting some insights from them to see if any more gains are possible. Maybe extracting the cabin letter (deck) or number (location) and extrapolating to the rest of the passengers' family if they're missing might be worth a try?