Graphs Ahoy! Exploring R with the Titanic Dataset

In this notebook we're going to explore the Titanic dataset. This dataset, although small, is quite popular for teaching the fundamentals of R. The dataset consists of 887 observations (that's *rows*, in database parlance) of 8 variables or *features*, (*columns* as they would be known in a database)

This notebook assumes some familiarity with a programming language (though not necessarily *R*). I'm going to try to avoid getting bogged down in the minutiae of syntax and mechanics, and am relying on the fact that *R* is an inherently readable programming language. Hopefully, even where constructs are unfamiliar, the meaning and intention will be clear from the context.

Overview

In this exercise we will

1. Download the dataset and parse it so we can begin manipulating it in R
2. Perform some basic data cleaning
3. Explore the data using visualisations

This notebook is intended to act as a guide to the kind of things you can do with R. We will be revisiting each of these areas as the semester goes on and learning about them in more detail. You are not expected to be able to dissect the code at this stage, but you will be able to do so by the end of the semester!

Preliminaries

First thing's first. We're going to be using the packages *ggplot2* and *lattice* later in this notebook. It's a good idea to put all of your library() commands at the top of a script. This makes it easy for others to see what's needed in order to make the script run and download the required packages.

The first two lines check if the required packages exist and if not, installs them. You don't need to do this when you're writing code for your own machine, but it's good practice if you're sharing your code with others who might not have the required packages

```
# require('packageName') returns TRUE if the package has been downloaded on the
# user's machine and FALSE otherwise. The '!' operator is read as "Not". Essentially,
# here we're saying "if the require() function returns false, install the package"
if (!require('ggplot2')) install.packages('ggplot2')
```

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

```
if (!require('lattice')) install.packages('lattice')
```

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

```
# Even though they've been installed we still need to tell R to load them into memory
library(ggplot2)
library(lattice)
```

Now that we've got our modules loaded we next want to load our data. We will be using the Titanic Dataset. As this data is in comma separated values format (*csv*) we can parse it using *R*'s read.csv() function. Read.csv takes a file location as its first parameter. This can be a path to a file on your computer, but can just as easily be a web address.

```
data <- read.csv(
  'https://web.stanford.edu/class/archive/cs/cs109/cs109.1166/stuff/titanic.csv'
  )
# The line below forces R to output the dataset we just created
head(data)
```

```
##   Survived Pclass                                                Name
## 1        0      3                             Mr. Owen Harris Braund
## 2        1      1 Mrs. John Bradley (Florence Briggs Thayer) Cumings
## 3        1      3                              Miss. Laina Heikkinen
## 4        1      1        Mrs. Jacques Heath (Lily May Peel) Futrelle
## 5        0      3                            Mr. William Henry Allen
## 6        0      3                                    Mr. James Moran
##      Sex Age Siblings.Spouses.Aboard Parents.Children.Aboard    Fare
## 1   male  22                       1                       0  7.2500
## 2 female  38                       1                       0 71.2833
## 3 female  26                       0                       0  7.9250
## 4 female  35                       1                       0 53.1000
## 5   male  35                       0                       0  8.0500
## 6   male  27                       0                       0  8.4583
```

Notice that we now have a variable called `data` in the top right panel of our IDE. If we click the dropdown arrow to the left of the variable name we can see each of the features belonging to this dataset. We can access those features using the `$` operator; e.g. to get only the ages of passengers we can use data$Age

Cleaning the Data

We'll start by asking *R* to give us a summary of our overall dataset. This will go through each of the features (columns) of our data one-by-one and give us headline summary statistics on each.

```
summary(data)
```

```
##     Survived          Pclass                                      Name
##  Min.   :0.0000   Min.   :1.000   Capt. Edward Gifford Crosby       :  1
##  1st Qu.:0.0000   1st Qu.:2.000   Col. John Weir                    :  1
##  Median :0.0000   Median :3.000   Col. Oberst Alfons Simonius-Blumer:  1
##  Mean   :0.3856   Mean   :2.306   Don. Manuel E Uruchurtu           :  1
##  3rd Qu.:1.0000   3rd Qu.:3.000   Dr. Alfred Pain                   :  1
##  Max.   :1.0000   Max.   :3.000   Dr. Alice (Farnham) Leader        :  1
##                                   (Other)                           :881
##      Sex           Age        Siblings.Spouses.Aboard
##  female:314   Min.   : 0.42   Min.   :0.0000
##  male  :573   1st Qu.:20.25   1st Qu.:0.0000
##               Median :28.00   Median :0.0000
##               Mean   :29.47   Mean   :0.5254
##               3rd Qu.:38.00   3rd Qu.:1.0000
##               Max.   :80.00   Max.   :8.0000
##
##  Parents.Children.Aboard      Fare
##  Min.   :0.0000          Min.   :  0.000
##  1st Qu.:0.0000          1st Qu.:  7.925
##  Median :0.0000          Median : 14.454
##  Mean   :0.3833          Mean   : 32.305
##  3rd Qu.:0.0000          3rd Qu.: 31.137
```

```
##  Max.   :6.0000          Max.   :512.329
##
```

Performing a summary on the entire dataset is always a good place to start when cleaning data. It gives us a nice general overview of the dataset, and lets us do a quick sanity check to make sure *R* has interpolated the data types correctly. We're going to work our way through each of the features and make sure that what we're seeing makes sense.

Essentially, we're checking that *R* has inferred the correct **data types** for each of our features. By making sure each feature is of the correct type we make our lives easier, because when it comes to creating graphs and models, *R* will be able to make intelligent guesses about what we want it to do. Some graphs and models can be picky about the types of data it will work with, too.

Logical Features

R is giving us strange statistics for the `Survived` feature. It's telling us that the mean value for `Survived` is 0.3856. This doesn't really make much sense. Someone either survived or they didn't.

The reason this is happening is that The number `0` is used to represent a passenger who died and `1` is used to denote a passenger who survived. *R* has interpreted these values as numbers, even though we know they're really just a way of encoding the values `TRUE` or `FALSE`. We'll convert them to `logical` data-types (true/false) to make it easier to work with

```
data$Survived.logical <- as.logical(data$Survived)
```

What's going on above? We generally read functions in *R* from right to left, as that's usually the order they're processed in. We can see above that we're taking the `Survived` column of our dataset, converting it to a *Logical* datatype, and we're assigning the result to a new column in our dataset which we're calling `Survived.logical`. There was no feature called `Survived.logical` before, so *R* will automatically create it for us.

Usually, though we don't want to keep old versions of our feature data lying around, and it's often cleaner and easier to just overwrite the original value with the value we want.

```
data$Survived <- as.logical(data$Survived)

# Let's tidy up after ourselves and drop the Survived.logical feature we created
# earlier. R has a handy trick to do this. We assign the value list(NULL) to a
# column and R will delete it for us. Note that we're using square bracket notation
# (rather than the usual $ operator) and passing the column name in as a string. We'll
# talk more about this in # later classes but for now, just know that these are
# by-and-large equivalent.
# The list(NULL) trick only works with square bracket notation, however.
data["Survived.logical"] <- list(NULL)
```

Character Data

Character data is the *R* equivalent to what most programmer's would think of as *strings*. Character data is a collection of human readable characters, with no special meaning.

```
# By passing in maxsum=5 I'm telling R to show me no more than 5 unique values.
summary(data$Name, maxsum=5)
```

```
##        Capt. Edward Gifford Crosby                      Col. John Weir
##                                  1                                   1
## Col. Oberst Alfons Simonius-Blumer            Don. Manuel E Uruchurtu
##                                  1                                   1
##                            (Other)
```

```
##                                883
```

When we ask R to give us a summary of the `Name` feature we see a list of all of the unique passenger names, and the number of times each appears in the dataset. R is treating the `Name` variable as if it's a category-type. This would be very handy for something like *Genre* in a movie collection, as it would allow us to see how many of each type of movie we had, but it doesn't make much sense for Names. *R* refers to this type of variable as a **factor** (you can think of it as a category). However, as far as we're concerned, the names are really just a string of letters, or in *R* parlance **character** data. Similarly to what we did above, we'll convert this data to character data and overwrite the column

```
data$Name <- as.character(data$Name)
summary(data$Name)
```

```
##    Length     Class      Mode
##       887 character character
```

That looks better. Character data usually isn't much use to us from a modelling point of view, and is mostly used for human-readable labels and the like. We can see how little interest R has in the character data from the terseness of the summary.

Factor Data

Next up we'll take a look at the `Sex` feature.

```
summary(data$Sex)
```

```
## female   male
##    314    573
```

The **factor** datatype makes much more sense for this feature. In the year 1912, it was widely held throughout Europe and the U.S. that every human could be categorized into one of two genders. `Sex`, in this dataset is used as a category, which is exactly what the **factor** datatype is meant for. The `summary` function tells us what proportion of our dataset identified as male/female. We can leave this feature as is. Next up we have the `Pclass` feature.

```
summary(data$Pclass)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   2.000   3.000   2.306   3.000   3.000
```

The Pclass feature refers to the type of ticket the passenger was holding. This can be first, second or third class. R is again treating this as numeric, even though that's not really what we want. For a start, the *Mean* (Average) class is 2.306. 2.306 class isn't a thing, and it doesn't make sense to say "multiply 1st class by 2". Even though First, Second and Third class are often denoted using numerals, they're not really numbers *per se*.

We want to tell R to treat this as a **factor** type. However, unlike the `Sex` feature, there seems to be an inherent ordering between the three classes, we know that a 1st class ticket is better (or more expensive, at least) than a 2nd class ticket, which is better (or more expensive) than a 3rd class ticket. The R `factor()` function has an option to treat the factor as ordinal (*i.e.* a **factor** type but with some idea of an ordering between the factors). To do this, we need to specify `TRUE` for the `ordering` parameter of the function.

```
?factor
```

We can look at the manual for the factor function by typing a question mark into the console followed by the function name. We can see from the output that the `factor()` function allows us to specify both the labels (human-readable text) and the levels (numbers reprenting the ordering between items). Notice that when we want to pass multiple values for `levels` and `labels` we use the `c()` (*combine*) function to combine the values into a single list. By specifying an ordering between the different categories (or to speak in R's language, **levels**) we allow R to be more intelligent when displaying graphs *etc.*

```
data$Pclass <- factor(data$Pclass,
                      order=TRUE,
                      levels=c(1,2,3),
                      labels=c("1st Class", "2nd Class", "3rd Class")
                      )
```

Integer (and Numeric) Data

Next we'll perform a summary on the `Age` feature.

```
summary(data$Age)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.42   20.25   28.00   29.47   38.00   80.00
```

Age has been recognised as a number. Notice, however, that the youngest passenger is 0.42 years old. This is strange. And it looks like the majority of ages are simply specified to the year. Let's take a closer look.

We can use the modulus (%%) operator to to remove everything to the left of the decimal point The statement `12 %% 5` instructs R to divide 12 by 5 and return only the remainder. By taking Modulo 1 for every age in the dataset we get the remainder of the number when divided by 1, which leaves us only the decimal places.

We're interested in the number of distinct values we have with a decimal place. A little hacky trickery can make this a lot easier for us. If we convert the `Age` feature to a **factor**, then *R* will keep track of each unique value for us. If we summarize the data *as a factor* we'll get a breakdown of each of the distinct values. Notice that even though we're converting `data$Age` to a factor we're not assigning the result back to our table so our data is left unchanged

```
summary(as.factor(data$Age %% 1))
```

```
##    0 0.42  0.5 0.67 0.75 0.83 0.92
##  862    1   18    1    2    2    1
```

We can see 25 ages have been specified to at least 1 decimal place and 862 have not. This looks like inconsistency in data collection. We can simply remove the decimal place to fix this, effectively rounding everything down. (Why might this be better than straight rounding)? By converting the number to an integer we'll simply ignore all decimal places

```
data$Age <- as.integer(data$Age)
```

Visualising the Data

Now that we've cleaned our data, and made sure everything is of the correct type we can start visualising our data, and hopefully begin to gain some insight into what the data has to tell us.Let's start by taking a look at how survival rate varied with Age. The ggplot2 package is an extremely useful package which makes it easy to create nice graphs.

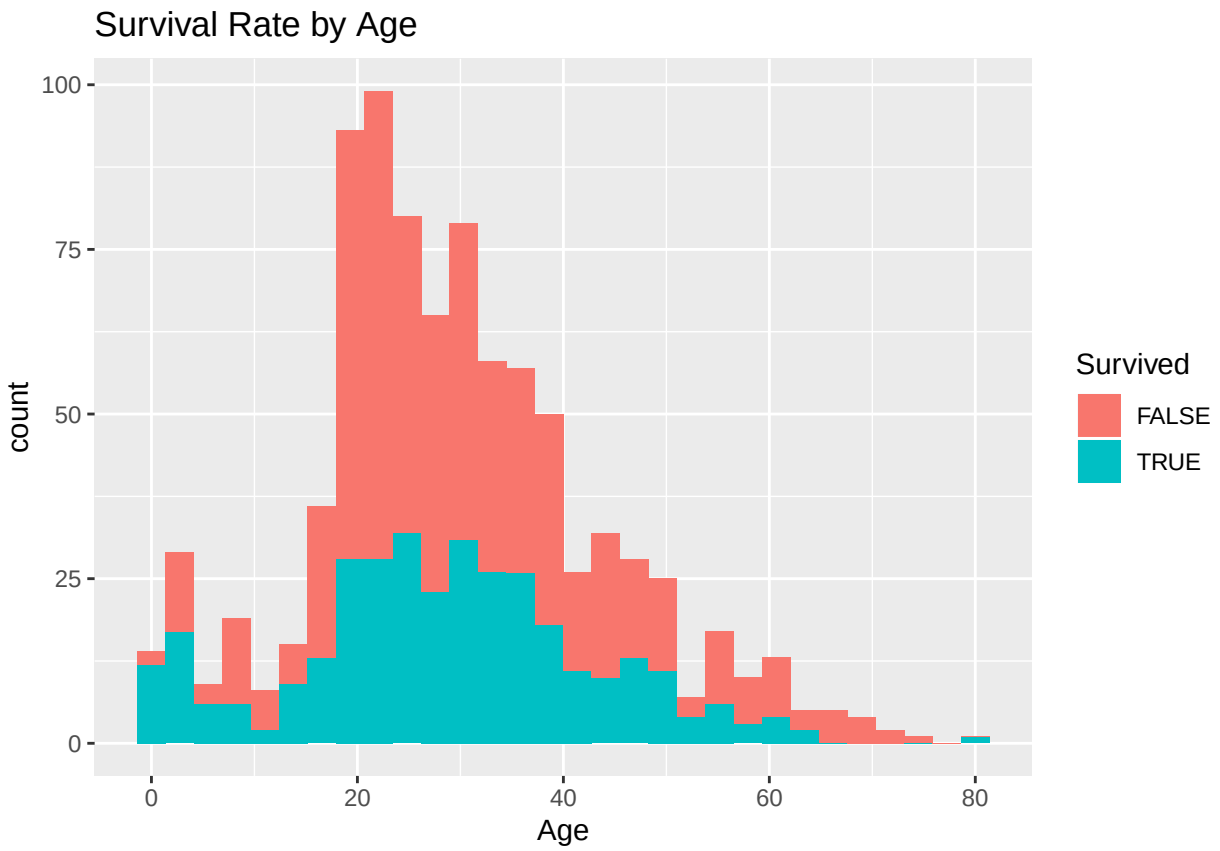
Histograms

A histogram is a bar chart for continuous (numeric) variables. A histogram takes a numeric variable on the x axis, groups similar values together into *bins*, and for each bin draws a bar where the height of the bar represents the number of values in that bin.

In the example below, we're telling ggplot to put Age on the X axis, andrepresent whether or not a passenger survived by colour (fill). We're then specifying that we want it to create a histogram for us, and finally, we're using the `labs()` (labels) function to specify a title for the graph.

```
ggplot(data, aes(x=Age, fill=Survived)) +
  geom_histogram() +
  labs(title="Survival Rate by Age")
```

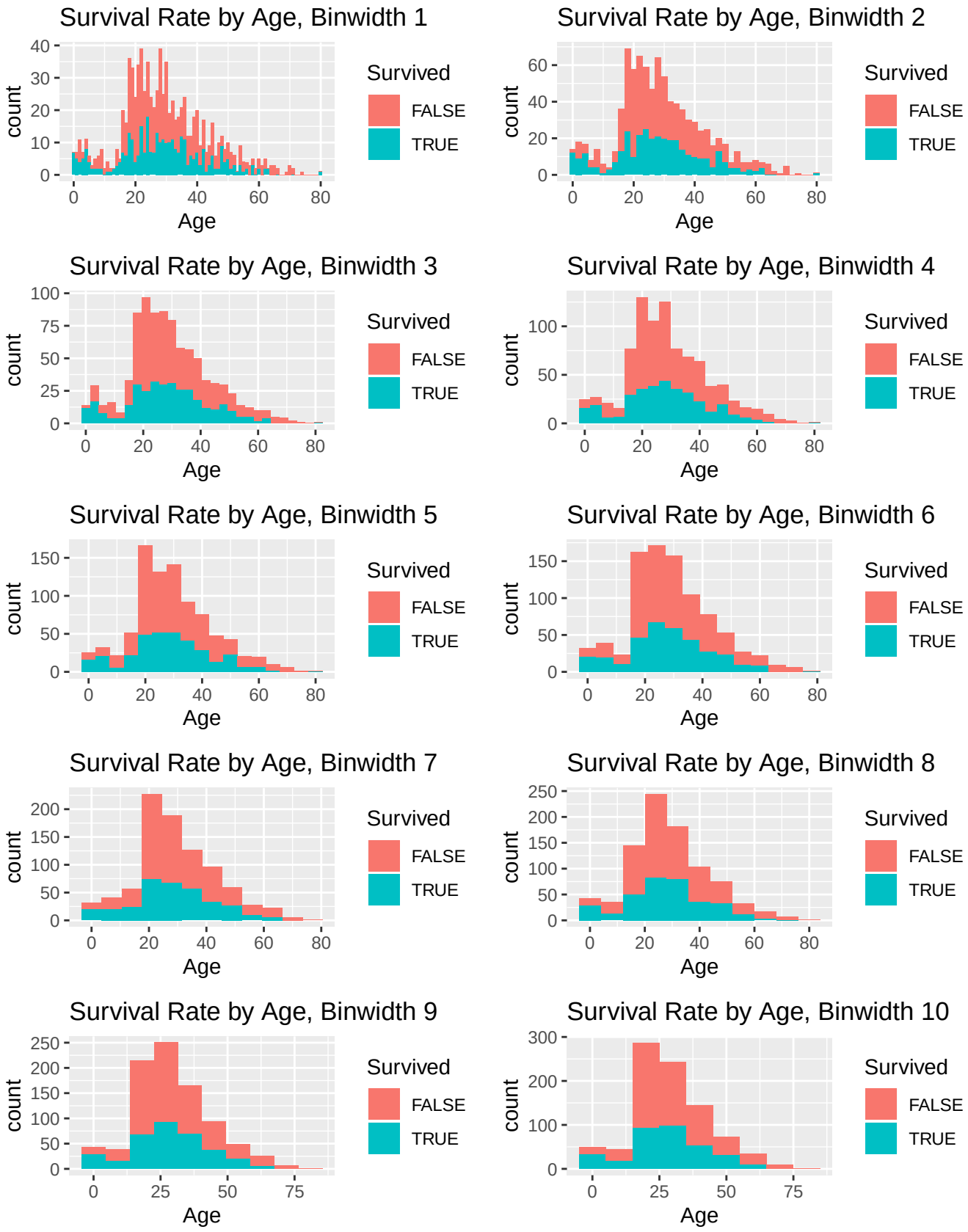
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Notice the warning message from R in the console. We didn't tell R how wide we want our bins to be. so it's defaulted to using 30 bins. Let's experiment with this a little and see if we can find a more useful bin-width.

We could do this manually, by changing the value for binwidth and eyeballing the plot it creates. However, we can automate all of this using a **for** loop.

```
# This for loop is telling R to create this plot with different binwidths.
# 1:10 is a shorthand way of writing c(1,2,3,4,5,6,7,8,9,10)
for (binwidth in 1:10) {
  # Because we're inside a for loop, we now need to explicitly tell ggplot
  # to draw our plot by using the print() function
  print(ggplot(data, aes(x=Age, fill=Survived)) +
    geom_histogram(binwidth=binwidth) +
    labs(title=paste("Survival Rate by Age, Binwidth", binwidth)))
}
```


Survival Rate by Age, Binwidth 1
Survival Rate by Age, Binwidth 2
Survival Rate by Age, Binwidth 3
Survival Rate by Age, Binwidth 4
Survival Rate by Age, Binwidth 5
Survival Rate by Age, Binwidth 6
Survival Rate by Age, Binwidth 7
Survival Rate by Age, Binwidth 8
Survival Rate by Age, Binwidth 9
Survival Rate by Age, Binwidth 10
count
Age
Survived
FALSE
TRUE

Narrower bins give more precise visualizations but if they're too narrow they just add noise to the graph. Pick a bin-width that shows an interesting trend without being too noisy (random ups and downs)

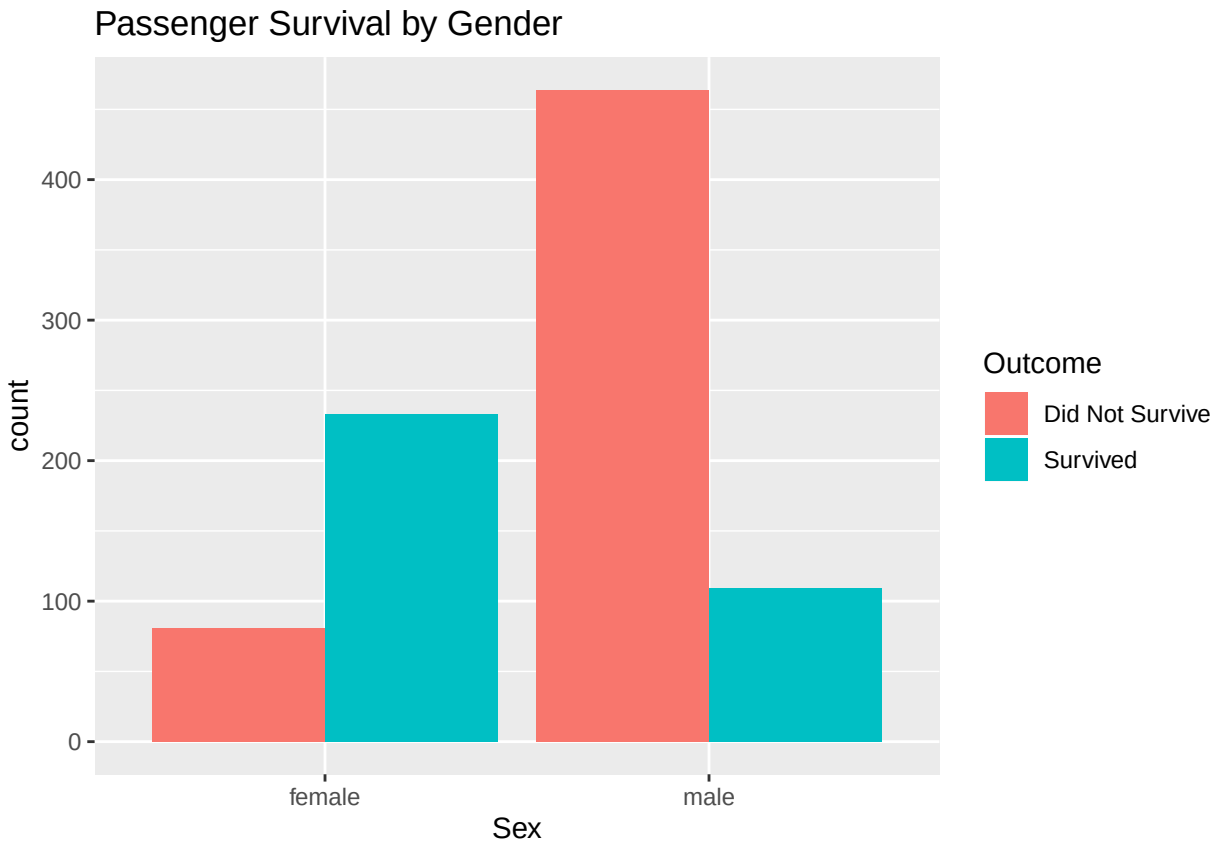
Bar Charts

Bar charts a good way to visualize counts of data broken down by category. Here we'll use a bar chart to get an idea of how the `Sex` feature interacts with the `Survived` feature.

Anyone who has watched James Cameron's Titanic will know that 'women and children' were given preference for lifeboats. Let's see if the data backs this up. While we're at it, let's make our legend a little more readable. Rather than saying Survived (True/False) we'll have more human-readable names.

The `scale_fill_discrete` function allows us to control the way fill is used within the graph. By passing explicit values for labels we can change the legend text. Also, by adding `position="dodge"` to the `geom_bar` function we tell R to make the bars sit beside (rather than on top of) one another.

```
ggplot(data, aes(x=Sex, fill=Survived)) +
  geom_bar(position="dodge") +
  scale_fill_discrete(name = "Outcome", labels = c("Did Not Survive", "Survived")) +
  labs(title="Passenger Survival by Gender")
```



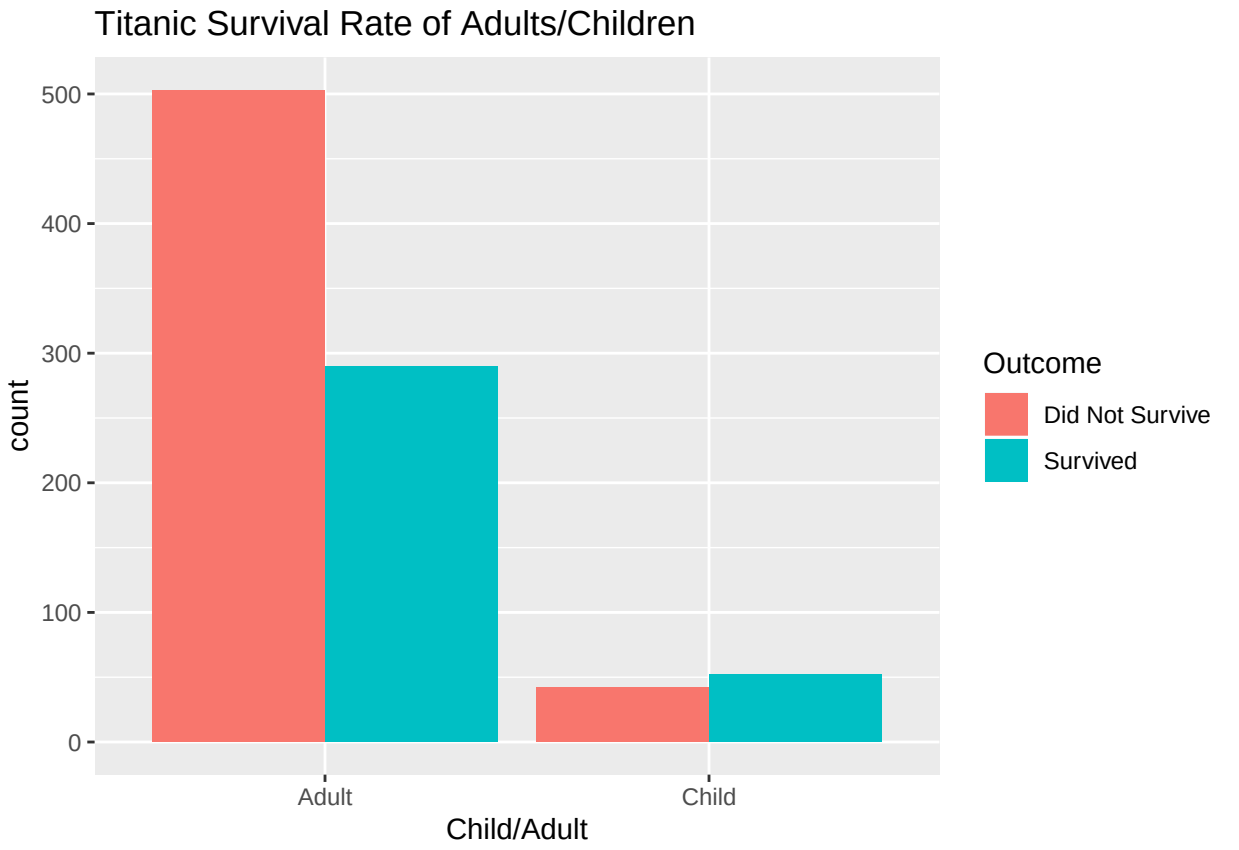
Aside: Computed Features

We've already looked at age, but bearing in mind that children were given special status, it might make sense to categorize our passengers into children and adults. By using the dollar operator we're creating a new feature on our dataset called `IsChild`. `IsChild` will be true if `Age` is less than 16, false otherwise. I just picked 16 as a value I thought appropriate, but any value you can justify would work here.

Let's create this new categorical variable and then plot the data using a bar chart

```
data$IsChild <- data$Age < 16

ggplot(data, aes(x=IsChild, fill=Survived)) +
  geom_bar(position="dodge") +
  scale_x_discrete(name="Child/Adult", labels=c("Adult", "Child")) +
  scale_fill_discrete(name = "Outcome", labels = c("Did Not Survive", "Survived")) +
  labs(title="Titanic Survival Rate of Adults/Children")
```



From this we can see that children were much more likely to survive the wreck than adults. It would be possible to infer this from the histogram above, but I believe this bar chart makes the point more forcefully, and given our extra domain knowledge we can justify the decision to add this extra category.

Faceting

So far we've only been looking at one variable at a time and seeing how that variable influenced survival. However, interactions between different variables can have much larger effects. One good way of visualising this is through the use of facets. Facets allow you to combine multiple graphs filtered on one or more categories, or **factors**.

We want to see how Age, Passenger Class and Gender interacted to influence survival. We start by defining a single histogram plot which shows how the survival rate varied with age. We're going to repeat this plot for each combination of class and gender. The facet_grid function allows us to specify which variables we'd like to split our data on.

```
# Make sure you've loaded the `lattice` module
# We did this with our library(lattice) at the top of the script
```

```
ggplot(data, aes(x=Age, fill=Survived)) +
  geom_histogram(binwidth = 5) +
  facet_grid(Sex ~ Pclass) +
  ggtitle("Titanic Survival by Age, Gender and Class")
```

The tilde (~) operator in R denotes a formula. We can read the example below as 'Sex' by 'Class' Try it out and then see what happens if you swap the formula around

We can see from the plot above that females travelling in first class were almost certain to survive the wreck of the titanic, while med travelling in third class were much less likely to do so.

Conclusion

In this notebook we looked at how to download and parse a csv dataset, how to clean that dataset and make sure each feature was of the right type, and how to perform some exploratory analysis with the resulting clean data.

A more comprehensive guide, which dives into re-structuring and re-shaping the data to give us more detail for our graphs can be found on a kaggle titanic notebook. This takes even more of a deep-dive into R, but might be worth while for the more adventurous among you!