R-Tutorials Using Titanic Data

Josemari Feliciano

7/10/2017

Motivation To This Tutorials:

Arguably, R has the steepest learning curve for Data Scientists and Statisticians. Once you are comfortable with it, however, R is a very rewarding tool for Data Analysis. Learning the many ways to represent and manipulate your data in R is always the main challenge for beginners; this is specially true to those with minimal programming background.

For instance, `barplot()` is able to read data from RData seemlessly. But it is always a challenge to learn the most proper – and perhaps the simplest – method to print custom data sets.

Plenty of the intermittent and sparse tutorials out there tend to overcomplicate their code. So my personal struggles with R inspired this tutorials. In short, I already performed the coquettish struggle between simplicity and properness to perform tasks in R. Someday, I hope to publish open-source textbooks in Statistics and Data Science.

There are four major parts to this tutorials:

1. How to read and extract data from RData file
2. How to create basic tally and graphs, and customized graphs
3. How to run basic descriptive statistics
4. How to run and read results of inferential statistics

Tutorial Notes:

The Beauty of R Studio is the ability to have notebooks for data analysis. It allows us to run R-code inline similar to IPython Notebooks.

Think of it as a fancy chemistry notebook where you can somehow run the experiments itself in the notebook – but for data analysis! For this notebook and R tutorial, I will use the titanic data for analysis. This assumes that you have installed both R and RStudio.

Note : The mosaic package is required if you want to run the data yourself in your local R environment – and not to simply view them from my github repo. If you do not have this installed, run this code in your r-console independently:

```
install.packages("ggplot2")
```

R-Tutorials Using Titanic Data

Part 1: Reading data

We probably want to load the Titanic data first. We probably want to load the mosaic library out of the way as well.

Loading Data:

We can accomplish both data loading and library call with the following R-script:

```
load("Titanic.Rdata")
library("mosaic")
```

Note: Always rerun/replay the code above when entering this file in your local environment. The pre-ran scripts will remain intact but rerunning them might display errors. So you might have to rerun this code eventually. If you see an error, know that this code might be the culprit.

Which Variables Are In Data?

Now that the data has been loaded, we probably want to see which data variables we have to deal with! Below, we will use `names()` to print the variables within our data.

```
names(Titanic)
```

```
## [1] "Gender"   "Age"      "Name"     "Fare"     "Class"    "Survived"
```

The `names()` function we just ran displayed the 6 variables within the Titanic Data which include Gender, Age, Name, Fare, Class and Survived.

Of course, you could have looked at the actual CSV file or RData file directly. But functions like `names()` are very useful when wrangling data from JSON and similar data types.

Exploring Data Types:

The `sapply()` function is very useful for this to see the variables and their data types.

```
sapply(Titanic,class)
```

```
##    Gender       Age      Name      Fare     Class  Survived
##  "factor" "numeric"  "factor" "numeric"  "factor"  "factor"
```

The output above does make sense; data types seem to match what we would expect. Age is numeric. And Gender is a 'factor', most commonly known as a string in other programming languages.

Another function we could have used is `str()`, but the output can be messy and dense. `str()` does provide more information on the variables in our data.

```
str(Titanic)
```

```
## 'data.frame':    1045 obs. of  6 variables:
##  $ Gender  : Factor w/ 2 levels "Female","Male": 1 2 1 2 1 2 1 2 1 2 ...
##  $ Age     : num  29 1 2 30 25 48 63 39 53 71 ...
##  $ Name    : Factor w/ 1307 levels "Abbing, Mr. Anthony",..: 22 24 25 26 27 31 46 47 51 55 ...
##  $ Fare    : num  211 152 152 152 152 ...
##  $ Class   : Factor w/ 3 levels "Lower","Middle",..: 3 3 3 3 3 3 3 3 3 3 ...
```

```
##  $ Survived: Factor w/ 2 levels "No","Yes": 2 2 1 1 1 2 2 1 2 1 ...
##  - attr(*, "na.action")=Class 'omit'  Named int [1:264] 16 38 41 47 60 70 71 75 81 107 ...
##   .. ..- attr(*, "names")= chr [1:264] "16" "38" "41" "47" ...
```

The output from `str()` did provide us more information about our variables. For instance, the output shows that there are 2 levels for Gender: "Famale" and "Male". But `str()` can be messy to look at.

Note: We could have used `View(Titanic)` to open the spreadsheet in a difference pane or tab in RStudio. But that will require sifting through the spreadsheet. It's good to familiarize with both `str()` and `sapply()` functions so we don't have leave our tab.

R-Tutorials Using Titanic Data

Part 2: Creating Quick Tally and Basic Graphs

Basic Data Tallying and Graph Creation

There are multiple ways of tallying variables.

Using count format of `tally()`:

```
tally(~Gender, format = "count", data = Titanic)
```

```
## Gender
## Female   Male
##    388    657
```

As you can see from above output, the count format will give us the raw number of count. For instance, 388 passengers from the Titanic are female.

Using proportion format of `tally()`:

Another format that statisticians use is the proportion format of `tally()`.

```
tally(~Class, format = "proportion", data = Titanic)
```

```
## Class
##     Lower    Middle     Upper
## 0.4784689 0.2497608 0.2717703
```

According to the output above, it is implied that roughly 47.85% of Titanic passengers bought the low-class tickets.

Using percent format of `tally()`:

Most of Statistics will deal with proportion format like those displayed in the previous section. So the percent format is really unnecessary. But if you will not use any other statistical test, this might be handy to use.

```
tally(~Survived, format = "percent", data = Titanic)
```

```
## Survived
##       No      Yes
## 59.13876 40.86124
```

The output above clearly shows that roughly 59.14% of Titanic passengers did not survive the infamous tragedy.

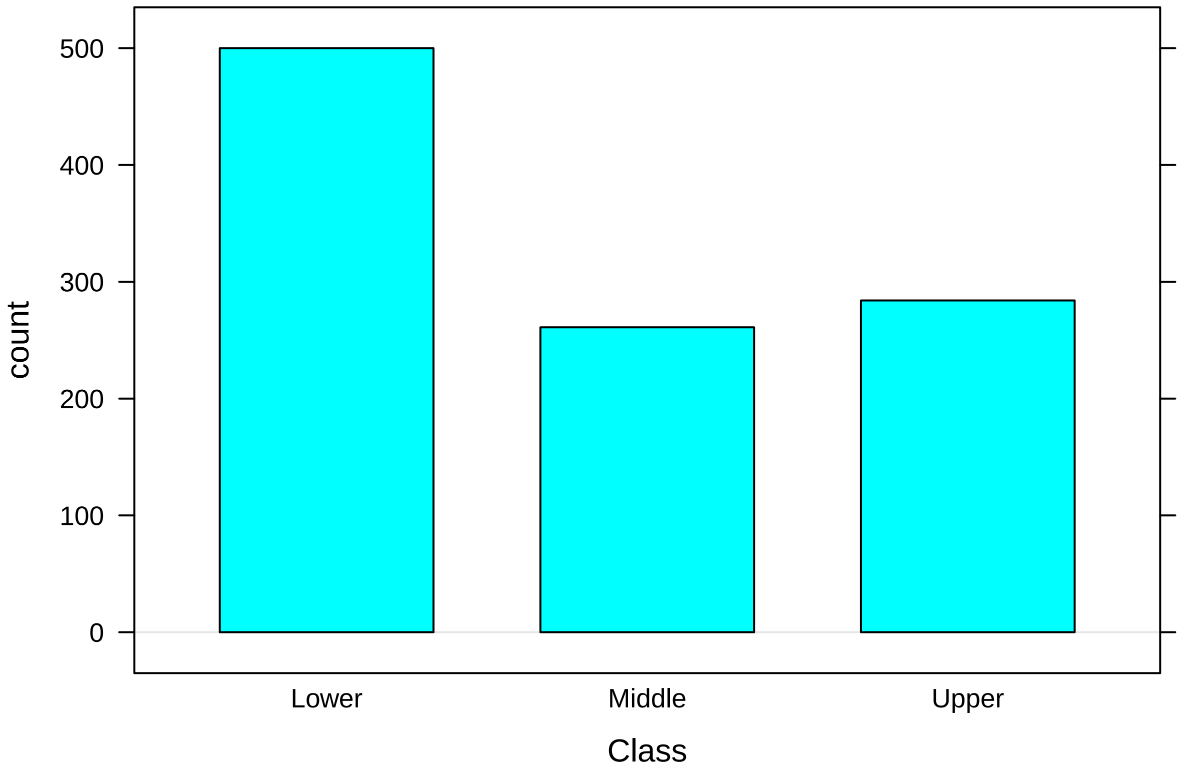
Just a quick note: Had we ran the proportion format, the output would have displayed 0.5914 for No. As students and practitioner of Statistics, you need to be fluent in reading via proportions.

Creating Graphs and Charts of One Categorical Data:

The function `bargraph()`

The easiest way to create graph in R is by using the `bargraph()` function:

```
bargraph(~Class, Titanic)
```



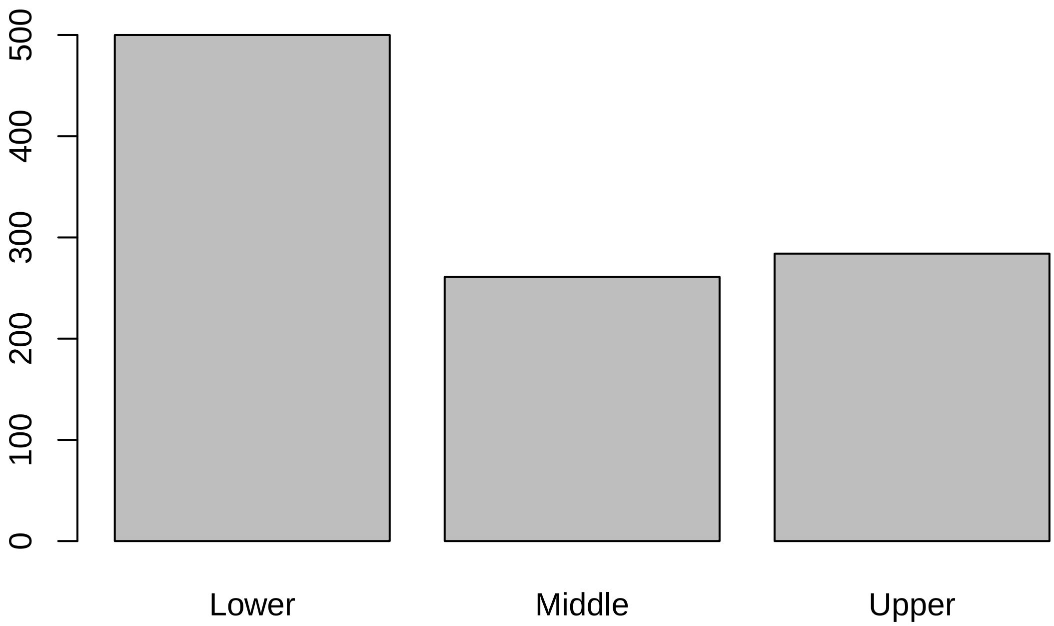
It looks like we were able to graph a decent graph. But the issue with this is: `bargraph()` lacks flexibility. So this might be good to create a simply bar graph. But if you want to creatively label or redesign any aspect of the graph, `bargraph()` will not yield to you.

The function `barplot()`

`barplot()` is one of the best functions to plot graphs. Unfortunately, a call like `barplot(~Class, Titanic)` will not work because `barplot()` only accepts arrays, contingency tables and similar data types.

So there's an extra step required though. But here is an illustration of how to call it properly:

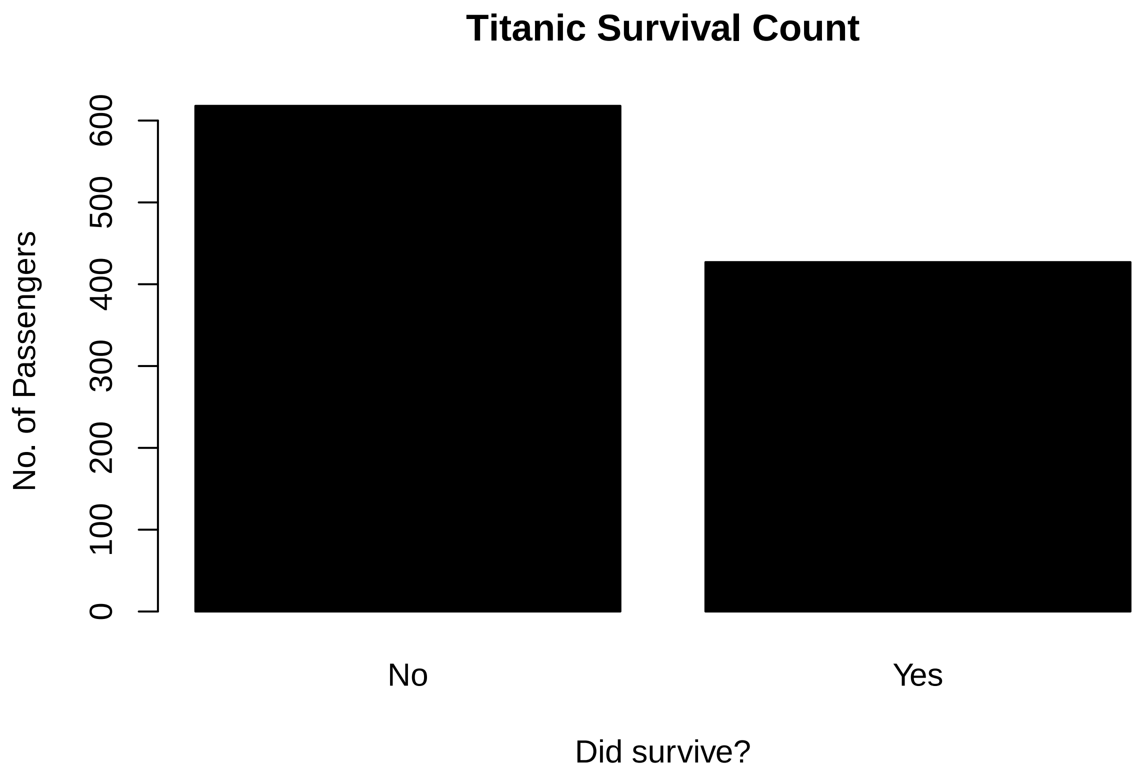
```
#here, table() is called to convert the class data
#to a contingency table that barplot() could read
#we are assigning this table into a variable called "data"
#then, we are passing it to barplot()

data <- table(Titanic$Class)
barplot(data)
```



Look how beautiful this graph is! Again, we converted the Class data from the Titanic into a contingency table using `table()`.

Beautify the graph: The graph is very bare. It lacks labels, titles, and color. Let us pass some paramaters to modify its properties. Below, I will create a graph for Survival.

```r
#obviously, we are passing data to xlab, ylab, main, color to add properties to our barplot
data2 <- table(Titanic$Survived)
barplot(data2, xlab = "Did survive?", ylab = "No. of Passengers",
        main = "Titanic Survival Count", col = "black")
```



Look how much better it looks! It looks much better with our custom color, labels, titles. There's plenty more of modification we could do (such as change the border color of graphs). But for this, check out the R-documentation page and see which parameters and default values we could change to fit our goals.

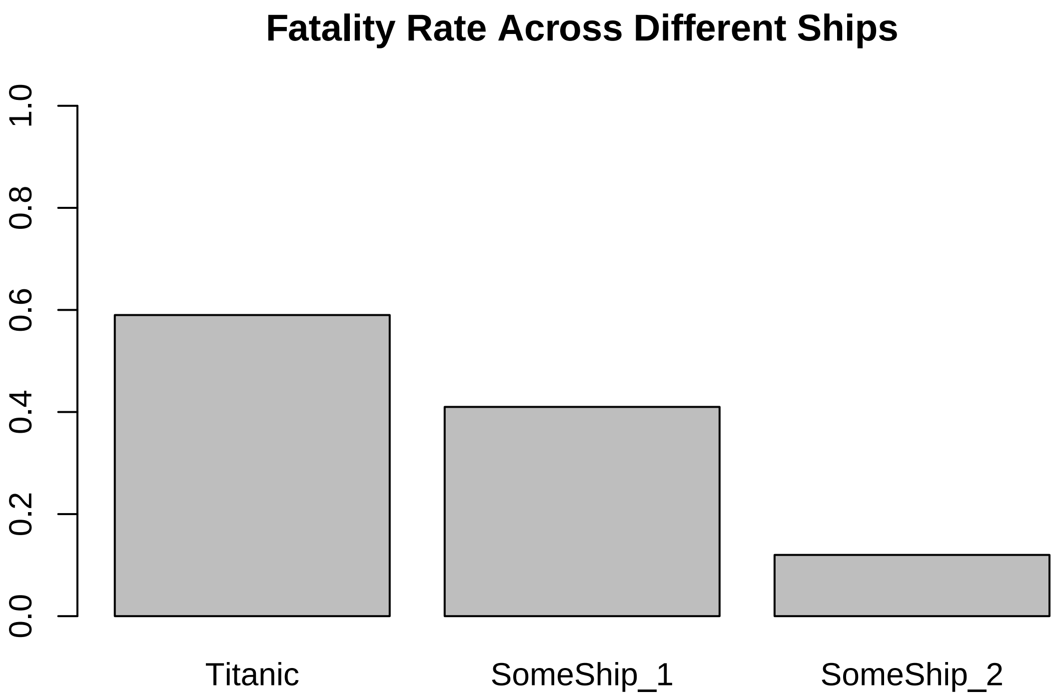
Graph From Customized Data Through `barplot()`

For complicated big data, we simply extract data and plot them. But for simple small data, we could simply pass an array of values for plotting.

This particular tutorial is helpful: Most tutorials will have you use names.arg inside the `barplot()` and other methodology/sythax that can be a nightmare. `names()` is your friend here as I will demonstrate below.

Suppose we found out from the web that Titanic has a fatality rate of .59 and want to compare it to the fatality rate of two other ships with rates 0.41 and 0.12 respectively. We can convert all the dependent and independent variables into their own arrays/collection.

```
fatality_rate <- c(.59,.41,.12)
names(fatality_rate) <- c("Titanic","SomeShip_1", "SomeShip_2")


barplot(fatality_rate, main="Fatality Rate Across Different Ships",
        ylim=c(0,1))
```



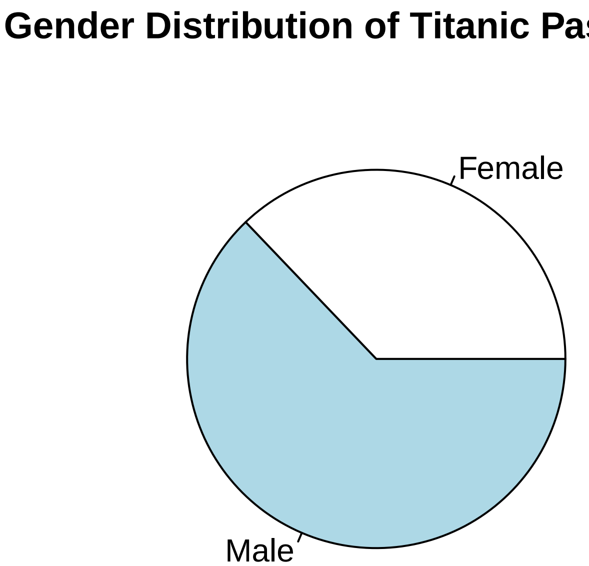
Voila, we were able to create custom bar graphs with ease by creating a collection and by utilizing `names()`

Important point about scaling: Do note that I also passed a `ylim=c(0,1)` argument inside `barplot()`. Often, you will find that graphs of any kind can look funky with scaling really off. In this case, we know that proportions only go between 0 and 1. So I passed a `ylim=c(0,1)` to force this y-window.

Pie Chart Through `pie()`

Now we are done with bar graphs – both basic and advanced – let us go over pie charts. `pie()` is very useful for this task. Below is the code to print this:

```
counts <-tally(~Gender,  data = Titanic)
pie(counts, main="Gender Distribution of Titanic Passengers")
```

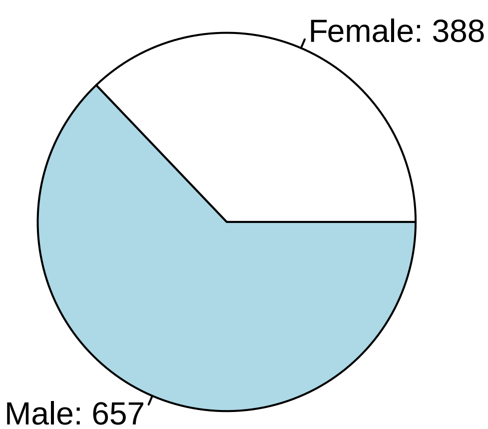


`pie()` helped us create a beautiful pie chart. But I would argue that `pie()` lacks flexibility and is one of the most frustrating functions to deal with.

Issue: The pie chart was a good visual but it did not print the numbers associated with them. There's actually no easy way to do this. And sadly, many outside resources will offer you confusing ways of doing this without proper explanation. Here what I am here for!

Basics: Custom labels have to be a "factor" – as explained earlier, "factor" is equivalent to string in other programming languages. So we want to create a label with both the categorical name and number count. `paste()` is our friend here since we can pass arguments that it can join together. See the script I ran and I will explain the logic behind them.

```
gender_data <- table(Titanic$Gender)
data_labels <- paste(names(gender_data),gender_data,sep=": ")
pie(gender_data,labels=data_labels)
```



As expected, the pie chart finally printed both the categorical variables and their number count!

The script that I made for customization can be overwhelming.

So here's what I did and their respective explanation:

1. First, I created a table of gender data and called it gender_data.
2. Next, I created a factor to act as my label. Here, `paste(names(gender_data),gender_data,sep=": ")` implies that I am joining the `names` of my gender_data (which is either Male or Female) and their respective count. `sep` – short-hand for separator – indicated how I wanted them to be seperated. Here, I wanted them to be separated by a colon and a space: `sep=": "`. I passed this entire function or argument to a new variable, `data_labels`, to carry this custom label we created.

3. Finally, I created the pie chart by calling `pie(gender_data,labels=data_labels)`.

Now, we are done creating graphs with just one variable. Now, let's focus on creating tables and graphs for two different data variables!

Creating Contingency Table for Two Variables

The code is similar to `tally()` we used earlier but instead, we are passing two variables. For this example, I want to create a contingency table between class and survival.

```
tally(~Class + Survived, format = "count", data = Titanic, margins=TRUE)
```

```
##         Survived
## Class      No  Yes Total
##   Lower   369  131   500
##   Middle  146  115   261
##   Upper   103  181   284
##   Total   618  427  1045
```

Like we did with one variables, we can indicate to tally to print the data in proportion format. For this, simply pass `format = "proportion"` inside `tally()`.

```
tally(~Class + Survived, format = "proportion", data = Titanic, margins=TRUE)
```

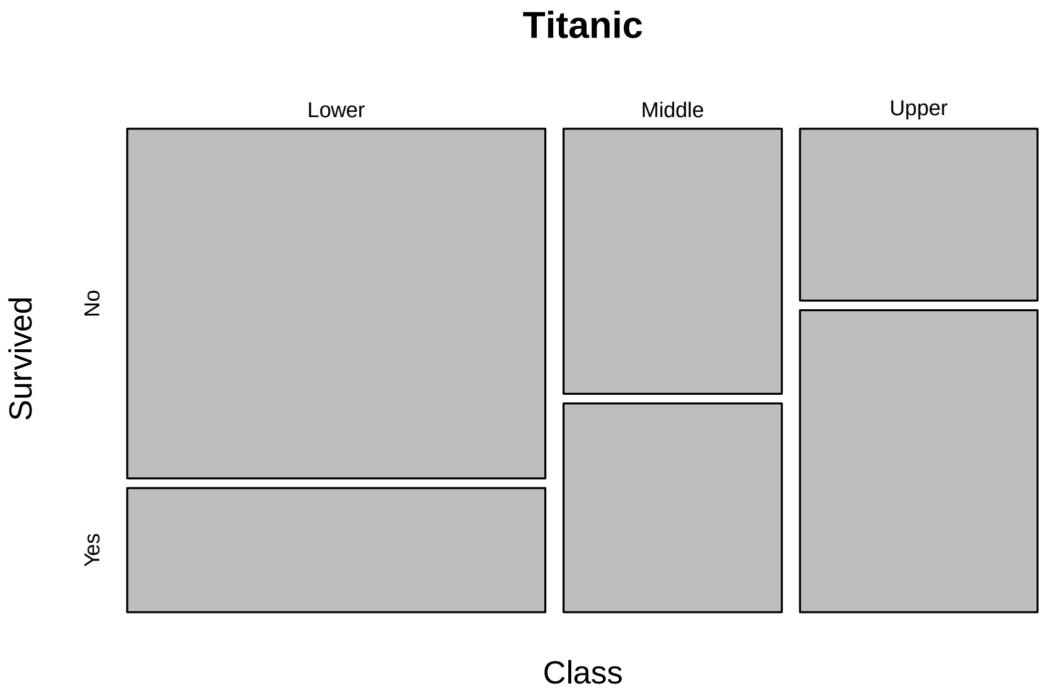
```
##         Survived
## Class             No        Yes      Total
##   Lower  0.35311005 0.12535885 0.47846890
##   Middle 0.13971292 0.11004785 0.24976077
##   Upper  0.09856459 0.17320574 0.27177033
##   Total  0.59138756 0.40861244 1.00000000
```

Voila, now we have a contingency table with proportions instead!

Creating Mosaic Plots:

Of all the plots, this is my favorite; and it is one of the least familiar graph type for the average folks. Mosaic plot conveys meaning in the width of the bars – normally, the height is the only one modified. To make sense of this easier, I will simply run the script and demonstrate what mosaic plots look like.

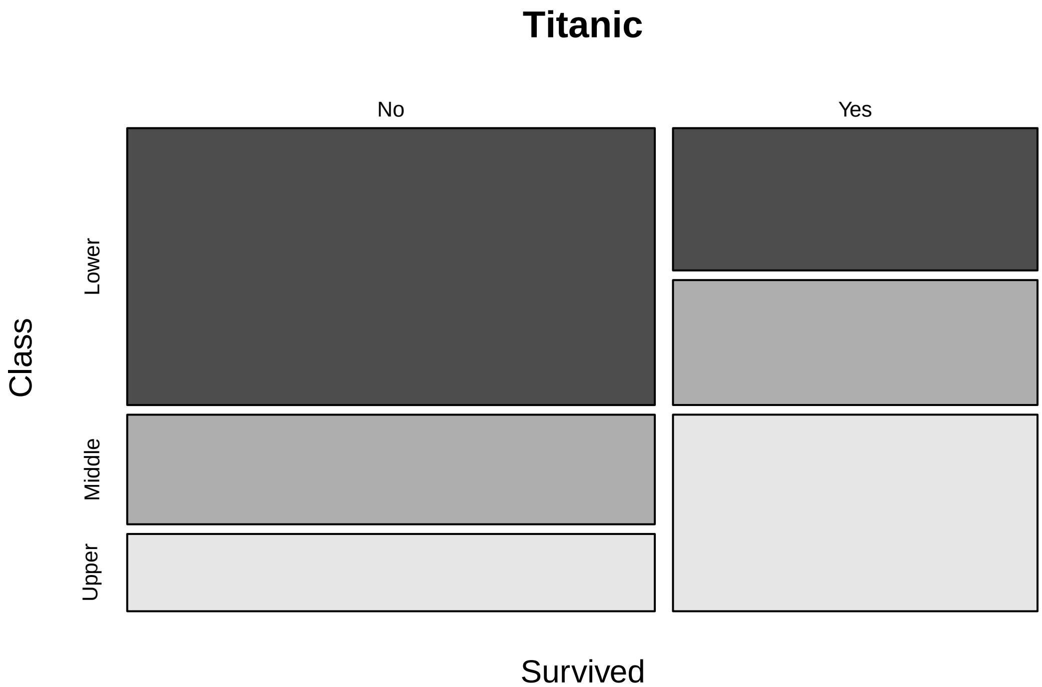
```
mosaicplot(~Class + Survived, data=Titanic)
```



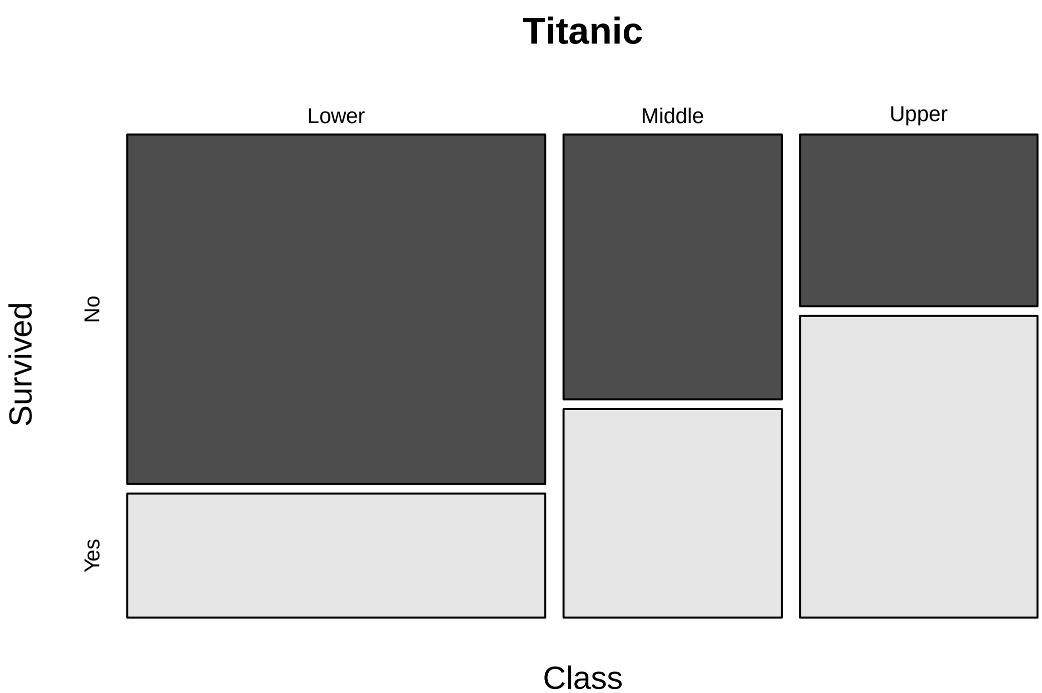
Mosaic Plots are really nice to look at. Also, they provide readers greater meaning as both the width and height are both modified and scaled. Here, the width of the lower, middle and upper class tickets are differentiated as well. So we are given additional context on the spread across the ticket holders.

Alternatively, I could have switched the x-axis with y-axis. Here's a side by side comparison – do note that I am passing the parameter `color=TRUE` inside the `mosaicplot()` in order to add basic colors and for contrast:

```
mosaicplot(~Survived + Class, data=Titanic, color=TRUE)
```




```
mosaicplot(~Class + Survived, data=Titanic, color=TRUE)
```

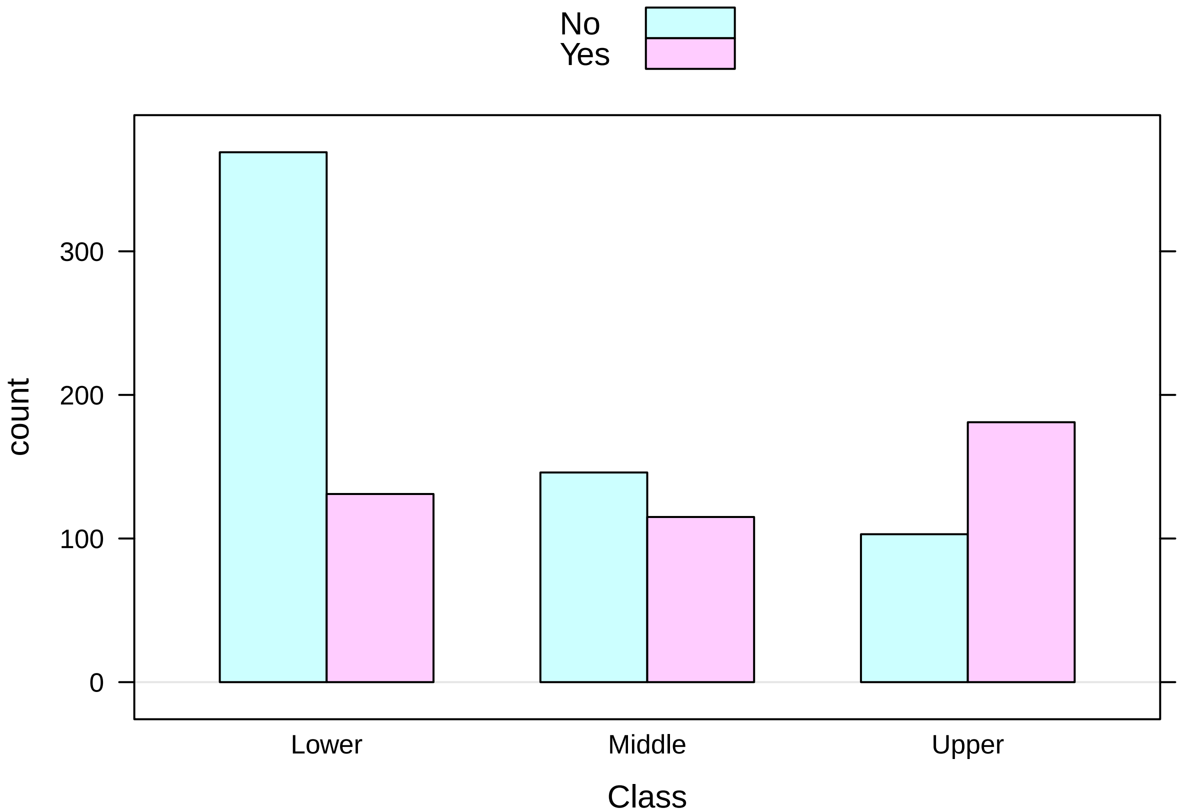


In either case, the mosaic plots were able to provide us extra information on the relationship between Survival and Class with context on their scales. Try to utilize it in your future projects!

Dodged bars for two categorical data:

Dodged bars are often used in comparing categorical data. It is another effective way to contrast patterns. The following script will contrast the distrubution of survival across ticket class.

```
bargraph(~Class, groups=Survived, auto.key=TRUE, data=Titanic)
```



Note: For `bargraph()`, I included what may seem foreign to you: `auto.key=TRUE`. While `bargraph()` without this parameter will work, it will not label the sub-categories in each x-axis category.

Advice: Try running the script in your local environment without the `auto.key=TRUE` to see the small but important change made by the said parameter.

Note 2: `bargraph()` lacks flexibility specially in the graphics arena. For example, colors cannot be modified unless done through a drastic and terribly unnecessary code. For 'full' flexibility, `ggplot()` through ggplot2 is preferred – which is frankly another tutorial on itself. The main goal here is to familiarize yourself what dodged bars looked like.

R-Tutorials Using Titanic Data

Part 3: Descriptive Statistics through R

This is where real Statistics begin. In statistics, we tend to focus on central tendencies of variables and data set. So the module will discuss Mean, Median and Mode, and how to extract these information using R.

Mean as the central tendency

Mean is the one we are most familiar. Most common folks refer to mean when they mention 'average'.

Suppose we are interested in the mean age of titanic passengers. In this case, we simply need to call `mean()`.

```
mean(~Age, data=Titanic)
```

```
## [1] 29.84211
```

By using `mean()`, we found out that the mean age of Titanic passengers is roughly 30 years old. While `mean()` might be a good measure of central tendency, it can be sensitive to large outliers specially when the sample size is small. So `median()` is another measure of central tendency we will use.

Median as the central tendency

Median is a very useful metric. It is not easily affected by outlier data points unlike mean. Moreover, many economists actually refer to the median home price when discussing 'average' home prices. So the use of median as the 'average' is not too uncommon.

Suppose we are interested in the median age of titanic passengers. In this case, we simply need to call `median()`.

```
median(~Age, data=Titanic)
```

```
## [1] 28
```

In our Titanic Data, the `median()` function tells us that 28 years old is the median passenger age for the Titanic.

Note: Although not discussed and relevant to the dataset here, one feature of median you should know or remember is that the median, as a measure of central tendency, is resistant to outliers.

Mode as the central tendency

Mode is perhaps the least used among the three central tendencies discussed in this tutorial. By definition, it is a metric to describe the most occuring results in a variable. It is not commonly used that there's not a built-in R function to handle Mode.

Advanced: As stated, there are no built-in R function for mode. For this, we would need to create a function from scratch.

The very first step is to create a table from the titanic data using `table()` method:

```r
table(Titanic$Age)
```

```
##
##  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
##  3 19 12  7 10  5  6  4  6 10  4  4  4  5 10  6 19 20 42 29 24 41 44 26 49
## 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
## 34 31 30 35 30 42 23 28 21 18 23 33  9 15 20 21 11 18  9 10 21  8 14 14  9
## 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 70 71 74 76 80
## 15  8  6  4 10  8  5  5  6  3  7  5  5  4  5  3  1  1  3  2  1  1  1
```

Reading and interpreting table data: The table above implies that there are 3 infants younger than 1, 19 children around the age of 1, etc.

Now, let's utilize the `sort()` function. `sort()` will accept table as a parameter, then sort it by ascending order.

```r
sort(table(Titanic$Age))
```

```
##
## 66 67 74 76 80 71  0 59 65 70  7 10 11 12 53 63  5 13 56 57 61 62 64  6  8
##  1  1  1  1  1  2  3  3  3  3  4  4  4  4  4  4  5  5  5  5  5  5  5  6  6
## 15 52 58  3 60 46 51 55 37 43 49  4  9 14 44 54 41  2 47 48 38 50 34 42  1
##  6  6  6  7  7  8  8  8  9  9  9 10 10 10 10 10 11 12 14 14 15 15 18 18 19
## 16 17 39 33 40 45 31 35 20 23 32 19 27 29 26 36 25 28 21 18 30 22 24
## 19 20 20 21 21 21 23 23 24 26 28 29 30 30 31 33 34 35 41 42 42 44 49
```

After running the script, indeed, `sort()` was able to sort our data in ascending order. But let's extract the very last value since the value – not the table – is the data we are interested in.

For this task, we need to indicate the index number we are interested in. Assume we have a table of length 10 called fakeData, `sort(fakeData)[1]` would call the very first item with the least occurence. And the `sort(fakeData)[10]` would call the very last data with the most occurence, which is the definition of mode.

Now, let's go back to our Titanic Data. We don't know the length of our data. But we can simply use `length(table(Titanic$Age))` and pass that inside the bracket:

```r
sort(table(Titanic$Age))[length(table(Titanic$Age))]
```

```
## 24
## 49
```

The script returns `24 49` which implies: the mode is 24 years old with a number count of 49. So our script works! **Note, the custom script only works for data sets with one mode** If you have multiple modes, the script will only return one. Iterative loops would be the easiest way to do this but it is outside the scope of this tutorials because functional programming is not the goal of this statistic tutorials.

Range

By definition, range is the difference between the smallest and largest value. To find the smallest and largest values, we can use `range()`:

```
range(~Fare, data=Titanic)
```

```
## [1]   0.00 512.33
```

As demonstrated, `range()` returns a vector with two values: the lowest and the highest value in the dataset. We can manually compute the actual range using thiese values by substracting these two manually.

Alternative, since `range()` returns a vector, we can do the following:

```
range(~Fare, data=Titanic)[2] - range(~Fare, data=Titanic)[1]
```

```
## [1] 512.33
```

As seen above, we were able to retrieve the actual range by substracting the lowest value at index 1 from the highest value at index 2.

IQR

IRQ, shorthand for interquartile range, is another measure of variability we see in Statistics. The inter-quartile range is a measure that indicates the extent to which the central 50% of values within the dataset are dispersed. In short, we can describe IQR as a measure of where the bulk of the values lie by substracting the first quartile value from the third quartile value.

```
IQR(~Fare, data=Titanic)
```

```
## [1] 27.45
```

By running `IQR()` above, we were able to calculate that the IQR for the ticket fare is exactly 27.45.

Variance

Work on this

Standard Deviation

Work on this

TO DO: ADD COMMENTS IN THE CODE

//experimenting

```
#result <- getmode(Titanic$Age)
#names(sort(-table(result)))[1]

sort(table(Titanic$Age))[length(table(Titanic$Age))]
```

```
## 24
## 49
```

```
favstats(~Age, data=Titanic)
```

```
##  min Q1 median Q3 max     mean       sd    n missing
##    0 21     28 39  80 29.84211 14.38826 1045       0
```

I will add stuff here as I go:

The rest – if any – needs to be edited/deleted as I go: