Problem Set Debrief

Most of you did fantastically on the problem set. Mostly very well presented as well! I didn't closely grade the practice problems but commented if I saw something was obviously wrong. This time, I only gave zeroes to people who did not hand the problem set in, so please see me privately if you received a zero and thought you handed your problem set in. Some common issues were:

- Work in groups - there is certainly a correlation between group work and quality of assignments.
- Make sure you actually practice the proofs for the midterm. It'll help.
- Optional arguments to R functions don't need to be included. You can tell they're optional in the documentation if they're given a default value in the Usage section of the documentation.
- Make sure you know what is meant by measure of fit - review the notes if you're not clear.
- Presenting numbers in awkward units. Humans don't like to look at too many digits, so always consider scaling/rounding when presenting in tables and plots.
- In general in tables, you should left align text, and right align numbers. These rules are not hard and fast.
- Titles on graphs.
- Don't repeat yourself (DRY). In programming, if you end up wanting to repeat code, you should almost always put that code into a function. It makes your work much more readable.
- In `knitr` chunks, you can use `message=FALSE` to suppress `R` messages.
- Words in your proofs will improve their presentation.
- Differing colors should only be used in figures if they convey extra information, like representing a third dimension in the data.
- Always put your final results in tables for presentation.
- You can use `\usepackage[margin=1in]{geometry}` in your preambles to get nicer margins.

The grammar of `ggplot2`

Now let's dig a little deeper into how `ggplot2` works.

`ggplot2` is an implementation of the "grammar of graphics," described in a book of the same title by Leland Wilkensen. The idea is that graphical representations of data, like language, have a logical grammatical structure. Most graphing packages ignore this structure and create one-off solutions for every different kind of graph that we might want to display. This is inefficient, and therefore displeasing to economists. `ggplot2` allows users to control the composition of statistical graphs by directly controlling the grammar of the graphical components.

Plots in `ggplot2` are built by putting together separate component parts. The four crucial components that we'll think about for now are:

1. data

2. aesthetics
3. layers/geometric shapes
4. themes

There are more, but these are the important ones. We'll tackle each separately.

Data

The *data* for ggplot2 should always be packaged into a data.frame. After loading the ggplot2 library, we'll load the classic Fisher's Iris data set to demonstrate:

```
library(ggplot2)
data(iris)
## ggplot(data = iris, ... )
```

The first argument we pass to ggplot() will be the data frame that we intend represent graphically. This isn't the only way to get data into your ggplot2 graphs, but is probably the best if you are graphing from a single data.frame.

Aesthetics

The second required argument for ggplot() is the aesthetic mapping of the plot. Aesthetics are used to map data to "things that you can see", such as the position of the data on the axes, the color, the shape, et cetera. Now we can create and display the ggplot2 object g using our data an aesthetics.

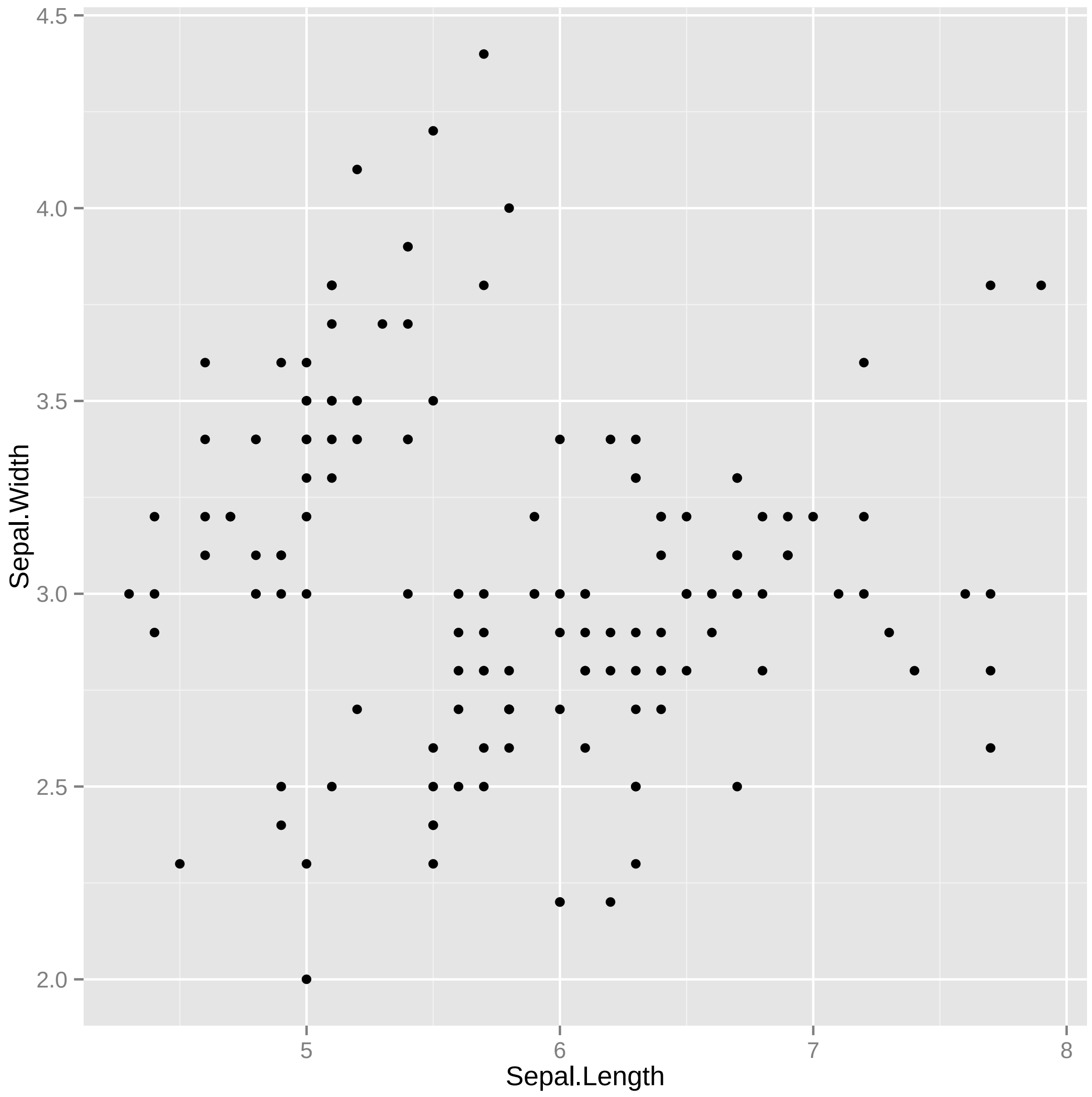
```
g <- ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width))
```

Or not. Why are we getting an error? Because we haven't specified any layers. So, in ggplot2, aesthetics are not the *only* thing that you see. We will also use layers to specify the class of graph that will be used to display the data.

Layers/geometric shapes

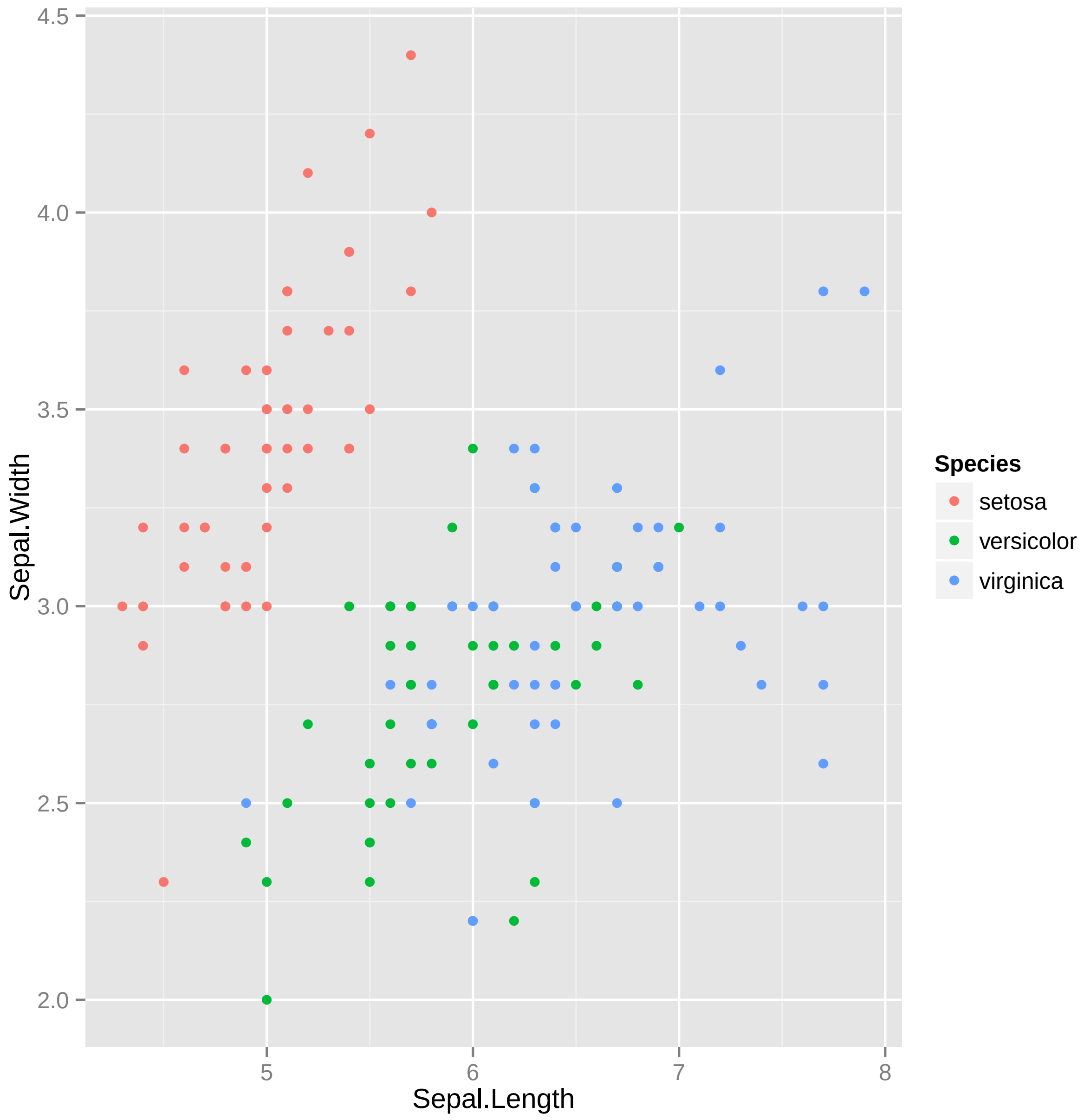
We've specified our data, and our data aesthetics, but not our *graphical layers* (i.e. geometric shapes, or geoms). Here, we'll add a layer of points:

```
g + geom_point()
```



We can also specify additional aesthetic options for each layer. Below, we'll tell `ggplot2` to graph the points again, this time specifying that each species should have a different color. Aesthetic options specified in the `ggplot()` function are the default for all layers, but aesthetics specified within layers can override the defaults for that layer only.

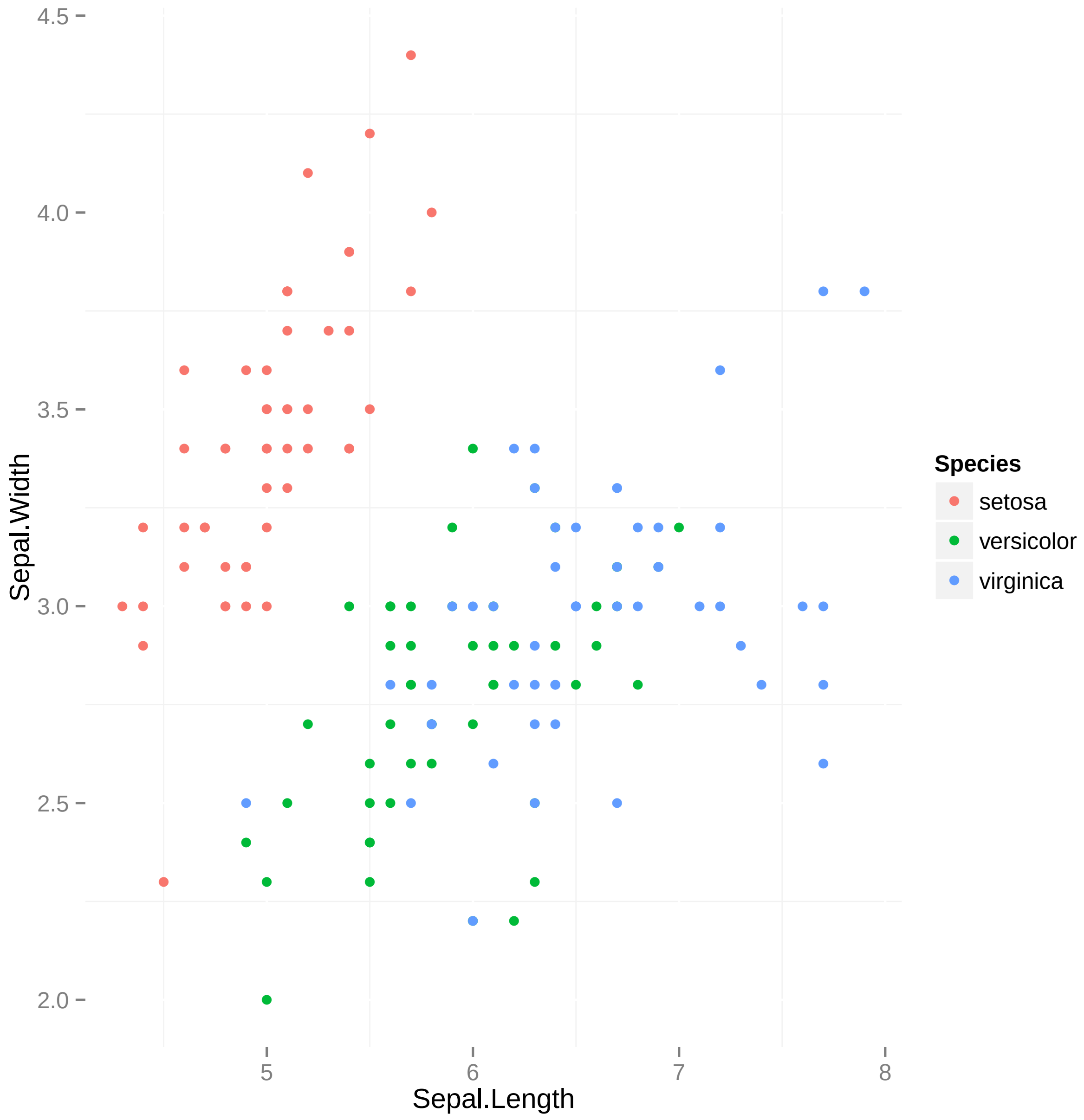
```
g + geom_point(aes(color=Species))
```



Themes

Themes cover the appearance of the graph that has nothing to do with the data itself. The background color, for example.

```
g + geom_point(aes(color=Species)) + theme(panel.background = element_blank())
```



Initially, putting together the grammar of `ggplot2` may seem cumbersome, because it is. In fact, the code to construct simple scatterplots or histograms in `ggplot2` is almost certainly going to be more complex than a simple `plot()` or `hist()` from the base graphics package.[1]. But as your graphics needs become more complex, you will almost certainly find that `ggplot2` scales much better and is far more powerful than the base functions provided by `R`. The documentation (`http://docs.ggplot2.org/current/`) is pretty good and most questions have been asked out there in the ether.

My personal solution is usually to write a plot wrapper function for each individual project, if I'm going to be using similarly styled graphs. Remember to apply DRY (Do not repeat yourself), even to `ggplot2`.

[1]In fact, `ggplot2` provides a function called `qplot()` that replicates the simpler syntax from the base graphing package, if you prefer.

Intro to API usage (simplest version of web scraping) in R

Since Max is away this week and the material has stalled, for this part of the section, we will look at the simplest version of web scraping using the `download.file()` function. We will see how far we get, and conclude at the start of next section if we have to. The context for this I will use is the popular weather dataset from Oregon State, PRISM. PRISM provides a daily gridded dataset for the contiguous United States that includes maximum temperature, minimum temperature, mean temperature, and precipitation at the 2.5 arc-minute by 2.5 arc-minute level. This corresponds to grid cells with dimensions of approximately 4.5km by 3-4km in the USA.

An API (application program interface) is a system for interacting with a website or application which is intended to simplify the process of getting data or something to happen. For us, we almost always will be using APIs to download data from an online repository. The basic principle is fairly simple; we provide a URL with correctly specified options, and receive the data in return.

We will go through a simplified version of the `getPRISMGrid` in my (incomplete) `prismUtils` package.

Please do not use the below function after today.

Oregon State has been really cool to allow this dataset to be accessed publicly and so easily. As such, we really don't want to do anything to make them not want to do this. Using more bandwidth than necessary by continually re-downloading the same file is one of these things which could make them not want to host these data. Please use the `downloadMany*` functions in the package, which permanently download the files so you only request them online once.

```r
library(rgdal)
getPRISMGrid <- function(date, type, range = "daily") {
    # get a temporary .zip file name
    tmpFile <- tempfile(pattern = paste(date, type, sep = ""))
    fileName <- paste(tmpFile, ".zip", sep = "")

    # The url with type, date, and range
    url <- paste0("http://www.prism.oregonstate.edu/fetchData.php?type",
        "=bil&kind=recent&elem=", type, "&range=", range,
        "&temporal=", date)

    # Download the file to fileName
    download.file(url = url, mode = "wb", destfile = fileName)

    # if a SpatialGridDataFrame is to be returned: unzip the
    # file to the working directory.
    tmpFile <- unzip(zipfile = fileName, overwrite = TRUE)

    # read into R
    spGrid <- readGDAL(fname = tmpFile[1])

    # delete the zip file
    file.remove(tmpFile)
    return(spGrid)
}
mygrid <- getPRISMGrid(date = "20140503", type = "tmax",
    range = "daily")
```

Calling `image(mygrid)` would then bring up an image of the data. The function has several steps; firstly, we come up with a file name to store the downloaded `.zip` file. The `tempfile()` function nicely finds an appropriate spot on our file system that we can download to. Next we provide the correct URL, which we talk about in more detail below. Finally, we read the spatial grid file into `R` using the `readGDAL()` from the `rgdal` package. We'll talk more about spatial data analysis in a future section.

Now, to work out what URL we're supposed to use. Sometimes, APIs have great documentation and it's really easy to work out what to put. PRISM is certainly one of these APIs (see `http://www.prism.oregonstate.edu/documents/PRISM_downloads_web_service.pdf`).

For now, let's do it the hard way. What we're going to do is replicate what your browser does when it goes through the website and you click the download button. Let's go to `http://www.prism.oregonstate.edu`, click "Recent Years". Choose "Daily Data", and enter a date. *Right* click on "Download Data (.bil)". Sometimes, you might get lucky and see a "Copy link address" option here. Usually this isn't the case for these sort of websites.

Now, open your developer tools. In Chrome, I have to go to Options (the three horizontal line sandwich thing), More tools, Developer Tools, and click Network. In Firefox, click the sandwich button, Developer Tools, and Network. I don't use Safari, and it looks like you might have to download a separate Safari for Developers program. Someone correct me if I'm wrong.

Now left click the "Download Data (.bil)" button. You should see things happening. Click on the POST request that comes up. After a bit of fishing you should see the pieces that go into the URL that gets created in the function:

```
## http://www.prism.oregonstate.edu/fetchData.php?type
## =bil&kind=recent&elem=tmax&range=daily&temporal=20140503
```

You should see that the first part is labeled with "Request URL:". The rest should be labeled with "Form Data" or "Params". The basic structure, you can see, is then:

```
## <request url>?<param1Name>=<param1Value>&<param2Name>=<param2Value>&...
```

It will be up to you to follow your nose with working this stuff out usually. Firstly, try and find readable documentation for the API, next go through this method we just went through, and as a last resort, email the managers of the API for help.