Plotting Real Data using ggplot

Nick Sumpter

2022-08-21

Today's lab

Today we will continue our lessons in `ggplot` by working with some simulated real data. The purpose of this lab is to show you how to plot data that is more like that which you will come across in your research. Additionally, we will work with some of the more complicated parts of `ggplot` so we can get closer to publishable plots. This will be a very valuable skill for the remainder of this course, as visualizing the relationship between variables is a vital step in both choosing and interpreting the appropriate statistical test.

Loading Packages and Data

Real data is usually recorded and stored in an Excel document of some sort (often .csv or .txt files) and thus needs to be read into R before we can work with it. There are several ways to read in data, but for this lab we will teach you the `tidyverse` method using the `read_delim` function from the `readr` package (included with `tidyverse`).

As usual, let's first look up the `read_delim` function in the Help panel. When you search for it, you will see two options, click on the first option to be taken to the `readr::read_delim` help page. You will see that this function 'reads a delimited file into a tibble'. The term 'delimited' refers to the 'delimiter' of a file. A 'delimiter' is essentially a single character that separates elements of a table. For example, a comma-separated value (csv) file uses the comma (`,`) to 'delimit' elements. A tibble is a way that tabular data is stored in R (specific to `tidyverse`). You can specify which delimiter `read_delim` should expect using the `delim` argument. Today's data is stored as both a .csv file type (comma separated value) and a .txt file type (tab delimited). For the .csv file the appropriate `delim` argument is `,`. For the .txt file, you need to tell it to expect tabs as the delimiter by setting the `delim` argument to `\t`, which is just the way that the tab character is coded in R.

As this is a new lab, we will first need to load our `tidyverse` library, then we can use functions from within the `tidyverse`. When we use `read_delim` we will need to tell R that we want the dataset to be stored in our local Environment. This will involve the assign function `<-`.

However, there is one more step that we need to do prior to reading in data: setting our working directory. The easiest way to do this is to open the Files panel in the bottom right corner of RStudio. From there, navigate to the directory you wish to work in, making sure today's data is sitting in that directory. Now, click on the 'More' drop-down menu (with a small cogwheel next to it) and select the fifth option 'Set As Working Directory'. You will notice that the function `setwd()` will appear in your console, with the path to your working directory as it's only argument. I would suggest copying this into your R script then running it from the script whenever you return to this document.

```
library(tidyverse)

# setwd("your_directory_here")
setwd("~/PhD/Teaching/GRD770/R Labs 2022/Lab 4 - Plotting Real Data using ggplot")

realdata <- read_delim(file = "realdata2.csv", delim = ",")

realdata_txt <- read_delim(file = "realdata2.txt", delim = "\t")
```

As you can see, `realdata` and `realdata_txt` are exactly the same, which simply shows you that no matter the file type your data is stored as, it will be read into R as the same type of object. We will use `realdata` throughout the rest of the lab, but we could use either.

Note: if you have an Excel file with the ".xlsx" extension, you should save it as a comma-separated value or a text file prior to loading it into R. There are ways to read in .xlsx documents but it might not work as you expect it to.

Getting to Know Your Data

Before we start plotting the data, let's look at the realdata object in our Environment. Firstly, we can see that we have 40 rows (obs.) and 3 columns (variables), and clicking on the blue arrow to the left of the object name shows us some information for these variables. The first column is a character variable called Group, with the first 4 values being "A". The other two variables are numeric variables called Activity and Diet. There is also some extra data below the Diet column, but we can ignore that. Note that the `read_delim` function guessed what sort of data was in these columns, thus we might want to do some manipulation of the data to make sure the columns are representing the data correctly. We can learn some more about our data using the `summary` function like so:

```
summary(realdata)
```

```
##     Group              Activity           Diet
##  Length:40          Min.   : 1.250   Min.   :1.0
##  Class :character   1st Qu.: 5.245   1st Qu.:1.0
##  Mode  :character   Median : 6.880   Median :1.0
##                     Mean   : 6.572   Mean   :1.4
##                     3rd Qu.: 8.383   3rd Qu.:2.0
##                     Max.   :11.170   Max.   :2.0
```

So we now have some statistics for each of the numeric variables, but the `summary` function didn't help us with the character variable Group. It also appears that the Diet variable might be a Factor with numeric levels of either 1 or 2. We can do a couple of tricks to learn more about these specific columns, and perhaps modify them accordingly. One option is to coerce the columns into Factor variables, then run the summary function on these factorized variables:

```
Group_factor <- factor(realdata$Group)
Diet_factor <- factor(realdata$Diet)

summary(Group_factor)
```

```
##  A  B
## 20 20
```

```
summary(Diet_factor)
```

```
##  1  2
## 24 16
```

As you can see, we created two intermediate Factor variables (Group_factor and Diet_factor) which we could then run the `summary` function on directly. The results of the `summary` functions tell us that the Group variable has values of either "A" or "B", with exactly 20 rows per value, and the Diet variable has values of either 1 or 2, with 24 rows equal to 1, and 16 equal to 2.

Before we move on, we should update the realdata tibble such that the Group and Diet columns are encoded as Factors rather than character/numeric variables. For this we will use a new function called `mutate`, from the `dplyr` package. The Help panel tells us that this function will "create, modify, and delete columns", however the remainder of the help document is not too helpful without further knowledge. For now, I will just show you the simplest way to use this function but later on the help document will make more sense:

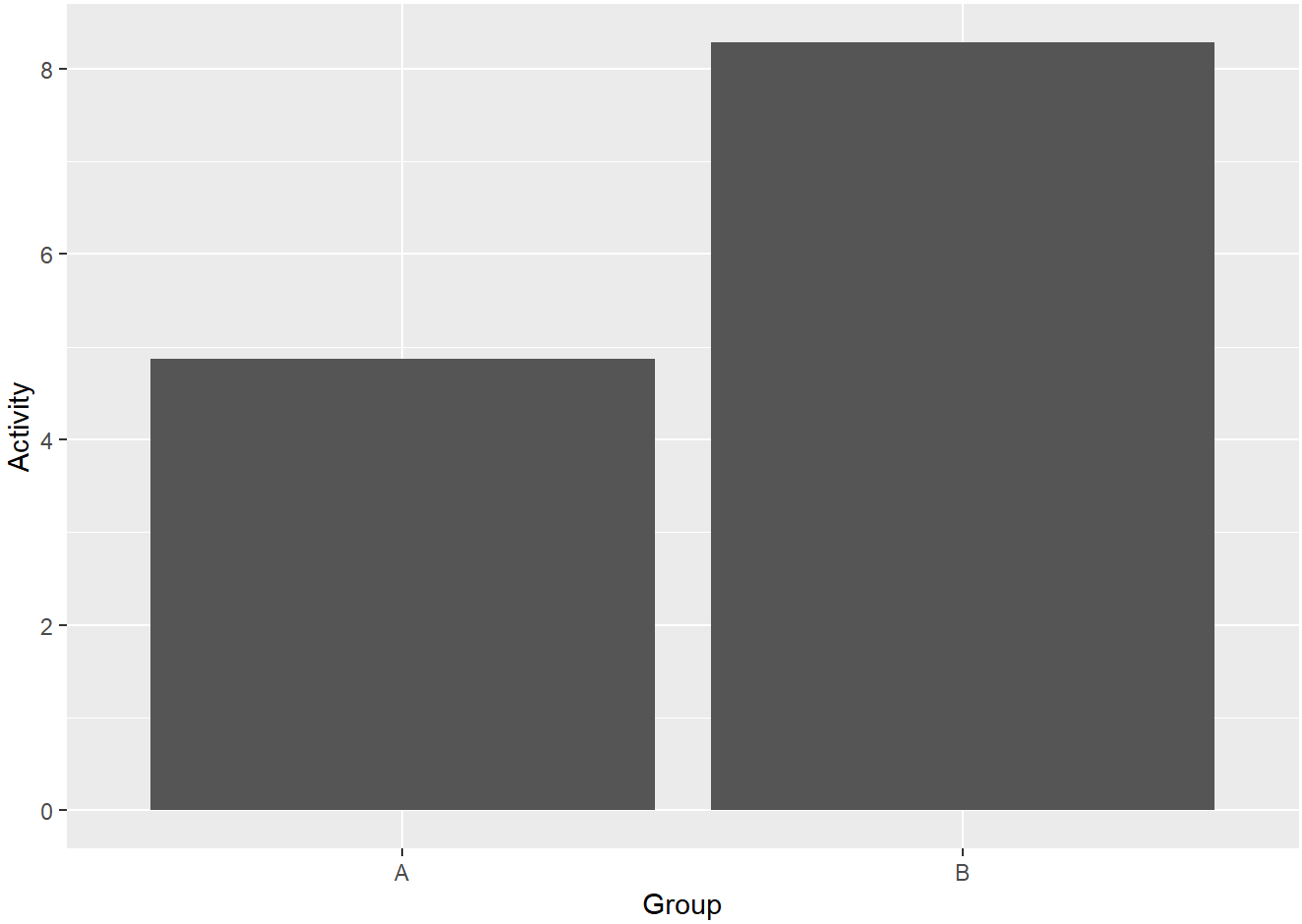
```
realdata2 <- mutate(realdata, Group = factor(Group), Diet = factor(Diet))

# Alternatively, using the variables we made in the previous step (be careful using this
method as the order of the values might have changed depending on what you did to the da
ta):
realdata2 <- mutate(realdata, Group = Group_factor, Diet = Diet_factor)
```

So we have created a new table called realdata2 that is the same as realdata, except now we have told R to make Group and Diet variables into Factors. If you now look at realdata2 in the Environment panel, clicking on the blue arrow tells us that we have successfully modified the Group and Diet columns, and that we still have the same data in the table (based on the "40 obs. of 3 variables"). Now that we are happy with our dataset, we can move on to plotting the data.

Plotting Real Data

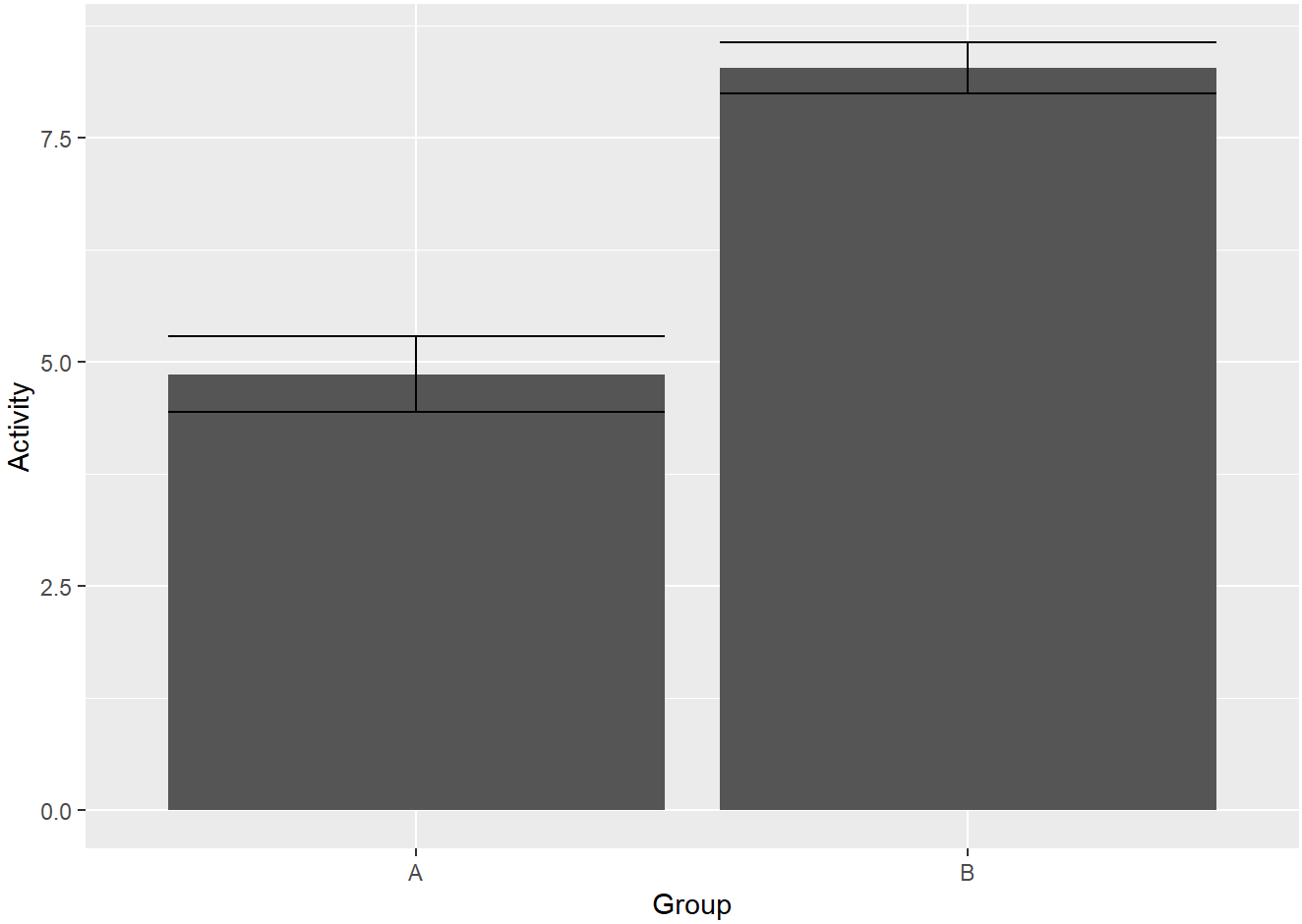
With our data cleaned up, we can now try plotting our variables of interest using `ggplot`. Lets start with a bar graph showing the mean 'Activity' in 'Group A' vs 'Group B'. This can be done using the `geom_bar` function, though we need to modify it a little by providing a `stat` and `fun` argument. The defaults for these arguments are not very useful, so we will change them to calculate the mean in each group and plot it. The `stat` we will need is `summary` and the `fun` (short for function) is `mean`.

```
ggplot(data = realdata2, mapping = aes(x = Group, y = Activity)) +
  geom_bar(stat = "summary", fun = "mean")
```



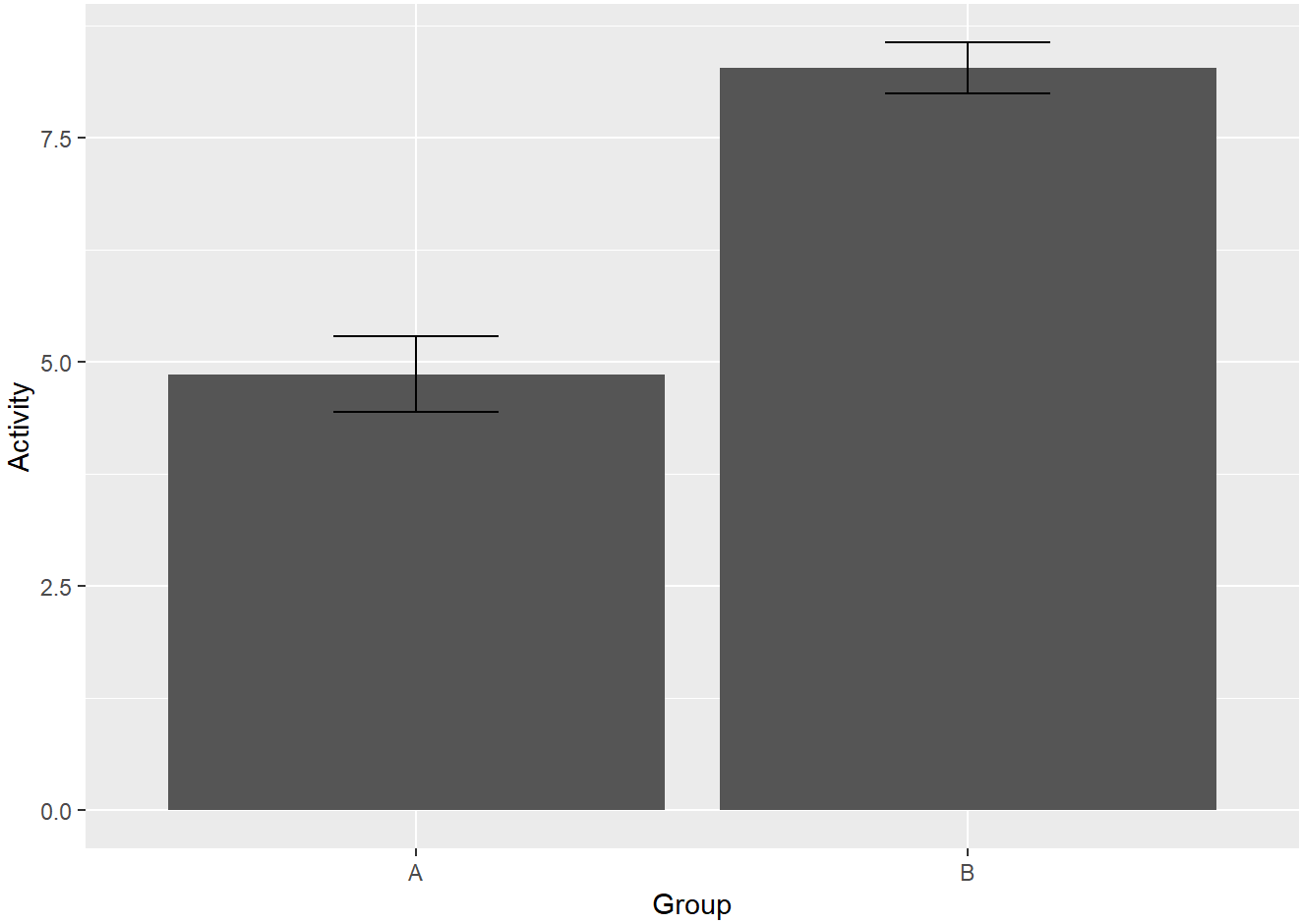
This can be improved a little by adding in the standard error as an error-bar around the mean for each group. The easiest way to do this is with the `geom_errorbar` function. Again, we will need to provide the `stat` argument, but this time we will provide the `fun.data` argument rather than just the `fun` argument. This will ensure it calculates the lower and upper bounds of the error bars correctly. The function we will tell it to use is called 'mean_se' which basically calculates the mean, the mean minus the standard error, and the mean plus the standard error. Don't worry too much about what a standard error is for right now, as you will learn this over the next few lectures/labs.

```
ggplot(data = realdata2, mapping = aes(x = Group, y = Activity)) +
  geom_bar(stat = "summary", fun = "mean") +
  geom_errorbar(stat = "summary", fun.data = "mean_se")
```



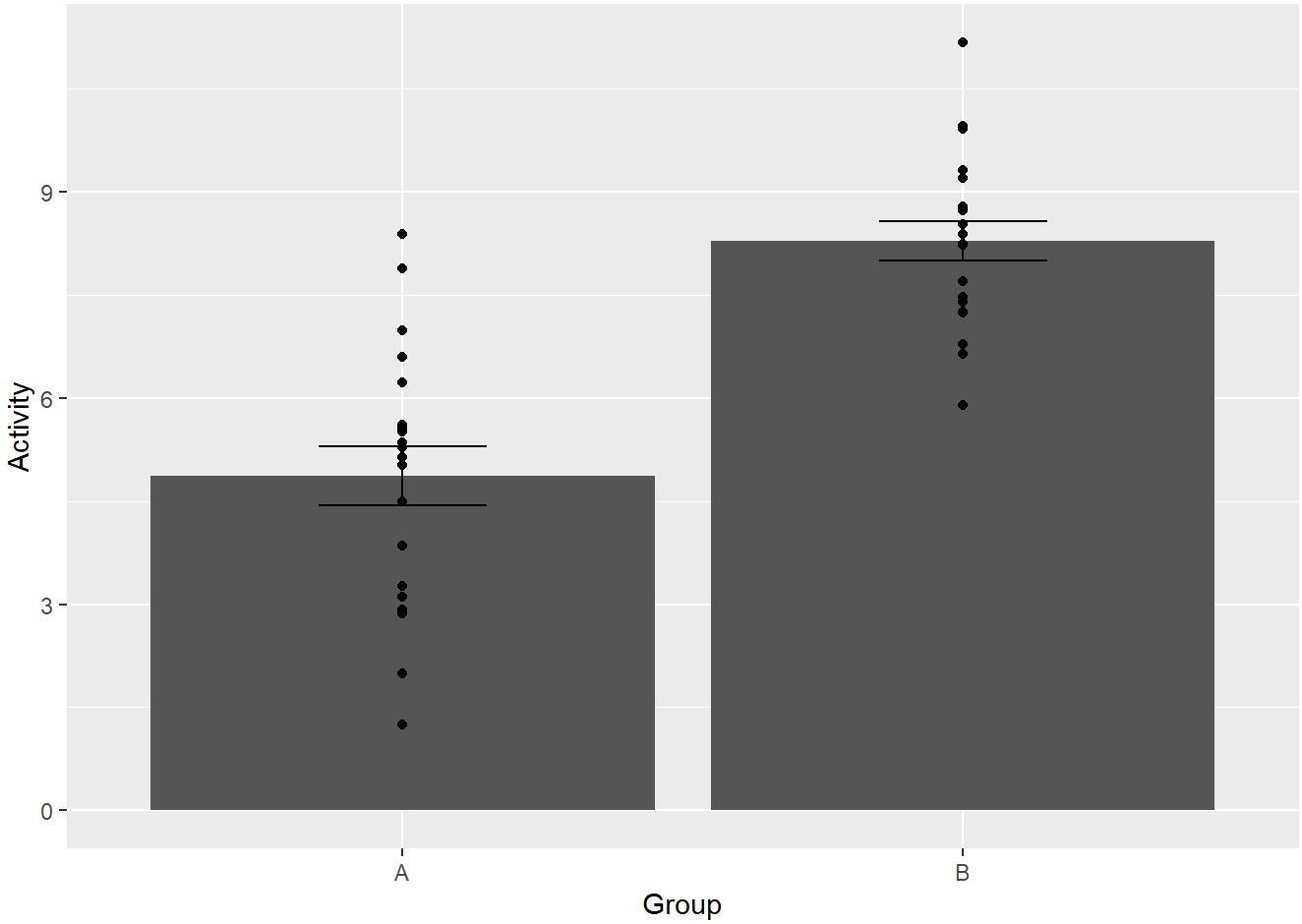
Now, even though this plot is informative, it looks a little ugly as the error bars shouldn't be the same width as the bars. We can modify their width using the `width` argument within `geom_errorbar`. The values that this takes on are a little weird, but just know that the entire width of the bar is just under 1, so you can specify a value relative to this. A good choice for today is 0.3, but you can play around to find the best value.

```
ggplot(data = realdata2, mapping = aes(x = Group, y = Activity)) +
  geom_bar(stat = "summary", fun = "mean") +
  geom_errorbar(stat = "summary", fun.data = "mean_se", width = 0.3)
```



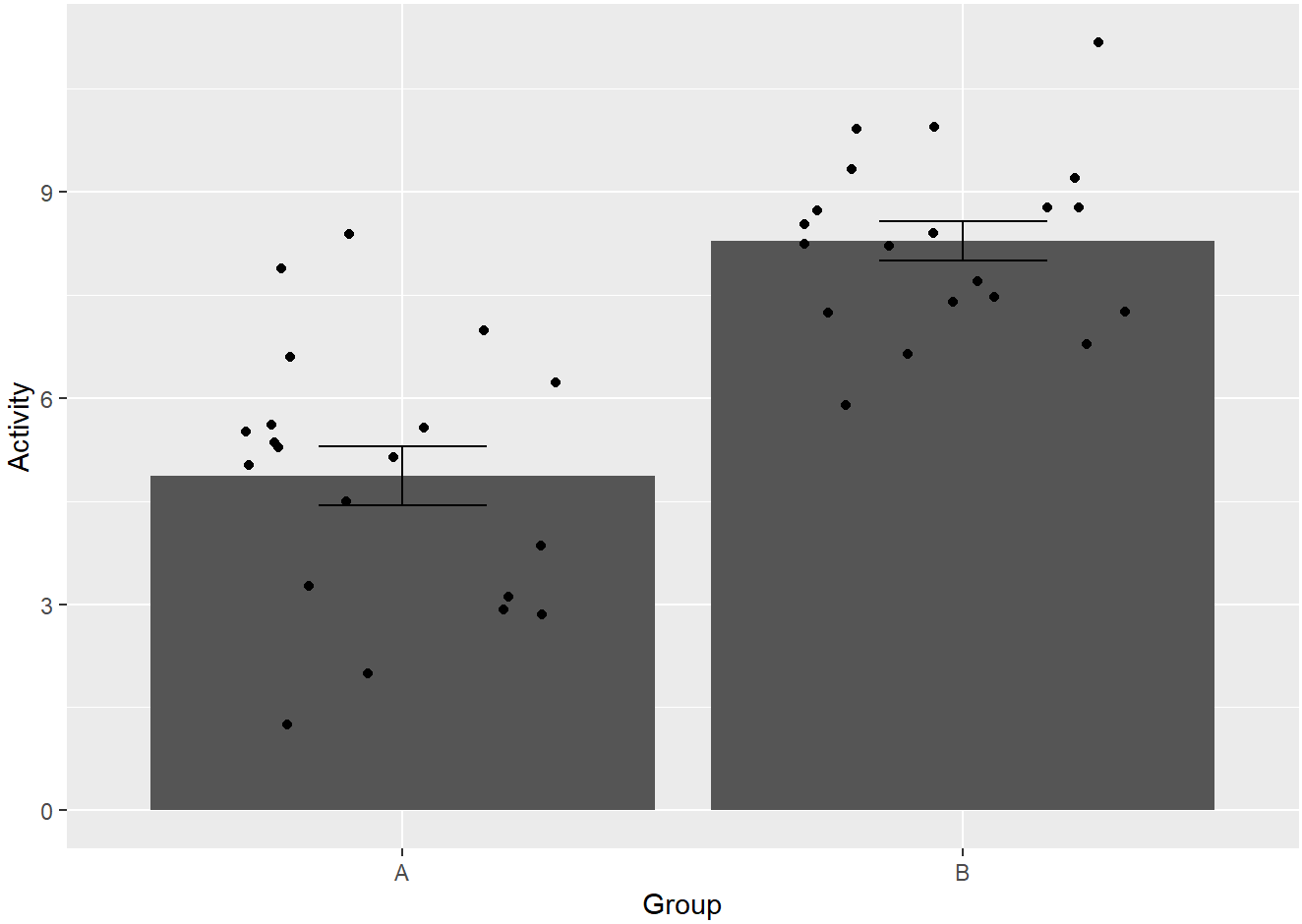
It can be informative to plot the raw data points overtop the bars, but only in cases such as this with a relatively small dataset. This can be done by adding the `geom_point` layer.

```
ggplot(data = realdata2, mapping = aes(x = Group, y = Activity)) +
  geom_bar(stat = "summary", fun = "mean") +
  geom_errorbar(stat = "summary", fun.data = "mean_se", width = 0.3) +
  geom_point()
```



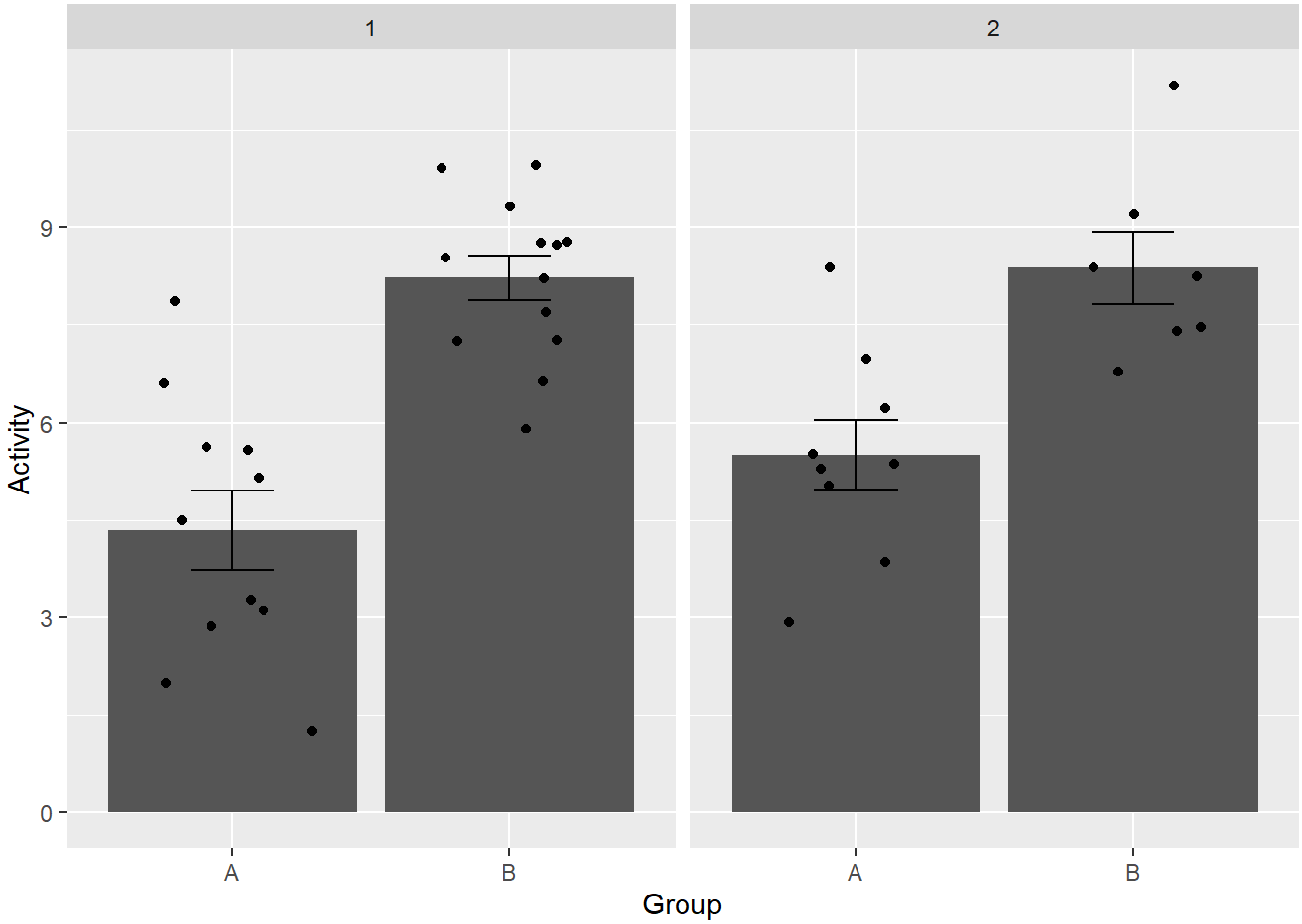
Though this shows the points, there could be situations where two points overlap one another, and so we can randomly distribute the points horizontally using the `geom_jitter` function instead of the `geom_point` function. We will want to set the `width` similarly to the `geom_errorbar` width, or else they will be too spread out. As this jitters the points randomly, it will look different every time you generate the figure.

```
ggplot(data = realdata2, mapping = aes(x = Group, y = Activity)) +
  geom_bar(stat = "summary", fun = "mean") +
  geom_errorbar(stat = "summary", fun.data = "mean_se", width = 0.3) +
  geom_jitter(width = 0.3)
```



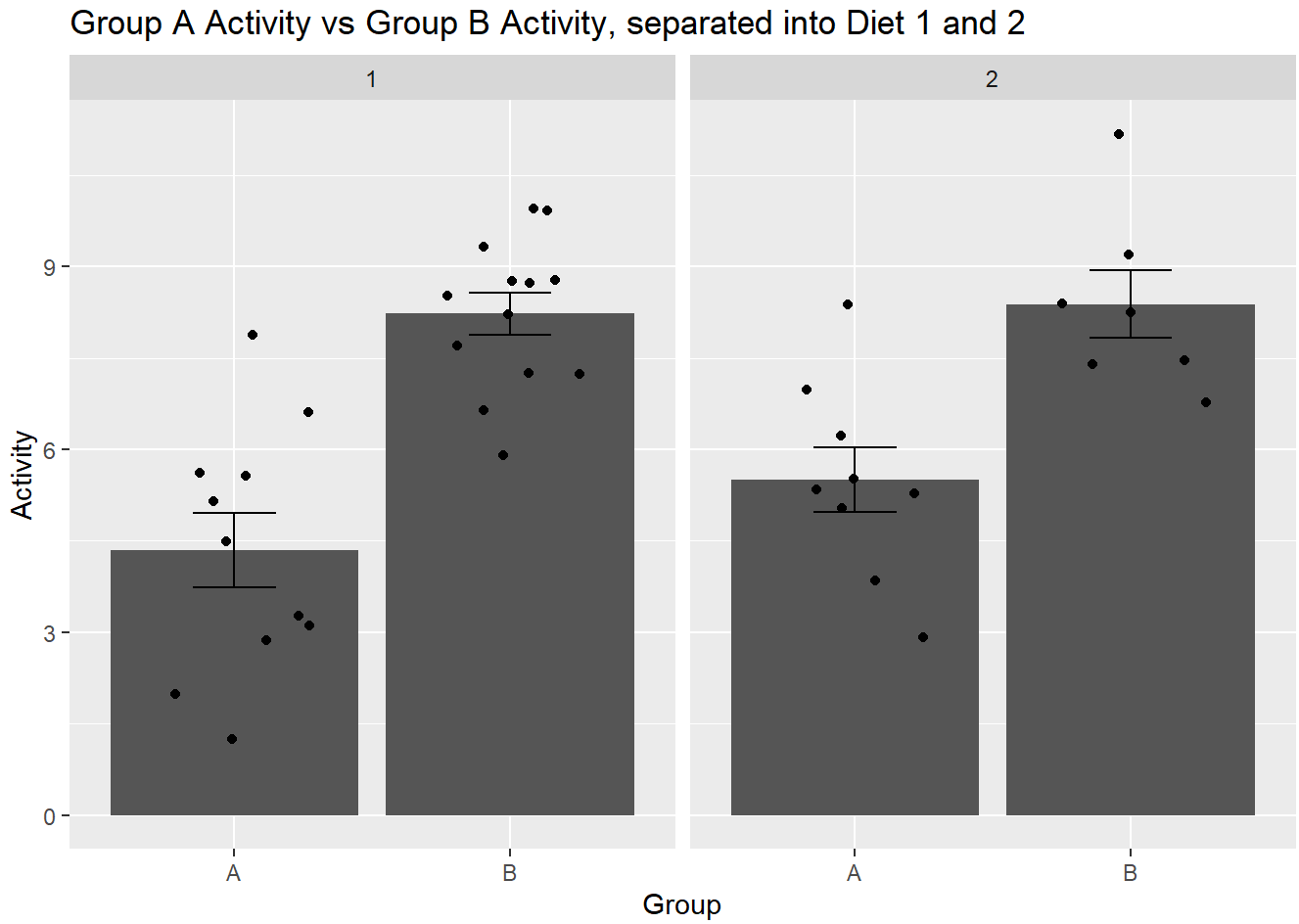
Now we may be interested in testing how the effect of Group is influenced by Diet (coded as 1 or 2). To visualize this, we can split our plot into two plots, one for each diet. We will use the `facet_wrap` function, which requires you to tell it which variable you want to split your plot based on by providing the tilde character (‘~’) before the variable name.

```
ggplot(data = realdata2, mapping = aes(x = Group, y = Activity)) +
  geom_bar(stat = "summary", fun = "mean") +
  geom_errorbar(stat = "summary", fun.data = "mean_se", width = 0.3) +
  geom_jitter(width = 0.3) +
  facet_wrap(~ Diet)
```



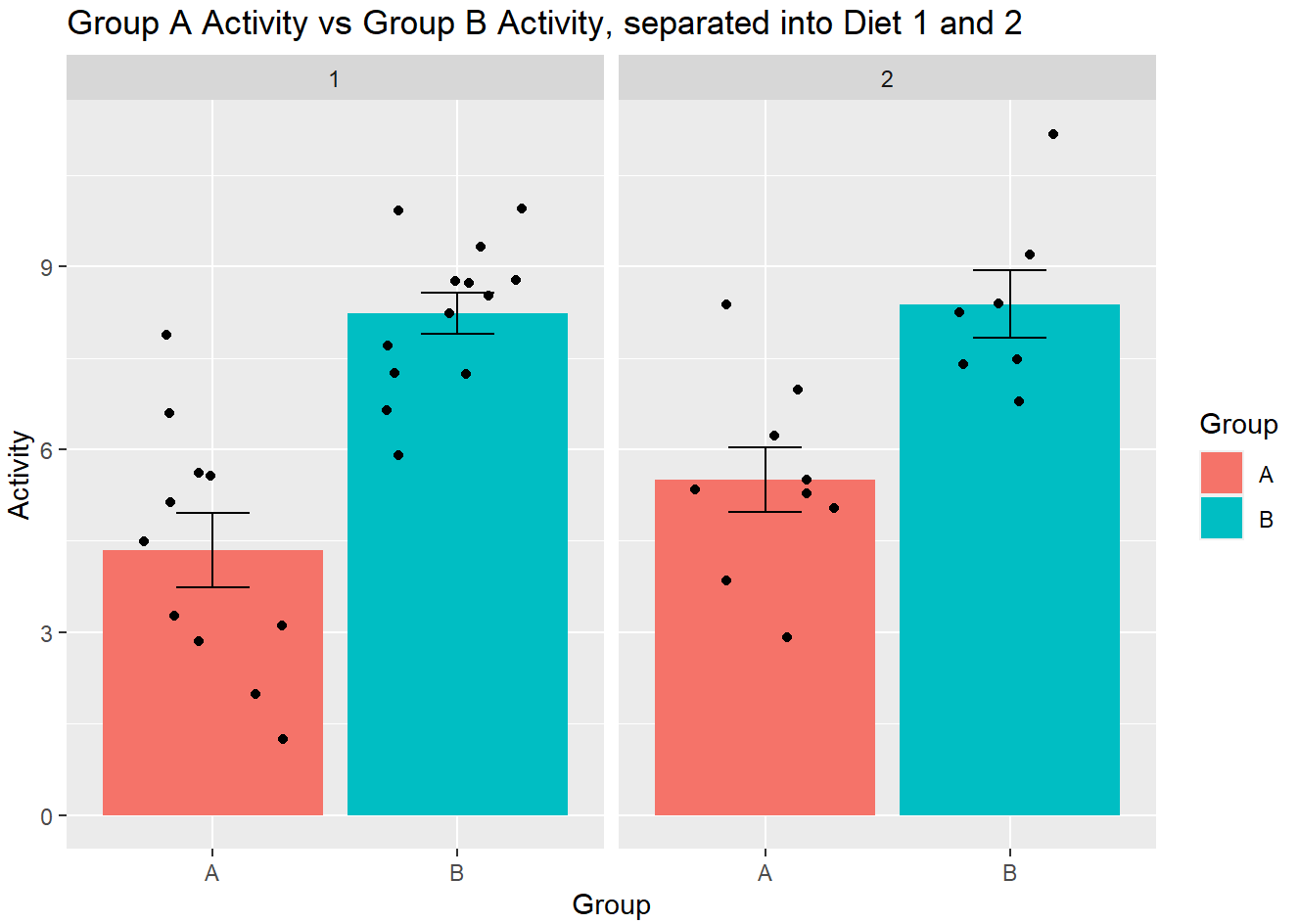
The plot now needs a title to make it a little more informative. We saw how to add this last lab.

```
ggplot(data = realdata2, mapping = aes(x = Group, y = Activity)) +
  geom_bar(stat = "summary", fun = "mean") +
  geom_errorbar(stat = "summary", fun.data = "mean_se", width = 0.3) +
  geom_jitter(width = 0.3) +
  facet_wrap(~ Diet) +
  labs(title = "Group A Activity vs Group B Activity, separated into Diet 1 and 2")
```



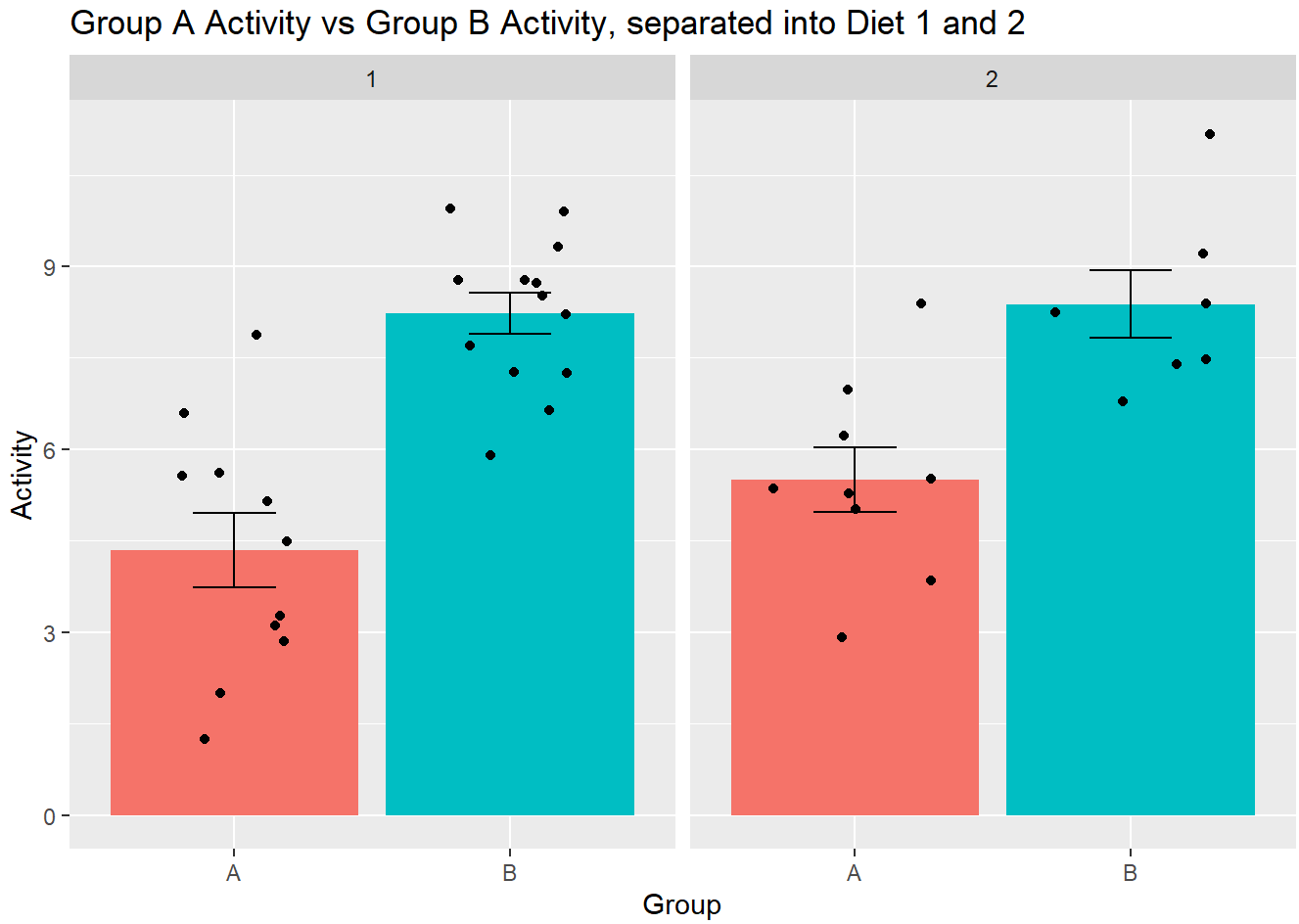
Finally, let's make our plot more attractive to look at. We will do this by including some colors and modifying the overall theme of the plot. First, lets add some color to the bars using the `fill` aesthetic in the `geom_bar` function.

```
ggplot(data = realdata2, mapping = aes(x = Group, y = Activity)) +
  geom_bar(mapping = aes(fill = Group), stat = "summary", fun = "mean") +
  geom_errorbar(stat = "summary", fun.data = "mean_se", width = 0.3) +
  geom_jitter(width = 0.3) +
  facet_wrap(~ Diet) +
  labs(title = "Group A Activity vs Group B Activity, separated into Diet 1 and 2")
```



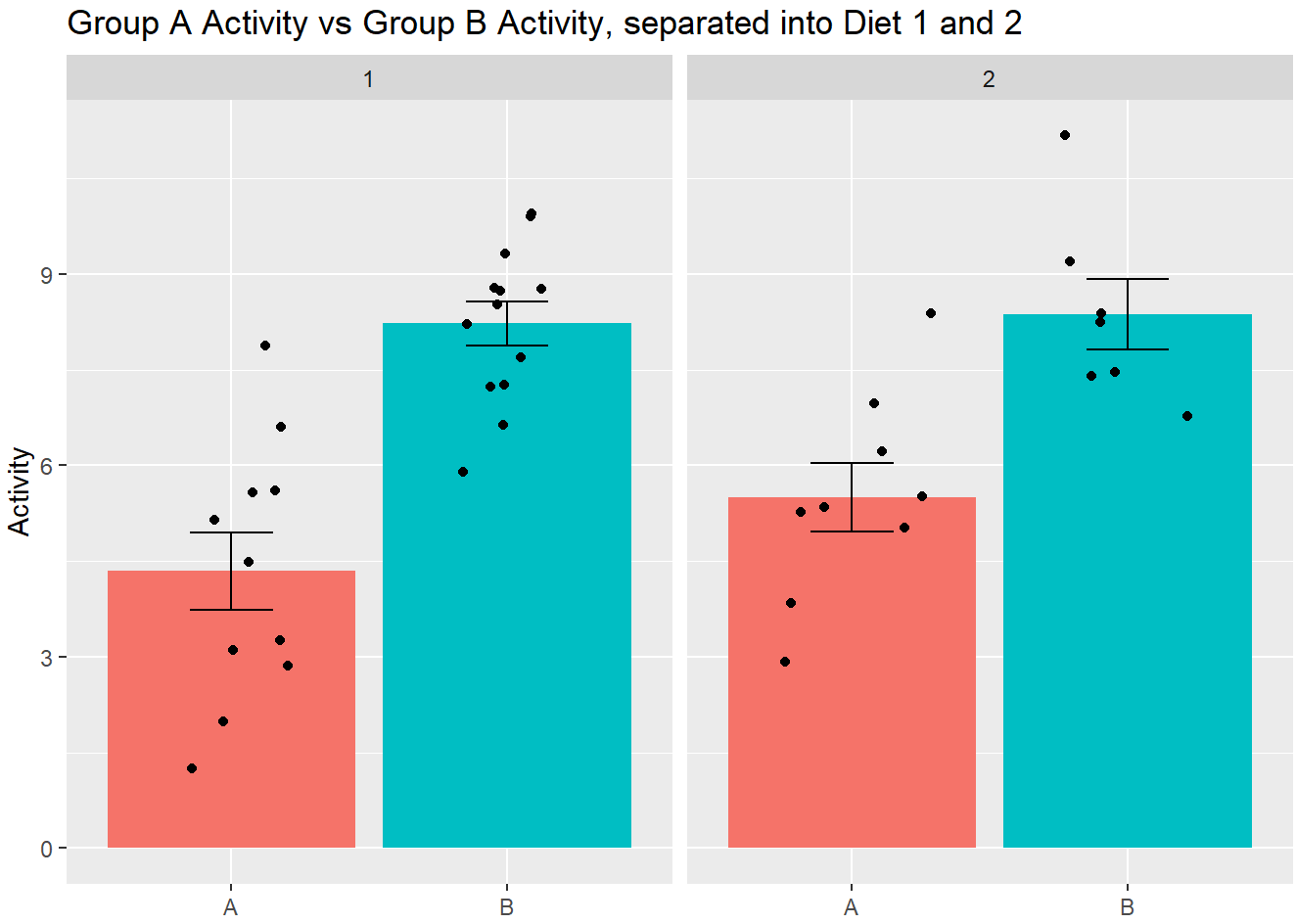
Note that the `fill` aesthetic added a legend by default. For our purposes, this is made redundant by the x-axis labels and so we can tell `ggplot` to remove it using the `show.legend` argument within the `geom_bar` function.

```
ggplot(data = realdata2, mapping = aes(x = Group, y = Activity)) +
  geom_bar(mapping = aes(fill = Group), stat = "summary", fun = "mean", show.legend = FA
LSE) +
  geom_errorbar(stat = "summary", fun.data = "mean_se", width = 0.3) +
  geom_jitter(width = 0.3) +
  facet_wrap(~ Diet) +
  labs(title = "Group A Activity vs Group B Activity, separated into Diet 1 and 2")
```



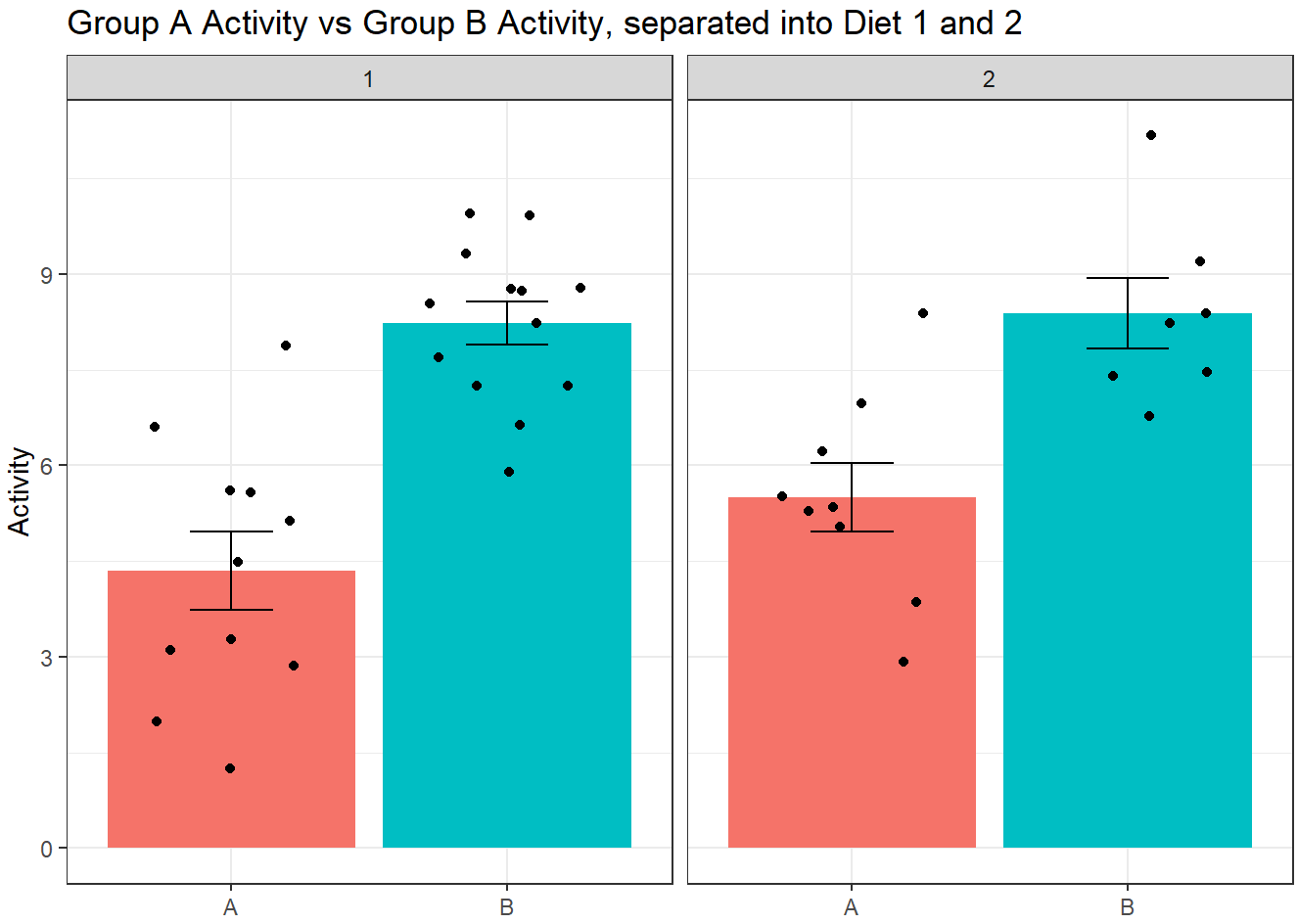
The `theme` function is possibly the most powerful function within `ggplot` as it enables customization of almost all features of a plot. Let's look at `theme` in the Help panel. As you can see, there are an enormous number of arguments for this function, which can feel a little overwhelming. In general, Google will be your friend for the `theme` function. I will show you one quick example of how to use the `theme` function by completely removing the x-axis label from the plot. This might not be a great idea in this example but can be useful for simplifying your plot.

```
ggplot(data = realdata2, mapping = aes(x = Group, y = Activity)) +
  geom_bar(mapping = aes(fill = Group), stat = "summary", fun = "mean", show.legend = FA
LSE) +
  geom_errorbar(stat = "summary", fun.data = "mean_se", width = 0.3) +
  geom_jitter(width = 0.3) +
  facet_wrap(~ Diet) +
  labs(title = "Group A Activity vs Group B Activity, separated into Diet 1 and 2") +
  theme(axis.title.x = element_blank())
```



There are also preset `theme` functions which modify the entire plot to make it look different. I personally like using the `theme_bw()` function as it gets rid of the gray background and makes things neater. **Note** you need to make sure to apply the `theme` function after the `theme_bw` function or else the `theme_bw` function will overwrite any changes from the `theme` function.

```
ggplot(data = realdata2, mapping = aes(x = Group, y = Activity)) +
  geom_bar(mapping = aes(fill = Group), stat = "summary", fun = "mean", show.legend = FA
LSE) +
  geom_errorbar(stat = "summary", fun.data = "mean_se", width = 0.3) +
  geom_jitter(width = 0.3) +
  facet_wrap(~ Diet) +
  labs(title = "Group A Activity vs Group B Activity, separated into Diet 1 and 2") +
  theme_bw() +
  theme(axis.title.x = element_blank())
```



We can set the theme for all plots in a script using the `theme_set` function. This will be used at the beginning of all future labs so that we always apply the `theme_bw` function to each plot.

```
theme_set(theme_bw())
```

Independent Practice

1. Try to plot the mean and standard error for the Sepal Width variable (y-axis) in each Species of iris (x-axis) using the `iris` dataset.

    a. Add a title and y-axis label for this data.

    b. Try customize a few more settings to make the plot how you would like it (change the colors etc.). When in doubt, Google is your friend.