# R in Rangelands Wkshp

Georgia Harrison, Leah Dreesmann, Claire Tortorelli

2023-01-31

## Welcome to Putting the R in Rangelands!

**Access the data and interactive code via github**

## Part 1: Getting started with R

## Before you begin:

### Install R and RStudio

As of January 20, 2023, the current R version is 4.2.2 and the current RStudio version is 2022.12.0+353. You should download current versions of both these programs and install them on your computer if you haven't already. You can download R from CRAN and the free version of RStudio from their site.

## Why use R?

- Free
- open source, always growing an evolving
- tons of packages
- great for data analysis, statistics and data visualization

## R vs RStudio?

- **R** is the software that performs instructions.
- **RStudio** is an interface to interact with R.

### RStudio

Let's take a tour through R studio * **Console** pane - contains the output push enter to run for example, run 4 + 4 output is in brackets on the next line [8] * **Source** pane - for working with scripts you want to save All files end in .R Run section using RUN button, or Ctrl+Enter (or Cmd+Enter for Macs) * Also take look at the environment & files panes.

## Quick terminology

- **Package** base R is bare bones. Packages are like apps that other people have made that you can download and use. Some are standard for most R users, others are subject specific
- **Directory** The working directory is your home base for this R session.
- **Environment** your current workspace. This includes any files you have read into R, and data tables you created, etc

## What is up with all the hashtags

```
## these are comments and notes
# anything following a hashtag is a comment. This is just text
# use this for notes within code, troubleshooting
```

## R scripts vs R Markdown (RMD)?

- In regular **R scripts**, outputs are outside of the script: in the console, plots or other windows. These files end with .R
- In **R Markdown**, the script is broken into text and code sections, called chunks. You can format the text (hence all the asterisks), and outputs are spit out under code chunks. To run a chunk of code, hit the green triangle in each chunk, or run all chunks using the upper Run arrow. These files end with .Rmd

## The Tidyverse

This workshop will be focused on using tools within the tidyverse for data manipulation and visualization. The tidyverse is a collection of packages that all play nice with eachother - they have the same philosophy, grammar, and data structures. This allows for fast, efficient workflows in R. The world of R is moving to the tidyverse, and you should too!

### Packages within the tidyverse:

- Importing data (readr)
- Data manipulation (dplyr, tidyr)
- Working with data types (stringr for strings, lubridate for date/datetime, forcats for categorical/factors)
- Data visualization (ggplot2)
- Data-oriented programming (purrr)
- Communication (Rmarkdown, shiny)

## Install packages

There are a few ways you can install a package:

1) In packages tab, using the install button. Then library a package (aka activate it for that session) by checking the box

2) Using install.packages("packagename")

```
# install.packages("tidyverse") #run this! only once, good practice to run in the console
```

**note, only do this once!** After installation, library a package to recall it and have it ready to use during that session

```
library(tidyverse)
```

**the tidyverse is special because it is actually a family of packages**

# Help

Here are a few of the key ways to get help in R. 1. **Help function**: to pull up the documentation for a package or function: ?package ?functionname

for example:

```
?readr
```

```
## starting httpd help server ... done
```

```
?mutate
```

These bring you to **R documentation**, which is always set up in the same way with these sections: * Description * Usage * Arguments * Examples or Details

2. **Stack overflow**: R questions and answers in most cases, someone else has already asked your question

There are cheatsheets for each tidyverse package (and many other R entities!)

# Working Directory

This is where R, by default, will go to look for any datasets you load and is the place R will save files you save. When working on a simple project, I save my R scripts and all files related to that project into a single folder that I set as my working directory. This makes it so I don't have to write out the whole directory path every time I want to load or save something. This also helps me keep organized.

You can check the current working directory:

```
getwd()
```

```
## [1] "C:/Users/harr4718/Documents/GitHub/RinRangelands"
```

You can do this with code of with the control panels: Session > set working directory > to source file location > navigate to your home base folder on your computer OR Files pane > More.. > Set working directory

## Let's all set the working directory for this workshop

Important: You must either use single forward slashes or double backslashes in the directory path in R instead of the single backslashes. If you work in Windows this will not be what you are used to. setwd("C:/Users/Aosmith/R workshops/r-basics-workshop") setwd("C:\Users\Aosmith\R workshops\r-basics-workshop")

```
setwd("C:\\Users\\harr4718\\OneDrive - University of Idaho\\R code (copies)\\SRM R workshop\\DemoData")
#customize to your own file path
```

## Read in the data

let's get started actually working with the data First step is to import the data using readr (a package within the tidyverse) using the *read_csv()* function

```
cover_data = read_csv("DemoData/cover_data.csv")
```

```
## Warning in gzfile(file, mode): cannot open compressed file 'C:/Users/harr4718/
## AppData/Local/Temp/RtmpkBSQ3K\file558070c173dd', probable reason 'No such file
## or directory'
```

```
## Rows: 1620 Columns: 4
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): PrimaryKey, LineKey, cover_type
## dbl (1): percent
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
#also read in the plot metadata
meta_data = read_csv("DemoData/meta_data.csv")
```

```
## Rows: 180 Columns: 7
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (6): PrimaryKey, LineKey, PlotID, Line, Trt, SH_GR
## dbl (1): Yr
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

# Part 2: Cleaning Data

### Take a peek at our data

The **head** function allows you to view the first 10 rows of data

```
head(cover_data)
```

```
## # A tibble: 6 x 4
##   PrimaryKey LineKey      cover_type percent
##   <chr>      <chr>        <chr>        <dbl>
## 1 101C_2021  101C_15_2021 AF_I          5
## 2 101C_2021  101C_15_2021 AF_N          1.67
## 3 101C_2021  101C_15_2021 AG_I          5
## 4 101C_2021  101C_15_2021 PF_N          5
## 5 101C_2021  101C_15_2021 PG_I          0
## 6 101C_2021  101C_15_2021 PG_N         23.3
```

```
head(meta_data)
```

```
## # A tibble: 6 x 7
##   PrimaryKey LineKey      PlotID Line       Yr Trt   SH_GR
##   <chr>      <chr>        <chr>  <chr>   <dbl> <chr> <chr>
## 1 101C_2021  101C_15_2021 101C   101C_15  2021 C     HSHG
## 2 101C_2021  101C_2_2021  101C   101C_2   2021 C     HSHG
## 3 101C_2021  101C_28_2021 101C   101C_28  2021 C     HSHG
## 4 101T_2021  101T_15_2021 101T   101T_15  2021 T     HSHG
## 5 101T_2021  101T_2_2021  101T   101T_2   2021 T     HSHG
## 6 101T_2021  101T_28_2021 101T   101T_28  2021 T     HSHG
```

# Cleaning Data

After a data set has been loaded into RStudio, the next step is getting it ready for analysis. The tidyverse has many functions in the **tidyr** and **dplyr** packages which allow you to tidy and transform your data set so you can use it for analysis.

# Tidy Data

It is important to have tidy data to work with. Tidy data sets meet three interrelated rules:
1. Each variable must have its own column
2. Each observation must have its own row
3. Each value must have its own cell

Our meta_data does meet these rules because each variable has its own column, all observations have its own row, and each value has its own cell. However, our cover_data does not meet these rules because observation of interest (LineKey) has multiple rows and each variable of interest (cover_type) has only one column. To make this tidy data we want to create a wide data set. We do this with the function pivot_wider().

```
cover_data = cover_data %>% pivot_wider(names_from = "cover_type", values_from = "percent")
```

To convert the data back to a long data set we use the function pivot_longer().

```
cover_data_long = cover_data %>% pivot_longer(c("AF_I", "AF_N", "AG_I", "PF_N", "PG_I", "PG_N", "SH_N",
```

# Join data

We have two different data sets that we want to work with, cover_data and meta_data. The meta_data provides more information about each line on which cover_data was collected. To easily work with this data, we should combine these data sets into one big data set. We do this by using mutating joins based on keys in the data.

## Keys

Keys are how we uniquely identify observations in tables and allow us to link observations between tables. There are two types of keys:
* Primary key: Uniquely identifies an observation in own table * Foreign key: Uniquely identifies an observation in another table

In our tables the keys we have are "PrimaryKey", which links to each plot, and "LineKey", which links to each line in a plot.

## Mutating Joins

The functions we use to join two or more data sets are called joins. There are several types of joins based on what you are trying to do:
* inner_join(): keeps observations that appear in both tables, unmatched rows excluded
* full_join(): keeps observations that appear in both tables, all unmatched rows included
* left_join(): keeps observations that appear in the 'left' table
* right_join(): keeps observations that appear in the 'right' table

For our data sets, we have no unmatched rows so the use of any of these functions will results in the table. Here is an example using the inner_join (note: the number of rows in our new table is still 180).

```
data = meta_data %>% inner_join(cover_data, by = c("PrimaryKey", "LineKey"))
```

# Manipulating Columns

Sometimes we want to change something about the columns in our data sets. We may want to choose only a subset of the variables, change a variable that we already have in our data set, or add a new variable to our data set.

## Subsetting Variables

To choose a subset of variables we can use the select() function. In the tidyverse you can either write out the column name or the position in the tibble to signify which columns you want to be selected.

Here is an example with our data set, because we are interested in keeping all of our columns, there is code to remove the new data set from our global environment so we aren't confused later.

```
select_columns = data %>% select(1:4, GAP)
#remove from global environment
rm(select_columns)
```

## Changing or adding variables

To change or add variables we can use the following functions:
* mutate(): adds the new variable onto the end of existing tibble. * transmute(): adds only the specified variables to the tibble.

There are many different uses of these functions and a few key situations you may run across are described here.

### Data Types

Data is stored as different types in R tibbles. There are many different types we may be interested in (https://tibble.tidyverse.org/articles/types.html). Here are a few of the more important ones:
* Logical (lgl): TRUE or FALSE
* Double (dbl): all real numbers (with or without decimal places)
* Character (chr): strings (we would code these in " ")
* Factor (fct): vector with set not ordered numeric codes to predefined character valued levels
* Date (date): date variable

There are many ways we can figure out how R is storing each variable including the output when first loading in data, the drop down arrow in the environment, the table from the function head() output, and the glimpse() function.

Sometimes R does not correctly guess what type of data you have, so we need to change it to avoid our analysis getting messed up. We can use the mutate() or transmute() function to change variable types in combination with data type changing functions a few are here:
* logical: as.logical()
* double: as.numerical()
* character: as.character()
* factor: as.factor()
* date: as.date()

Based on my knowledge of our data set, I want to change Yr to a factor with the levels "2021" and "2022" and Trt to a factor with the levels "Control" and "Treatment". (note: I also had to change the level names in Trt from "C" and "T" to "Control" and "Treatment" which I did with the recode() function).

```
data = data %>% mutate(Yr = as.factor(Yr), Trt = as.factor(Trt), Trt = recode(Trt,"C" = "Control", "T" =
```

### Adding Variables

Another common use of these functions is to add new column. We can add a new variable of a constant or derived from the other columns.

In this example, I can get the cover on each line that is not a 'gap', which will tell us if there is some cover.

```
data = data %>% mutate(NOT_GAP = (100 - GAP))
```

This new variable is added to the end of the table, if we were to use transmute() instead of mutate() it only adds the variables we specify. We also don't need this table and will remove it from the global environment.

```
not_gap = data %>% transmute(PrimaryKey, LineKey, GAP, NOT_GAP = (100 - GAP))
#remove from global environment
rm(not_gap)
```

# Manipulating Rows

Sometimes we want to change something about the rows, like rearranging them in the tibble or subsetting to only rows with specific values.

## Arranging Rows

We can use the function arrange() to change the order of the rows based on specific values.

In this example, we can arrange our rows by year, then by treatment type, then by NOT_GAP cover.

```
data = data %>% arrange(Yr, Trt, NOT_GAP)
```

## Subsetting rows

We can use the function filter() to subset the rows based on specific values.

You can filter by any of the different data types and can use many different comparison operators:
>, >= (greater than)
<, <= (less than)
!= (not equal)
== (equal)

You can also multiple combinations using Boolean operators:
& (and)
| (or)
! (not)

In this example, we are going to filter to lines with a control plot and gap values greater than 25%.

```
control_AG = data %>% filter(Trt == "Control", AG_I >= 25)
#remove from global environment
rm(control_AG)
```

# Dealing with NA values

If you have NA values in your data set it is important to carefully consider what they mean before you do anything with them. Are the NA values signifying true missing information? Are they actually signifying a 0? Do you only have a few NA's and removal of those observations won't impact analysis? Do you have a lot of NA's and you need to figure out a way to deal with them?
If you are dealing with a data set with NA values, you should [spend more time looking into this topic] (https://towardsdatascience.com/data-cleaning-with-r-and-the-tidyverse-detecting-missing-values-ea23c519bc62)

In this data set, from exploration, I know that the only NA values are in the variable PF_I (invasive perennial forbs). To identify which rows have NA values in the data set we can use the filter() function.

```
na_rows = data %>% filter(is.na(PF_I))
#remove from global environment
rm(na_rows)
```

From my knowledge of how this data was generated, I know that these NA's actually signify 0 values. In 2021, there was no invasive perennial forbs found so the package used to generate this summary information did not output a column for it, while in 2022 there was invasive perennial forbs found so the analysis did generate an output for it.

With this knowledge we now want to replace the NA values with 0's.

```
data = data %>% mutate(PF_I = replace(PF_I, is.na(PF_I), 0))
```

## All Together

Don't forget that you can pipe many of these functions together. For fun, here is a way to get the same data set with just one long pipe.

```
data_all = meta_data %>%
  inner_join(cover_data_long, by = c("PrimaryKey","LineKey")) %>%
  pivot_wider(names_from = "cover_type", values_from = "percent") %>%
  mutate(Yr = as.factor(Yr), Trt = as.factor(Trt), Trt = recode(Trt,"C" = "Control", "T" = "Treatment")
  arrange(Trt, Yr, NOT_GAP)
```

## Summaries

Now that we have tidy and cleaned data we can do some analysis with it. One of the most basic things we may want to do is summarize the data. The function to do this is summarise(). We often pair this function with the function group_by() which allows us to get grouped summaries.

For our data, lets see if there is a difference in annual invasive grass from in our control and treatment plots between 2021 and 2022.

```
invasive_change = data %>%
                    group_by(Trt, Yr) %>%
                    summarise(cover_avg = mean(AG_I))
```

```
## `summarise()` has grouped output by 'Trt'. You can override using the `.groups`
## argument.
```

## Exercises

1. Create a new data set with the variables PrimaryKey, LineKey,Trt, Yr, and the annual species cover (those starting with A). Include only the observations that have more than 25% invasive annual grass cover (AG_I).
2. Create a summary table of median percent cover for invasive annual grass for each year and treatment also include the number of observations in each group (hint: to get counts you can simply use n() ).
3. Convert your new data set into a long data set.

### Save the data in final form for vizualizations

```
final_data <- data
```

# Part 3: Data Vizualizations

Read in data

```
load(file = 'DemoData/final_data.rda') #stored within the DemoData file

# do this manually
dat = final_data
```

# Graphics with ggplot2

## Why GGplot2?

ggplot2 is a popular R package for producing data and statistical graphics. It is based on the "Grammer of Graphics" which allows you to create graphs by combining multiple independent components. In this way, ggplot2 is more powerful and useful for customizing graphics to your needs.

ggplot2 works iteratively by layering data, plot types, and aesthetics onto our graphics. All plots start with the function **ggplot()**, include *data* (what we want to plot), and a *mapping* (how we want to visualize the data).

Within the **mapping**, we can add our graphical layers or **geom**, that tell ggplot how to display our data (e.g. **geom_point**, **geom_line**, **geom_bar**). Within our **geom**, we can add the aesthetic function **aes()** to assign data to the geom, change color, size, fill, or transparancy (alpha).

We can also change the aesthetics of our graph by **scale**ing our values to different colors, sizes, or shapes. This is also where our legends come from. We can add a **theme** to our plots to control additional aesthetics like font size.

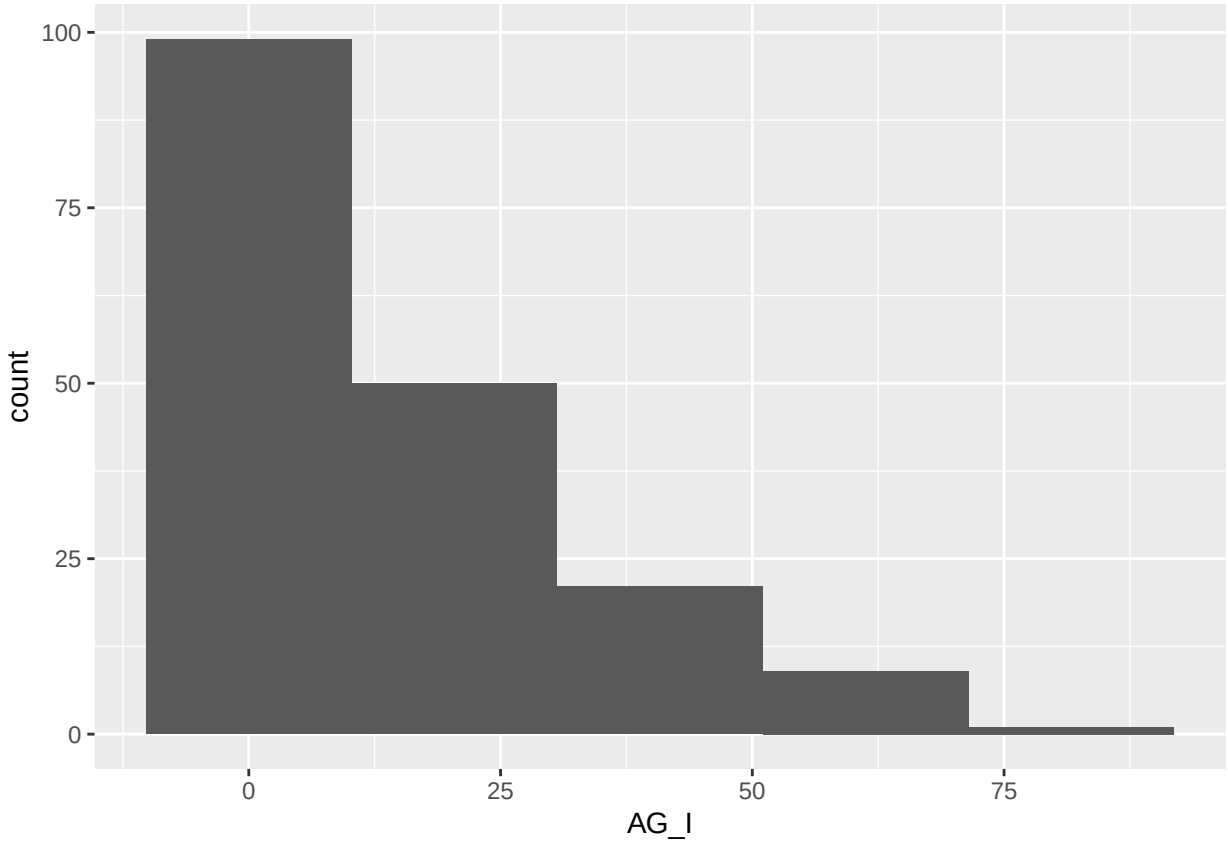
**Facet**ing allows us to break up and display our data by different groups

## Histogram

Let's start out with a basic histogram to get a feel for our response variable, invasive annual grasses (AG_I)

We can call the variable of interest and ggplot will count up the number of observations that fall within a pre-defined number of "bins".

```
#we can pipe in ggplot2!
final_data %>% #pipe in our data here
  ggplot() + #then call ggplot and add on our mapping and aesthetics
  geom_histogram(aes(x = AG_I), bins = 5) #our variable of interest goes on the x axis and we can assig
```

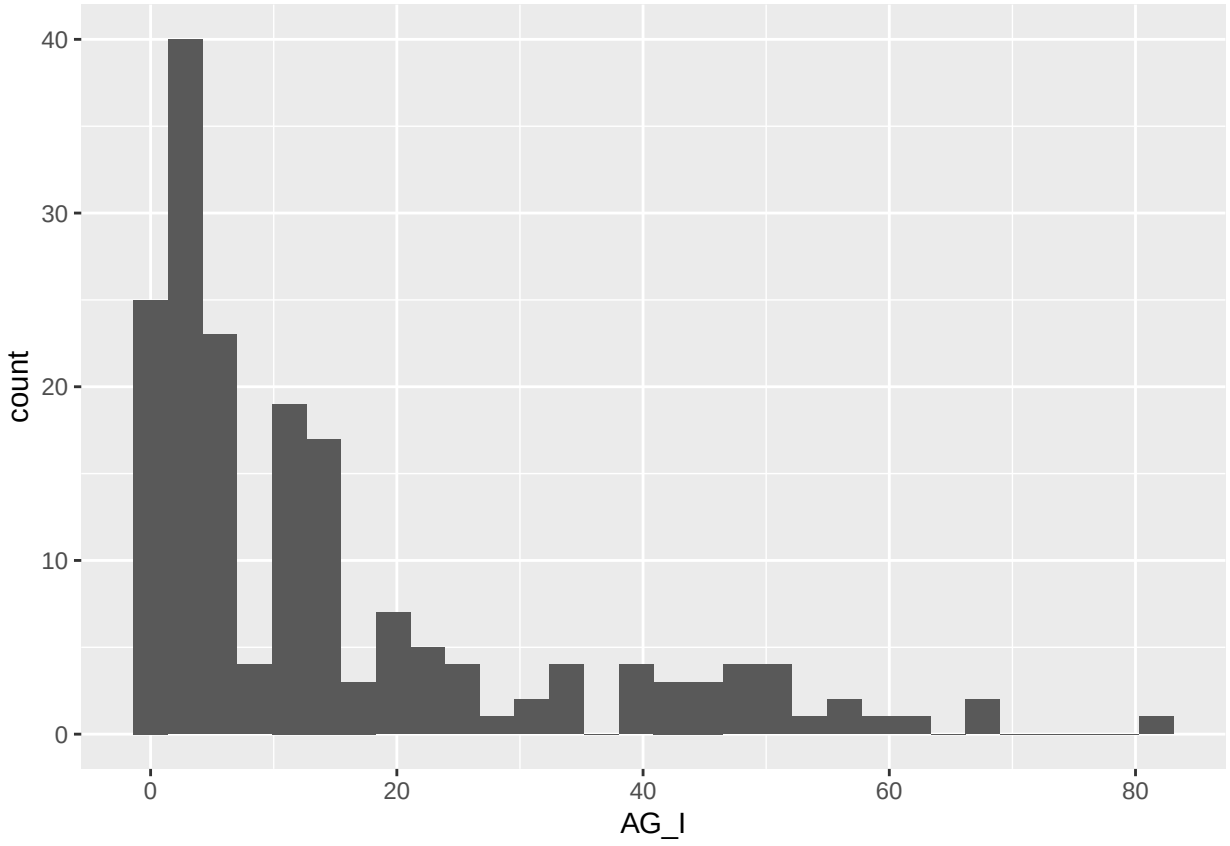



```
#these bins are a little too wide to get a good feel for our data distribution
```

### Changing bin size

```
#change the bin size to 10
final_data %>%
  ggplot() +
  geom_histogram(aes(x = AG_I), bins = )
```

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

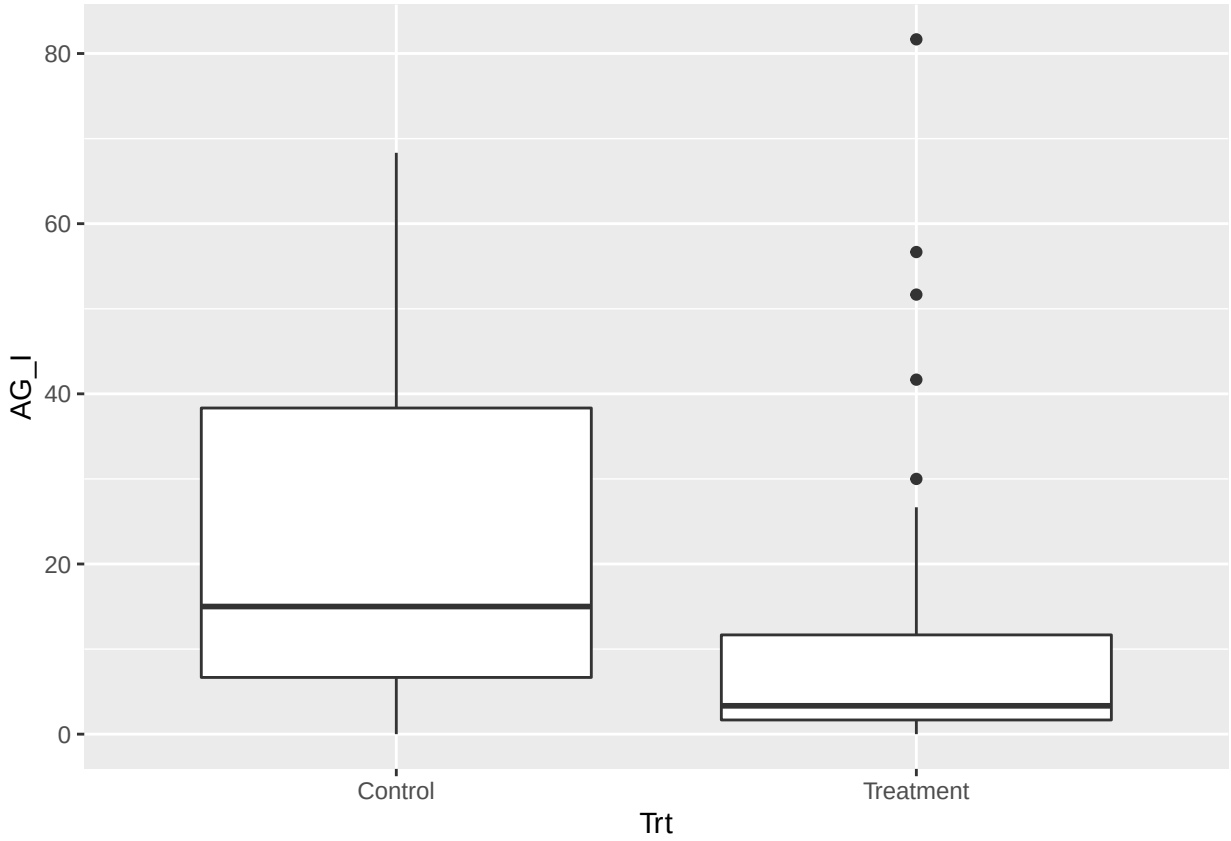



```
#Looks like we have some 0 inflated data!
#We would want to keep that in mind for any statistical analyses.
```

## Box plot

Now let's try a basic box plot to visualize how invasive annual grasses vary by our categorical treatment variable Box plots provide a little more information about our variable including the quantiles and outliers.

```
final_data %>%
  ggplot() +
  geom_boxplot(aes(x = Trt, y = AG_I)) #instead of a histogram, we'll call a box plot with x as our tre
```



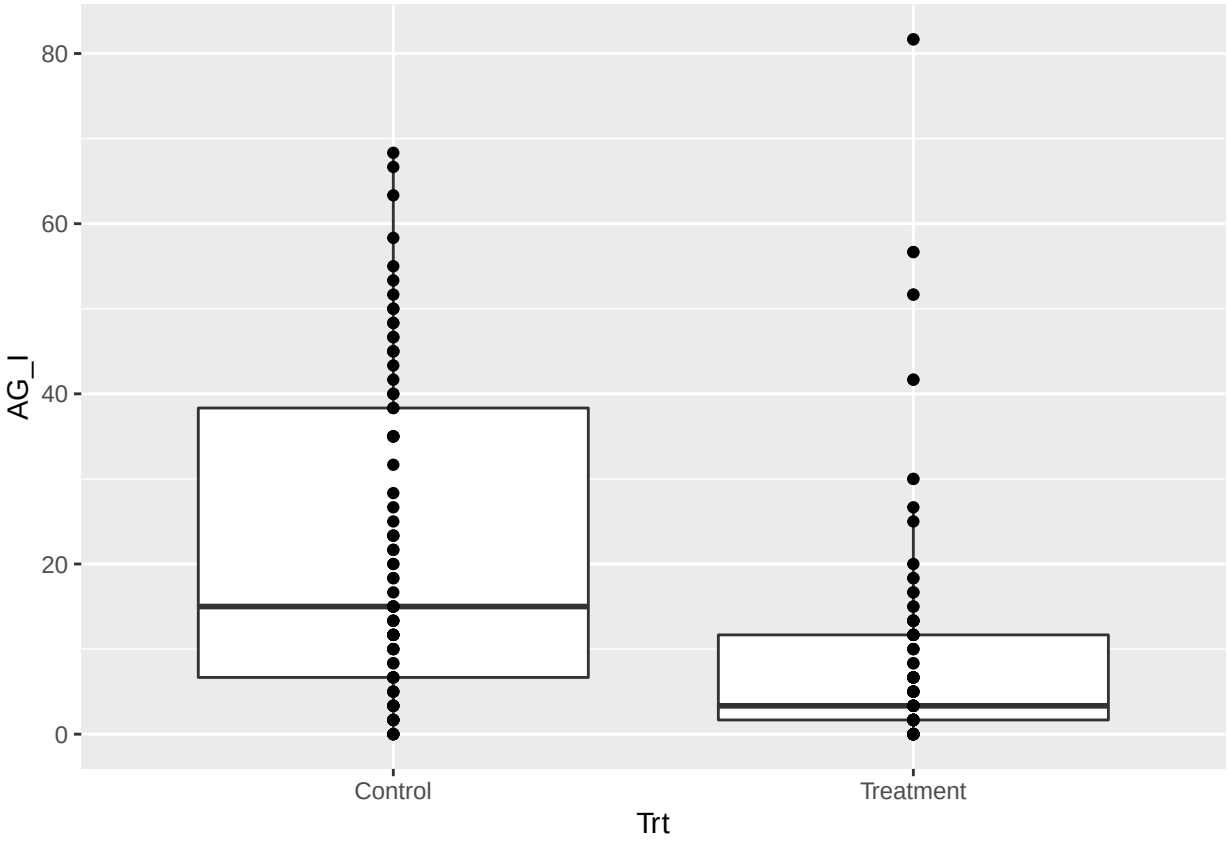

```
#neat! Looks like annual grass cover may vary by treatment.
```

## Layering plot types

What if we want to see the raw data displayed on top of the box plot? We can layer mappings by adding geoms to our existing plot.

### Adding points

```
final_data %>%
  ggplot() +
  geom_boxplot(aes(x = Trt, y = AG_I)) +
  geom_point(aes(x = Trt, y = AG_I)) #add raw data as points
```

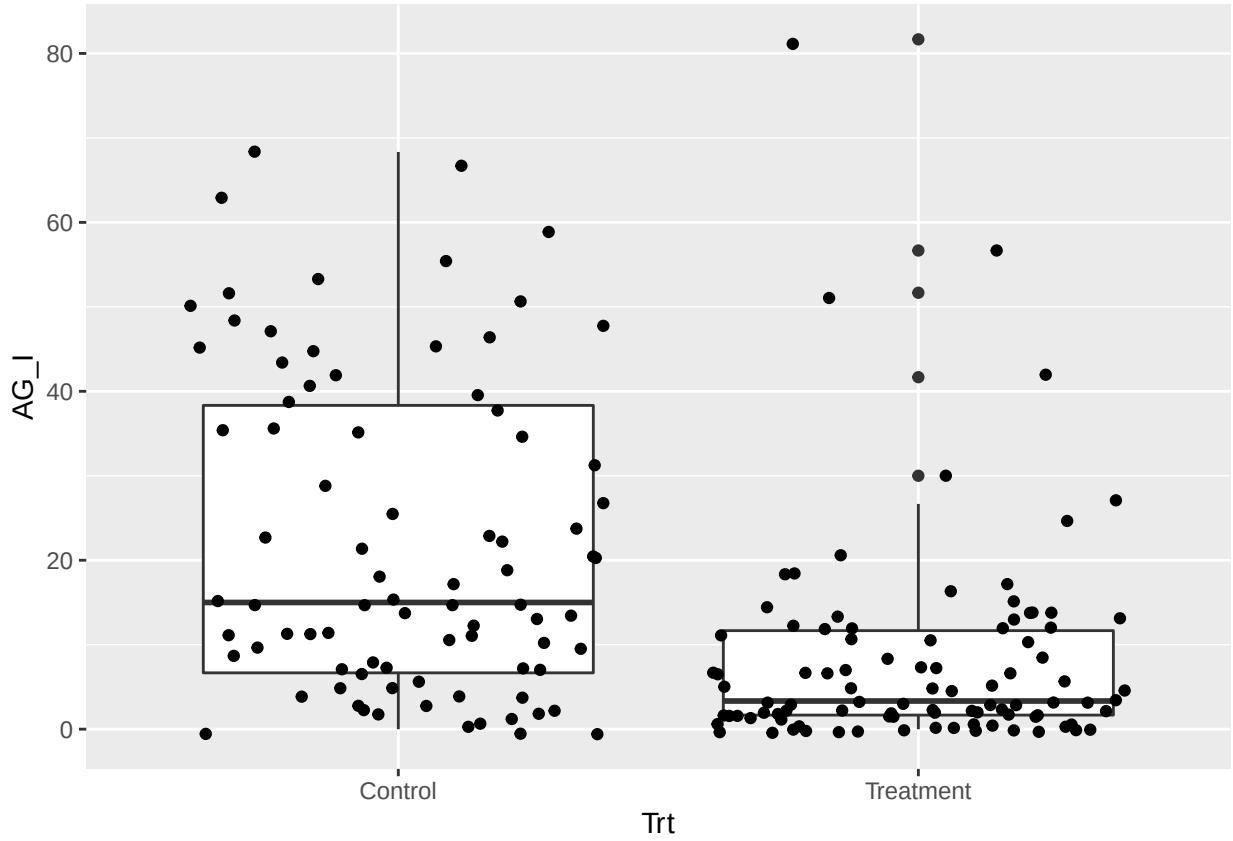



```
#those points are difficult to see when they're all crowded on top of each other, let's try out geom_ji
```

### Jittering points

Jittering points allows us better visualize overlapping points.

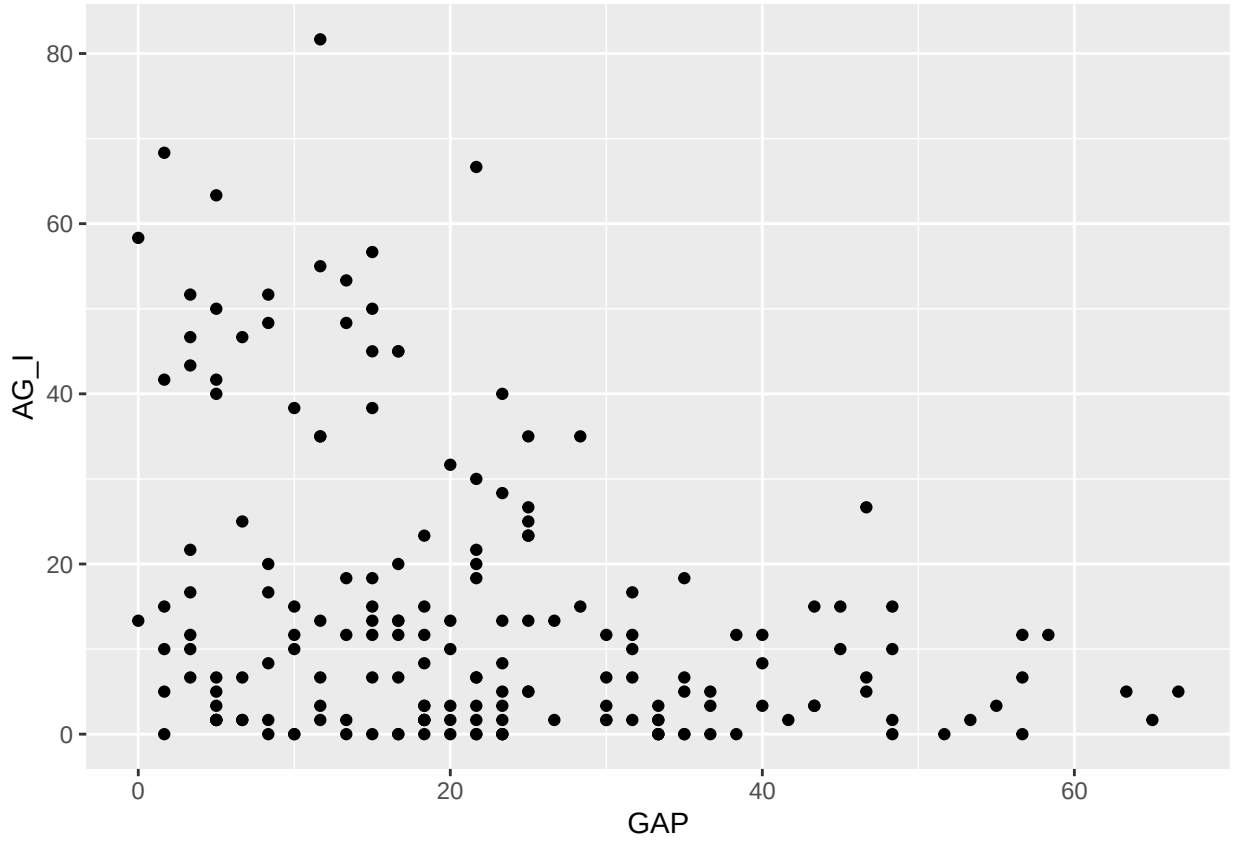
```
final_data %>%
  ggplot() +
  geom_boxplot(aes(x = Trt, y = AG_I)) +
  geom_jitter(aes(x = Trt, y = AG_I)) #add jittered points
```




## Scatter Plot

We can also use geom_point to create scatter plots to visualize relationships between two continues variables. In this case invasive annual grasses vs. gap size

```
final_data %>%
ggplot() +
  geom_point(aes(x = GAP, y = AG_I)) #now we have GAP as our X variable and AG_I as our y variable
```




## Pretty graphics!

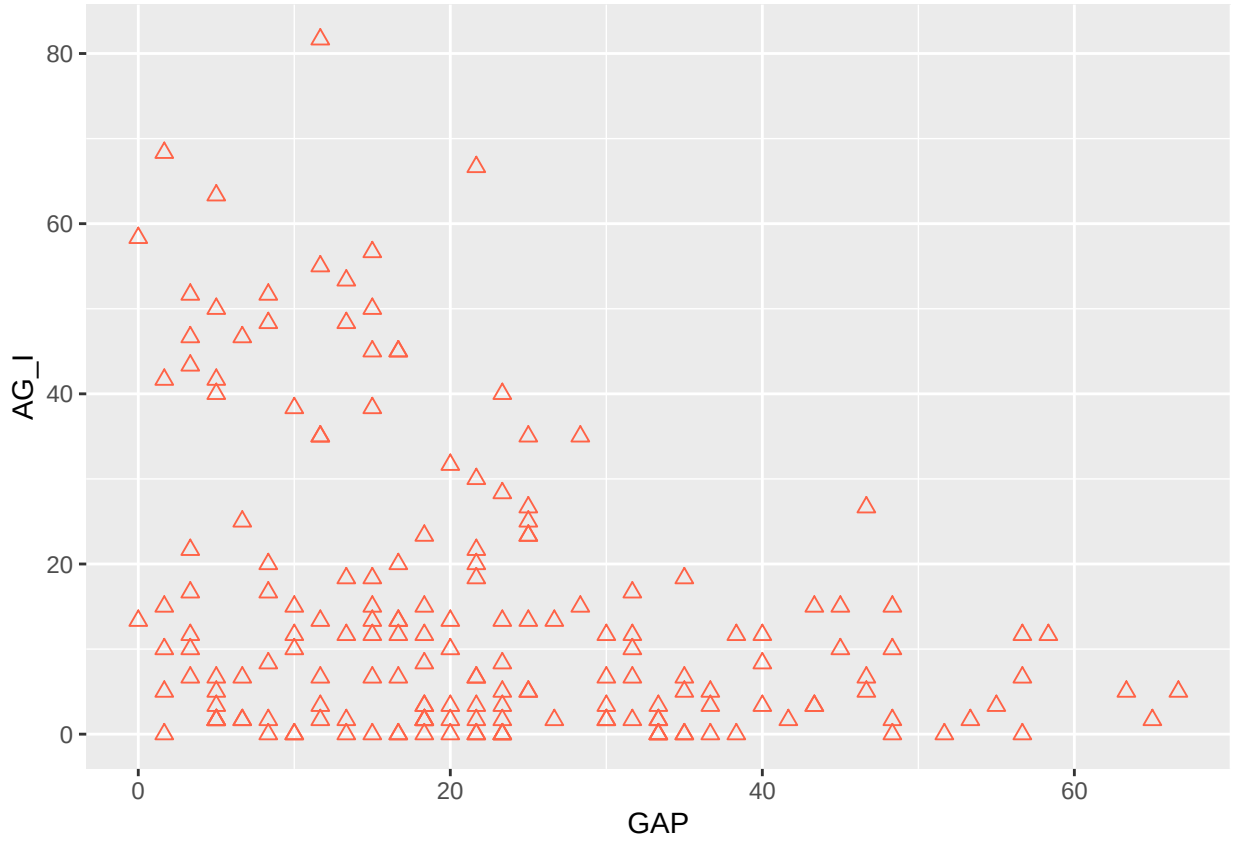
### Changing colors and shapes

Let's make it a little prettier by changing the color, size, and shape of the points

Some colors can be spelled out in ggplot like 'red', 'blue', 'yellow', others can be called using a code. Here's a great resources for selecting colors in r

We can also change the size and shape of points by assigning values to *size* = and *pch* = Here is a resource for assigning shapes (pch) to points

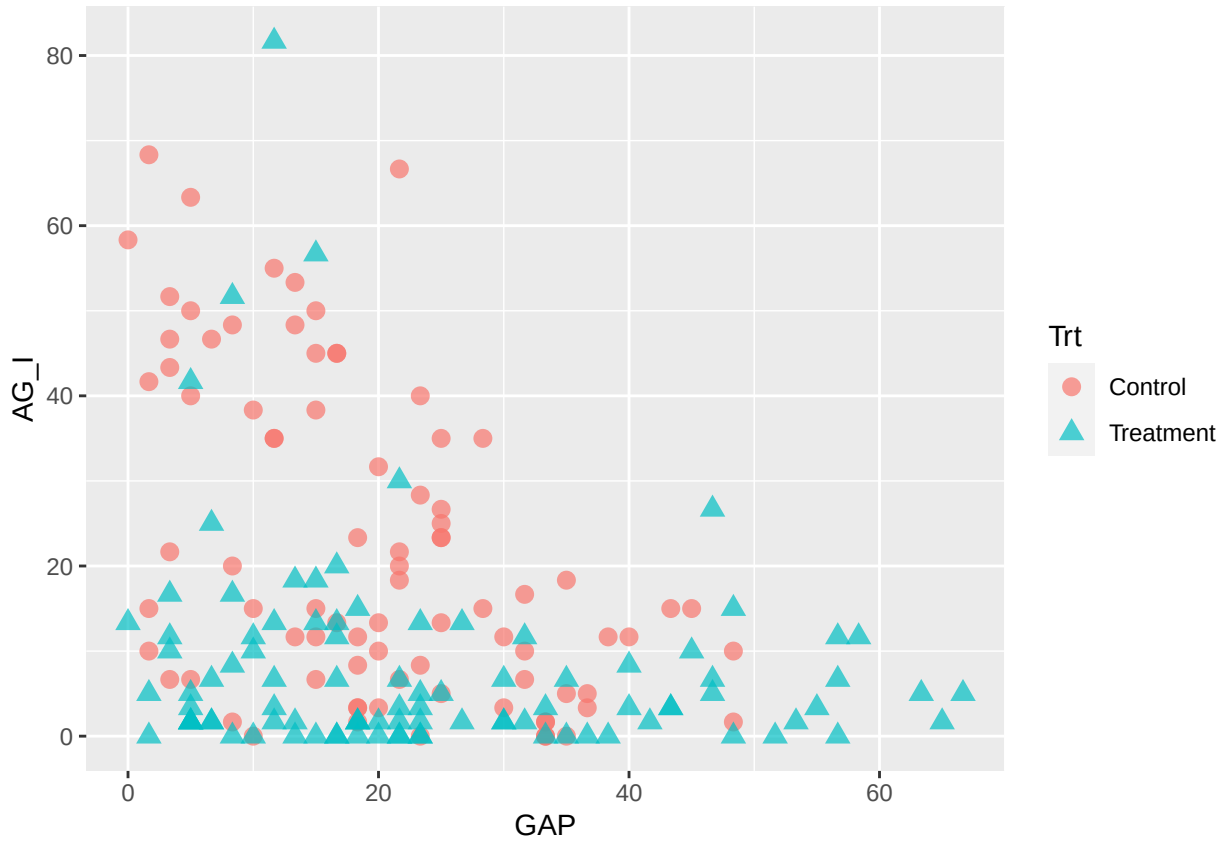
```
final_data %>%
  ggplot() +
  geom_point(aes(x = GAP, y = AG_I),
             color = 'tomato', size = 2, pch = 2) #color, size of points, pch = shape of point can go o
```




We can also change point properties based on other variables in our plots.

Here we'll change the color and shape of the points to reflect which treatment they correspond with. To do this, we'll add *color* = and *shape* = elements to our geom_point within the *aes()* call and assign them both to *Trt*.

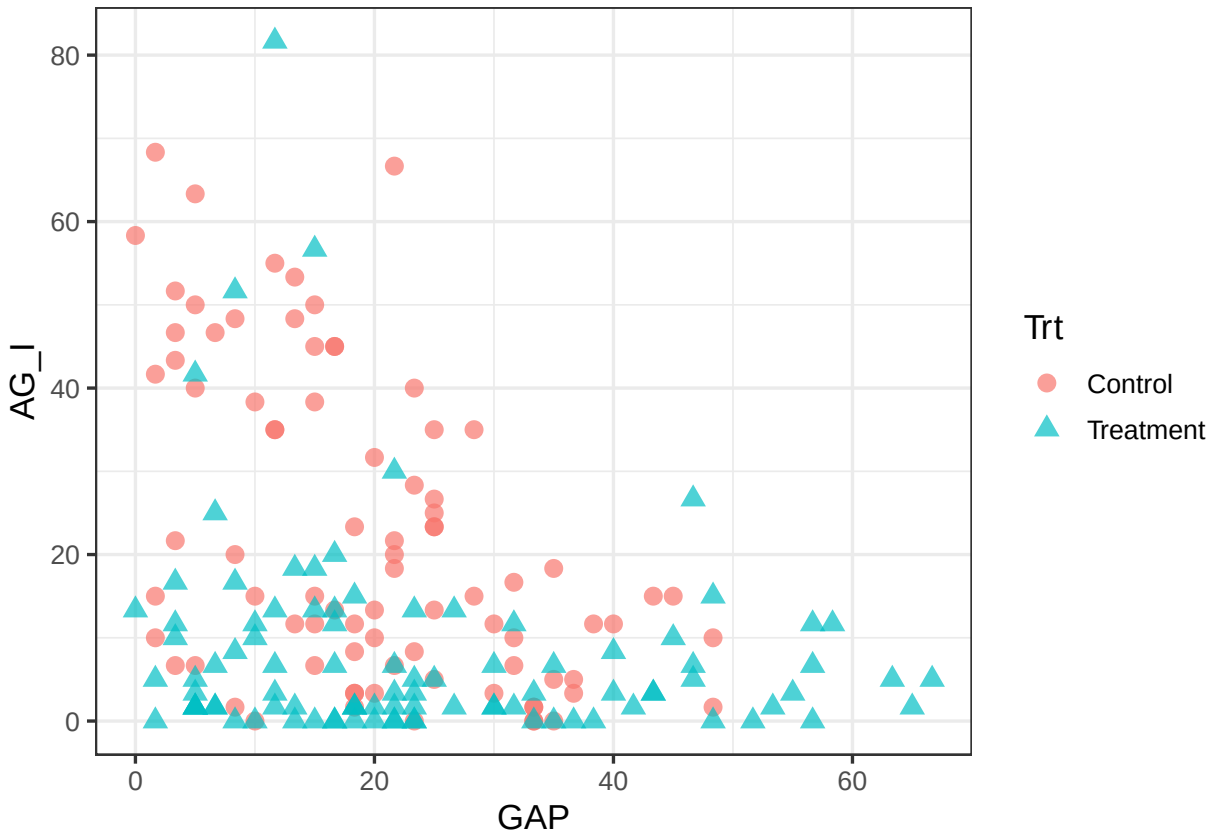
```
final_data %>%
  ggplot() +
  geom_point(aes(x = GAP, y = AG_I, color = Trt, shape = Trt), #change color and shape by treatment
             size = 3, alpha = 0.7) # make  points semi transparent to help with overlap
```




**Themes!**

ggplot2 has pre-set themes that change multiple aspects of our graphics at once. Read more about themes here

```
final_data %>%
  ggplot() +
  geom_point(aes(x = GAP, y = AG_I, color = Trt, shape = Trt),
             size = 3, alpha = 0.7) +
  theme_bw(13) #add a theme and base size to change the background color and font
```




Practice time! See what happens when you increase or decrease alpha and size. You can also try changing to a different theme: theme_classic, theme_dark, etc.

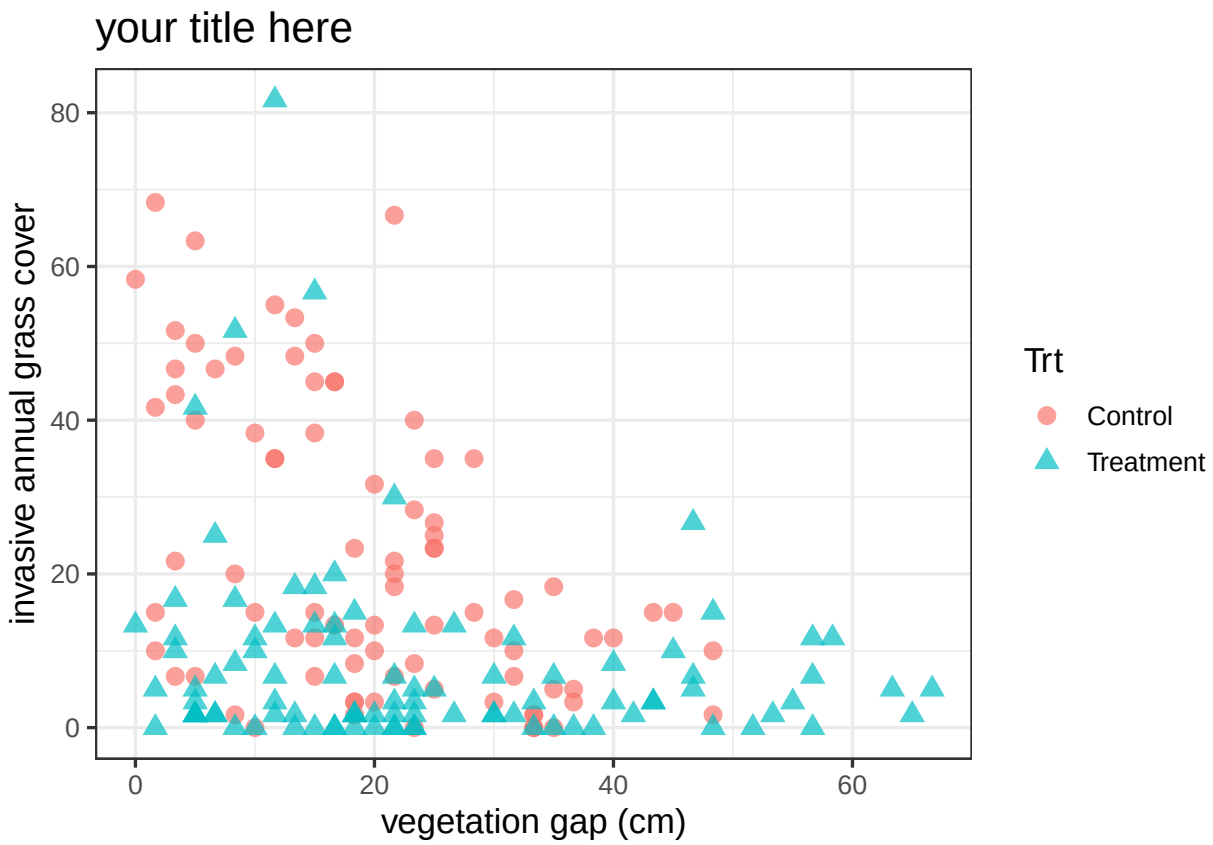
```
# remove the hashtags from every line


# final_data %>%
#   ggplot() +
#   geom_point(aes(x = GAP, y = AG_I, color = Trt, shape = Trt),
#              size = , alpha = ) +
#   theme_bw(13)
```

### Adding Labels

ggplot2 makes it easy to add custom axis labels and titles to our plots. Give it a try!

```
final_data %>%
  ggplot() +
  geom_point(aes(x = GAP, y = AG_I, color = Trt, shape = Trt),
             size = 3, alpha = 0.7) +
  theme_bw(13)+
  ggtitle("your title here") + #add title
  xlab("vegetation gap (cm)") + #add x label
  ylab("invasive annual grass cover") #add y label
```



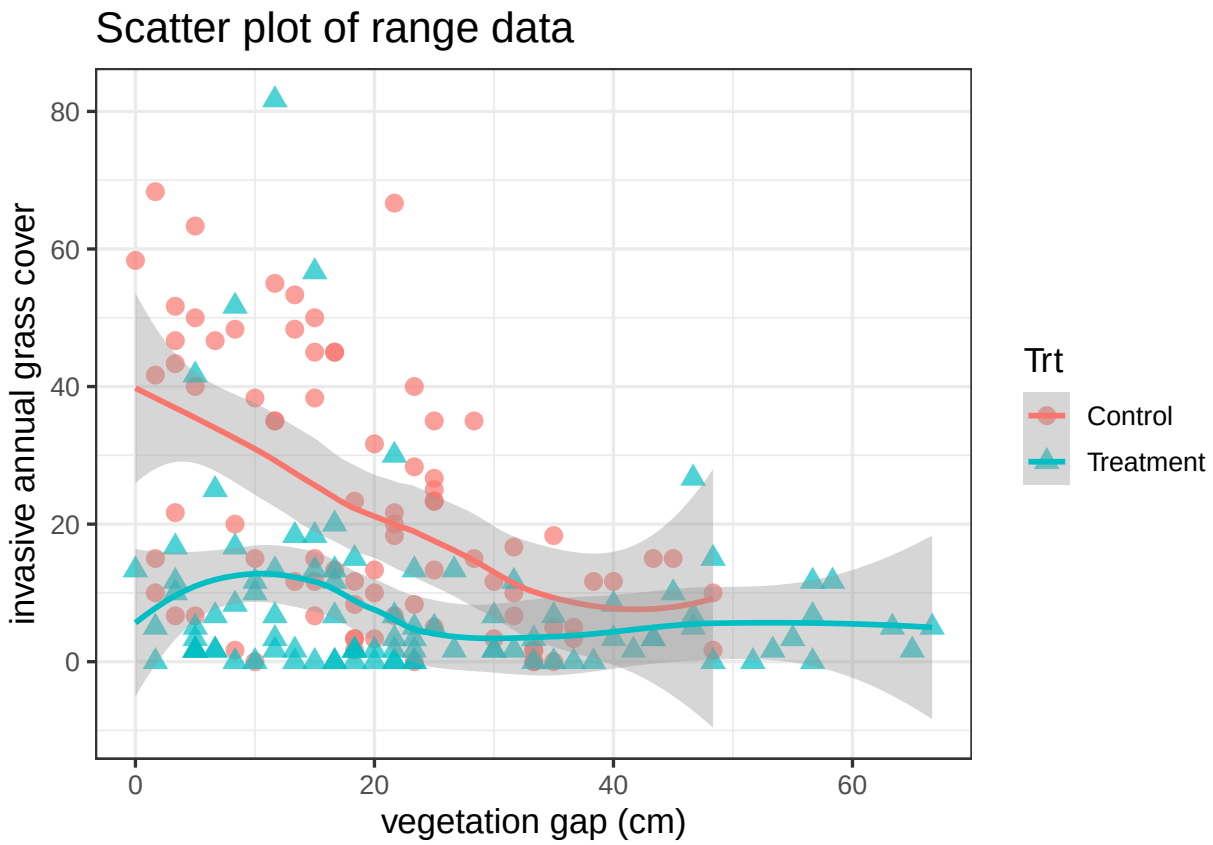

### Adding trend lines

The geom_smooth() function allows us to add a trend line over an existing plot. The default smoother is a loess smooth line, but we can change the type of fit to a straight fit or wigglier fit if we want (e.g. method = "lm").

*Remember, this is for data visualization/exploration purposes, and model assumptions should be checked before using trend lines to display statistical results.

```
final_data %>%
  ggplot() +
  geom_point(aes(x = GAP, y = AG_I, color = Trt, shape = Trt),
             size = 3, alpha = 0.7)+
  geom_smooth(aes(x = GAP, y = AG_I, color = Trt))+ #add a smoothed term that is separated by treatment
  theme_bw(13)+
  ggtitle("Scatter plot of range data") +
  xlab("vegetation gap (cm)") +
  ylab("invasive annual grass cover")
```

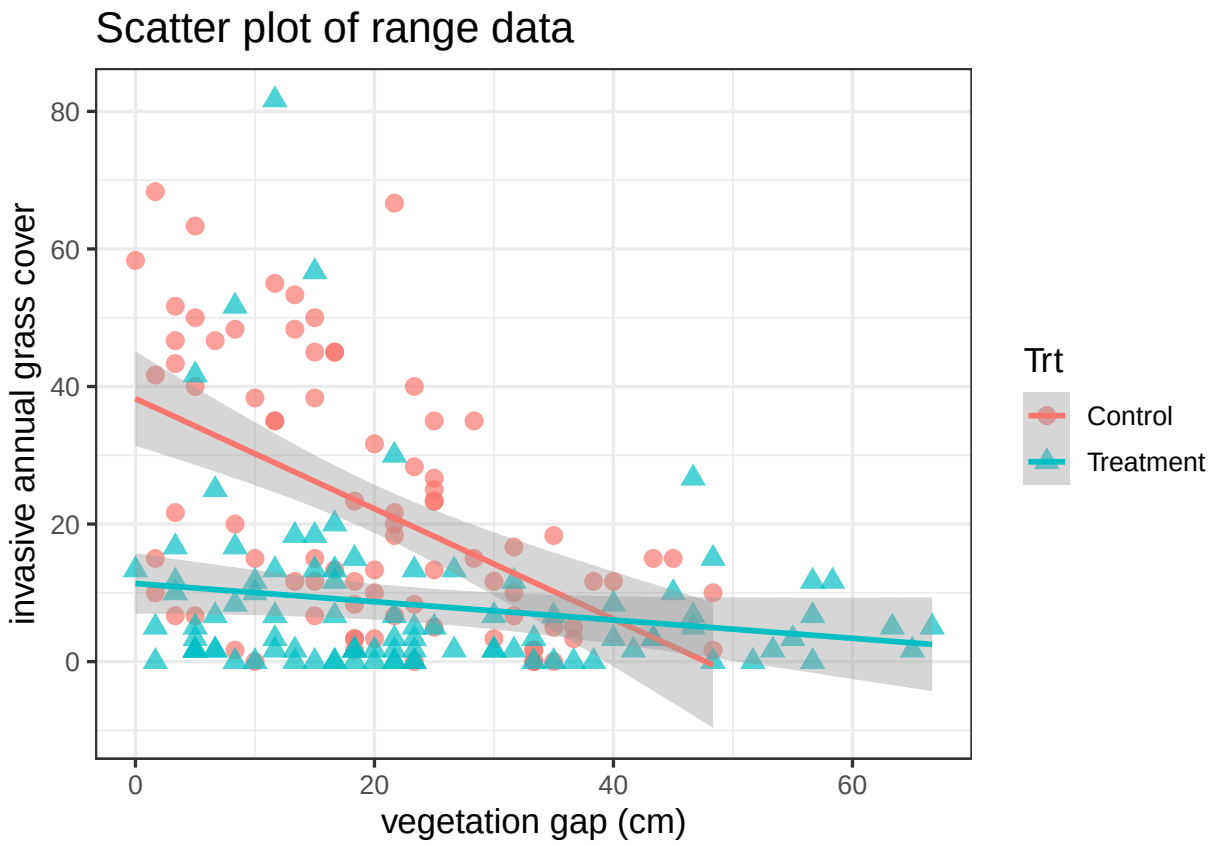
```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




Now let's see what it looks like if we change the line type to "lm" and fit the data with a straight line.

```
final_data %>%
  ggplot() +
  geom_point(aes(x = GAP, y = AG_I, color = Trt, shape = Trt),
             size = 3, alpha = 0.7)+
  geom_smooth(aes(x = GAP, y = AG_I, color = Trt), method = "lm")+ #changing method to "lm"
  theme_bw(13)+
  ggtitle("Scatter plot of range data") +
  xlab("vegetation gap (cm)") +
  ylab("invasive annual grass cover")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



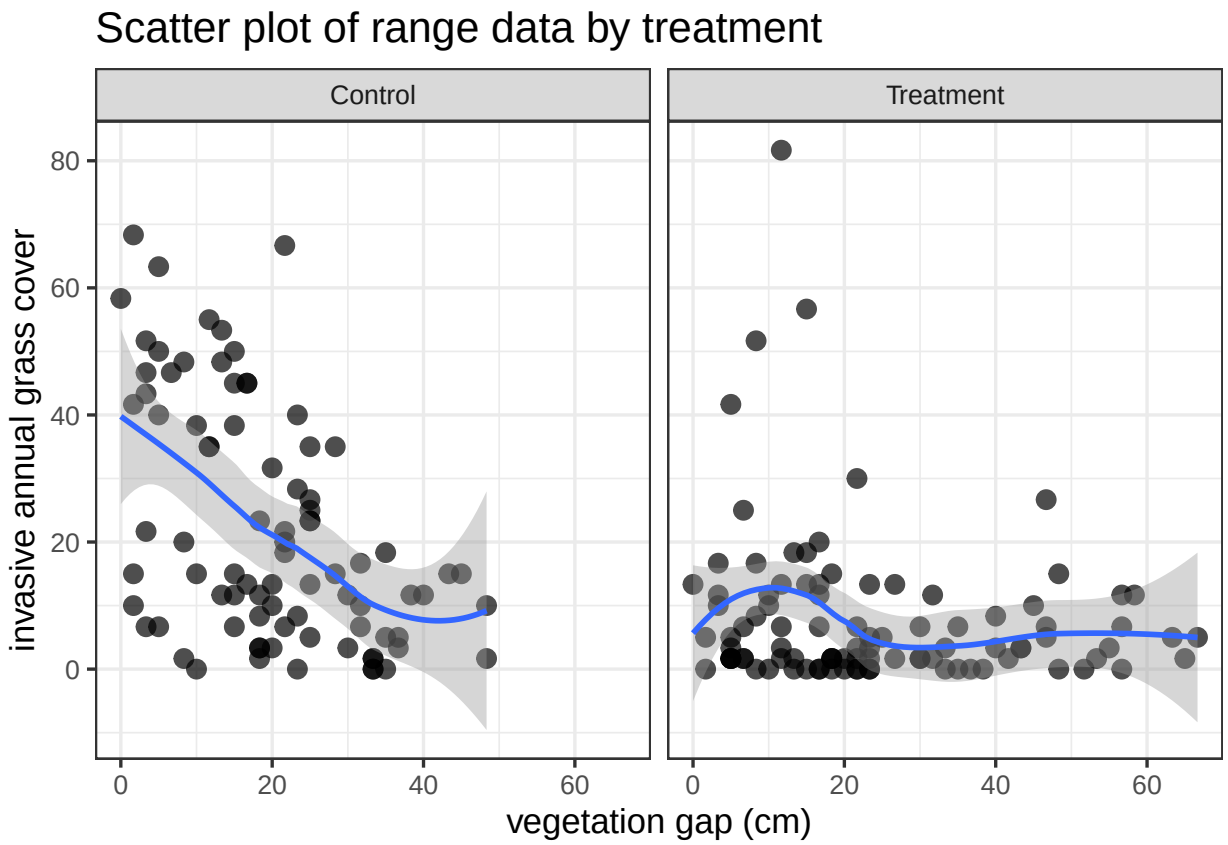

**Faceting**

ggplot allows us to split one plot into multiple plots based on a factor in the data using the facet_wrap() function.

Instead of on the same plot, Let's visualize control and treatment on different plots by adding a facet

```
final_data %>%
  ggplot() +
  geom_point(aes(x = GAP, y = AG_I),
             size = 3, alpha = 0.7)+
  geom_smooth(aes(x = GAP, y = AG_I))+
  theme_bw(13)+
  ggtitle("Scatter plot of range data by treatment") +
  xlab("vegetation gap (cm)") +
  ylab("invasive annual grass cover") +
  facet_wrap(facets = vars(Trt))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




**Exporting plots**

After we make our graphics, we can save them to a file in whatever format you'd like. The ggsave() function allows you to change the dimension and resolution of your plot by adjusting the width, height, and dpi arguments.

```
#first we need to assign our plot to an object that we can export.
my_plot <- final_data %>%
  ggplot() +
  geom_point(aes(x = GAP, y = AG_I),
             size = 3, alpha = 0.7)+
  geom_smooth(aes(x = GAP, y = AG_I))+
  theme_bw(13)+
  ggtitle("Scatter plot of range data by treatment") +
  xlab("vegetation gap (cm)") +
  ylab("invasive annual grass cover") +
  facet_wrap(facets = vars(Trt))


ggsave("my_awesome_plot.png", my_plot, width = 15, height = 10)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

**Now you try!**

Build a ggplot graphic from scratch using the code above for reference. **Challenge**: Create a scatter plot with AG_I on the y axis and PG_N on the x axis that is faceted by the factor Yr. **Bonus challenge**: Change the color of the points according to the size of the GAP.

Let us know if you run into issues - We're happy to help!

```
#final_data %>%
#  ggplot() +
```

## Resources

There's so much more we can do with ggplot2! Here are some sites that can take your plotting in ggplot to the next level!

https://urldefense.com/v3/___https://towardsdatascience.com/data-visualization-with-ggplot2-db04c4956236_
_;!!JYXjzlvb!kDz6d09IT9j5Y4PKUEa6qoRDcLV9SZy1uBFjopaloca5Ie3oZAPWEGA1DCrF5X7AxARsoqKAHcagMShElJA
imPbc$
https://urldefense.com/v3/___https://ggplot2-book.org/index.html___;!!JYXjzlvb!kDz6d09IT9j5Y4PKUEa6qoRDcLV9SZy
https://urldefense.com/v3/___https://datacarpentry.org/R-ecology-lesson/04-visualization-ggplot2.html_
_;!!JYXjzlvb!kDz6d09IT9j5Y4PKUEa6qoRDcLV9SZy1uBFjopaloca5Ie3oZAPWEGA1DCrF5X7AxARsoqKAHcagMShElJA
https://urldefense.com/v3/___https://www.sharpsightlabs.com/blog/geom_smooth/___;!!JYXjzlvb!kDz6d09IT9j5Y4PKUE