R MANUAL, CODES AND NOTES(BASIC)

From swirl, a couple of books and Evi

Paraskevi Massara

Contents

Introduction-General information on R

First question: why am I using the word processor to write this "manual". Good question, I am not so familiar with R markdown to use it quick enough for a personal manual. However, I will try my best to be familiar soon.

General information on R:

- The R package foreign (http://cran.rproject. org/web/packages/foreign/index.html) is part of the standard R distribution and enables you to read data from the statistical packages SPSS, SAS, Stata.
- R is more than just a domain-specific programming language aimed at statisticians
- You can save individual variables with the `save()` function.
- You can save the entire workspace with the `save.image()` function.
- You can save your R script file, using the appropriate save menu command in your code editor.
- Formula with the tilde symbol ~ this means that we want to model the observations using the variables to the right of the tild
- Always put spaces around the less than (<) and greater than (>) operators
- `> getwd()`
- `> load("yourname.rda")`

Matrix Operations

- Multiplying by a scalar

```
X <- matrix(1:12,4,3)
a <- 2
print(a*X)
```

- Matrix multiplication

```
X <- matrix(c(1,3,2,1,-2,1,1,1,-1),3,3)
abc <- c(3,2,1) #use as an example
rbind( sum(X[1,]*abc), sum(X[2,]*abc), sum(X[3,]%*%abc))
```

- The inverse

```
X <- matrix(c(1,3,2,1,-2,1,1,1,-1),3,3)
y <- matrix(c(6,2,1),3,1)
solve(X)%*%y #equivalent to solve(X,y)
```

Sequences

- Create a sequence of numbers:

```
> pi:10
> 15:1 (from 15 14 13 12 11 10…..1)
```

- The command seq does the same but with greater control

```
> seq(0,10,by=0.5)
> my_seq <- seq(5, 10, length=30)
```

- I want to create a vector that includes a sequence from 1 to the number that equals with the length of the vector my_seq. There are 2 ways to do this:

A)
```
> 1:length(my_seq)
```
B)
```
> seq(along.with = my_seq)
```

- If we're interested in creating a vector that contains 40 zeros, we can use rep(0, times = 40). Try it out.

```
> rep(0,times=40)
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
> rep(c(0, 1, 2), each = 10)
> rep(x, 1000)
```

Missing values (NA)

- Normal distribution (this vector contains 1000 draws from a standard normal distribution

```
y <- rnorm(1000)
```

- Take a sample (of 100 elements) from vector x

```
>sample(c(y,z),100)
```

- is.na() function tells us whether each element of a vector is NA (missing value)

```
>sum(my_na)
```

```
> sum(data$V1, na.rm = TRUE)
```
The na.rm=TRUE overlooks the NANs

Dimensions-Giving Names

- length of the vector

```
> length(my_vector)
```

- give dimensions to the vectors (to transform them in rows and columns c(4-r,5-c)

```
dim(my_vector) <- c(4, 5)
```

- to check the number of the rows and the columns of the vector

```
> attributes(my_vector)
```

- create a matrix with the numbers 1-20 that has 4 rows and 5 columns

```
> my_matrix2 <- matrix(1:20,4,5)
```

- confirm that two matrix are actually identical

```
> identical(my_matrix,my_matrix2)
```

- we want to add an extra column with e.g. names

```
A way:
patients<- c("Bill","Gina","Kelly","Sean")
cbind(patients,my_matrix)

B way:
my_data <- data.frame(patients, my_matrix)
```

- I want to give names to the columns of my data set. First I have to create a vector with the names of the columns. After that I use the command colnames

```
> cnames <- c("patient","age", "weight","bp", "rating", "test")

> colnames(my_data) <- cnames
```

- This tells us that there are 194 rows, or observations, and 30 columns, or variables

```
> dim(flags)
[1] 194  30
```

Logic (useful in order to create code)

```
> (FALSE == TRUE) == FALSE
```

- Not equal (!=)

```
> 5!=7
```

- The opposite of 5==7

```
> !5 == 7
```

- There are two AND operators in R, `&` and `&&`. Both operators work similarly, if the right and left operands of AND are both TRUE the entire expression is TRUE, otherwise it is FALSE

- You can use the `&` operator to evaluate AND across a vector. The `&&` version of AND only evaluates the first member of a vector

- TRUE && c(TRUE, FALSE, FALSE) (True) In this case, the left operand is only evaluated with the first member of the right operand (the vector). The rest of the elements in the vector aren't evaluated at all in this expression.

- The OR operator follows a similar set of rules. The `|` version of OR evaluates OR across an entire vector, while the `||` version of OR only evaluates the first member of a vector

- `IsTRUE()`
-
- The function `identical()` will return TRUE if the two R objects passed to it as arguments are identical.

- The xor() function stands for exclusive OR. For xor() to evaluate to TRUE, one argument must be TRUE and one must be FALSE.
- `ints <- sample(10)`

Functions

Functions are one of the fundamental building blocks of the R language. They are small pieces of reusable code that can be treated like any other R object.

- The Sys.Date() sapion returns a string representing today's date
  `> Sys.Date()`
- `submit()` and the script will be evaluated

Assign functions in the following way his function takes the argument `x` as input and manipulates it in some way as it to be an output.

- `function_name <- function(arg1, arg2){`

```
# Manipulate arguments in some way
# Return a value
# }
```

- my_mean <- function(my_vector) {
  mean(my_vector)
  }
  submit()

By providing one argument to the function, and R matched that argument to 'num' since 'num' is the first argument. The default value for 'divisor' is 2, so the function used the default value you provided. So by simply writing remainder(5) the function takes 5 as the first argument and gives directly results.

- remainder <- function(num=5, divisor=2) {
  num %% divisor
  }
  submit()

You can also explicitly specify arguments in a function. When you explicitly designate argument values by name, the ordering of the arguments becomes unimportant e.g. remainder(divisor = 11, num = 5).

```
>evaluate(function(x){x+1}, 7)
[1] 8
> evaluate(function(x){x[1]}, c(8, 4, 0))
```

lapply() and sapply()

These powerful functions, along with their close relatives (vapply() and tapply(), among others) offer a concise and convenient means of implementing the Split-Apply-Combine strategy for data analysis. Each of the *apply functions will SPLIT up some data into smaller pieces, APPLY a function to each piece, then COMBINE the results.
A more detailed discussion of this strategy is found in Hadley Wickham's Journal of Statistical Software paper titled 'The Split-Apply-Combine Strategy for Data Analysis'.

- What we really need is to call the class() function on each individual column. The lapply() (l from the list) function takes a list as input, applies a function to each element of the list, then returns a list of the same length as the original one. Since a data frame is really just a list of vectors (you can see this with as.list(flags)), we can use lapply() to apply the class() function to each column of the flags dataset

- cls_list <- lapply(flags, class) to apply the class() function to each column of the flags dataset and store the result in a variable called cls_list. Actually, it creates a list and as it was previously mentioned split, apply, combine (= into a set of many objects)

- sapply() allows you to automate this process by calling lapply() behind the scenes, but then attempting to simplify (hence the 's' in 'sapply') the result for you

- In general, if the result is a list where every element is of length one, then sapply() returns a vector. If the result is a list where every element is a vector of the same length (> 1), sapply() returns a matrix. If sapply() can't figure things out, then it just returns a list, no different from what lapply() would give you.
- `sum(flags$orange)`

Task: We want the sum of a specific number of columns

- `flag_colors <- flags[, 11:17]` to extract the columns containing the color data and store them in a new data frame called flag_colors. (Note the comma before 11:17. This subsetting command tells R that we want all rows, but only columns 11 through 17.)
  `> lapply(flag_colors,sum)`
- However, this is a list which is not useful. We want to create a vector (apply sum to each part of the flag_colors
  `> sapply(flag_colors,sum)`

- The fact that the elements of the unique_vals list are all vectors of *different* length poses a problem for sapply(), since there's no obvious way of simplifying the result.

- Apply unique to every column of the flags

sapply(flags, unique) returns a list containing one vector of unique values for each column of the flags dataset

```
> sapply(flags,unique)
```

Vapply Tapply

Whereas sapply() tries to 'guess' the correct format of the result, vapply() allows you to specify it explicitly. If the result doesn't match the format you specify, vapply() will throw an error, causing the operation to stop. This can prevent significant problems in your code that might be caused by getting unexpected return values from sapply ().

- 'character(1)' argument tells R that we expect the class function to return a character vector of length 1 when applied to EACH column of the flags dataset
- vapply() may perform faster than sapply() for large datasets
- Table function: for variables that have categories (e.g. 1, 2, 3, 4, 5, 6..) and we want to see how many variables fall within each category

```
>table(flags$landmass)
> table(flags$animate)
```

- tapply(flags$animate, flags$landmass, mean) to apply the mean function to the 'animate' variable separately for each of the six landmass groups thus giving us the proportion of flags containing an animate image WITHIN each landmass group

```
> tapply(flags$animate, flags$landmass, mean)
        1         2         3         4         5         6
0.4193548 0.1764706 0.1142857 0.1346154 0.1538462 0.3000000
```

- Similarly, we can look at a summary of population values (in round millions) for countries with and without the color red on their flag with tapply (flags$population, flags$red, summary).

Looking at Data

- ls() to list the variables in your workspace

```
> ls()
```

- Begin by checking the class of the plants variable with class(dataset=plants)

- `> class(plants)`
- `[1] "data.frame"`

- dim(plants) to see exactly how many rows and columns we're dealing with

```
> dim(plants)
```

- You can also use nrow(plants) to see only the number of rows and ncol(plants) for the number of columns

- how much space the dataset is occupying in memory

```
> object.size(plants)
```

- names(plants) will return a character vector of column (i.e. variable) names

```
> names(plants)
```

- The `head()` function allows you to preview the top of the dataset

- head(plants, 10) will show you the first 10 rows of the dataset

- tail() to preview the end of the dataset
- summary() provides different output for each variable, depending on its class. For numeric data such as Precip_Min, summary() displays the minimum, 1st quartile, median, mean, 3rd quartile, and maximum. These values help us understand how the data are distributed
- For categorical variables (called 'factor' variables in R), summary() displays the number of times each value (or 'level') occurs in the data
- understanding the *str*ucture of your data is str()

Getting and Cleaning Data

Manipulating Data

Manipulate data using dplyr. dplyr is a fast and powerful R package written by Hadley Wickham and Romain Francois that provides a consistent and concise grammar for manipulating tabular data.

One unique aspect of dplyr is that the same set of tools allow you to work with tabular data from a variety of sources, including data frames, data tables, databases and multidimensional arrays.

- It's important that you have dplyr version 0.4.0 or later. To confirm this, type packageVersion("dplyr").
- The first step of working with data in dplyr is to load the data into what the package authors call a 'data frame tbl' or 'tbl_df'. Use the following code to create a new tbl_df called cran: `cran <- tbl_df(mydf).`
- To avoid confusion and keep things running smoothly, remove the original data frame from your workspace with `rm("mydf").`
- First, we are shown the class and dimensions of the dataset. Just below that, we get a preview of the data. Instead of attempting to print the entire dataset, dplyr just shows us the first 10 rows of data and only as many columns as fit neatly in our console. At the bottom, we see the names and classes for any variables that didn't fit on our screen.

- According to the "Introduction to dplyr" vignette written by the package authors, "The dplyr philosophy is to have small functions that each do one thing well." Specifically, dplyr supplies five 'verbs' that cover most fundamental data manipulation tasks: `select(), filter(), arrange(), mutate(), and summarize().`

- Instead of specifying the columns we want to keep, we can also specify the columns we want to throw away. To see how this works, do select(dataset, -time) to omit the time column.

```
> -(5:20)

> select(cran, -(X:size))
>select(cran,size:ip_id)
```

: every thing between X and size including

- >= sepal.Lenght 7.0 (everything that has 7 length)
- -"V1" everything except this variable
-
- "How do I select a subset of rows?" That's where the filter() function comes in.

- Use filter(cran, package == "swirl") to select all rows for which the package variable is equal to "swirl". Be sure to use two equals signs side-by-side!

```
> filter(cran, package == "swirl")
 > filter(cran, r_version == "3.1.1", country == "US")
```

- filter(cran, country == "US" | country == "IN") will gives us all rows for which the country variable equals either "US" or "IN"
- `is.na(c(3, 5, NA, 10)) for missing variables`
- filter() to return all rows of cran for which r_version is NOT NA.

```
> filter(cran, !is.na(r_version))
```

- Sometimes we want to order the rows of a dataset according to the values of a particular variable. This is the job of arrange().

```
> arrange(cran2, ip_id)
```

- To do the same, but in descending order, change the second argument to desc(ip_id), where desc() stands for 'descending'

```
> arrange(cran2, desc(ip_id))
```

- We can also arrange the data according to the values of multiple variables. For example, arrange(cran2, package, ip_id) will first arrange by package names (ascending alphabetically), then by ip_id. This means that if there are multiple rows with the same value for package, they will be sorted by ip_id (ascending numerically).
- create a new variable based on the value of one or more variables already in a dataset. The mutate() function does exactly this

```
> mutate(cran3, size_mb = size / 2^20)
```

- One very nice feature of mutate() is that you can use the value computed for your second column (size_mb) to create a third column, all in the same line of code. To see this in action, repeat the exact same command as above, except add a third argument creating a column that is named size_gb and equal to size_mb / 2^10.

```
> mutate(cran3, size_mb = size / 2^20, size_gb=size_mb / 2^10)
```

The last of the five core dplyr verbs, summarize(), collapses the dataset to a single row. Let's say we're interested in knowing the average download size. summarize(cran, avg_bytes = mean(size)) will yield the mean value of the size variable. Here we've chosen to label the result 'avg_bytes', but we could have named it anything.

- `> avg_bytes <-summarize(cran, avg_bytes = mean(size))`

Grouping and Chaining with dplyr

The main idea behind grouping data is that you want to break up your dataset into groups of rows based on the values of one or more variables. The group_by() function is responsible for doing this.

- Group cran (your dataset) by the package variable and store the result in a new object called by_package.
- Instead of returning a single value, summarize() now returns the mean size for EACH package in our dataset

  `> summarise(mydata,mean(size))`

- The 'count' column, created with n(), contains the total number of rows (i.e. downloads) for each package. The 'unique' column, created with n_distinct(ip_id), gives the total number of unique downloads for each package, as measured by the number of distinct ip_id's. The 'countries' column, created with n_distinct(country), provides the number of countries in which each package was downloaded. And finally, the 'avg_bytes' column, created with mean(size), contains the mean download size (in bytes) for each package.

```
pack_sum <- summarize(by_package,
                      count = n(),
                      unique = n_distinct(ip_id),
                      countries =n_distinct(country) ,
                      avg_bytes =mean(size))
submit()
```

- We need to know the value of 'count' that splits the data into the top 1% and bottom 99% of packages based on total downloads. In statistics, this is called the 0.99, or 99%, sample quantile. Use quantile(pack_sum$count, probs = 0.99) to determine this number
- View all rows with View(mydata). Note that the 'V' in View() is capitalized.
- arrange() the rows of mydata based on the 'count' column and assign the result to a new object called top_counts_sorted. We want the packages with the highest number of downloads at the top, which means we want 'count' to be in descending order.
- In order to arrange() our dataset by the i.e. 'cheese' column, in descending order

  `> vector <-arrange(mydata,desc(cheese))`

Tidying Data with tidyr

```
> library(tidyr)
```

Tidy data is formatted in a standard way that facilitates exploration and analysis and works seamlessly with other tidy data tools. Specifically, tidy data satisfy three conditions:

1) Each variable forms a column

2) Each observation forms a row

3) Each type of observational unit forms a table

Any dataset that doesn't satisfy these conditions is considered 'messy' data. Examples:

1. Variables are stored in both rows and columns
2. A single observational unit is stored in multiple tables
3. Multiple variables are stored in one column
4. Column headers are values, not variable names
5. Multiple types of observational units are stored in the same table

Tidying messy datasets

- When you have column headers that are values, not variable names

I've created a simple dataset called 'students' that demonstrates this scenario

```
grade male female
1     A    1      5
2     B    5      0
3     C    5      2
4     D    5      5
5     E    7      4
```

The first column represents each of five possible grades that students could receive for a particular class. The second and third columns give the number of male and female students, respectively, that received each grade. This dataset actually has three variables: grade, sex, and count. The first variable, grade, is already a column, so that should remain as it is. The second variable, sex is captured by the second and third column headings. The third variable, count, is the number of students for each combination of grade and sex.

To tidy the students data, we need to have one column for each of these three variables. We'll use the gather() function from tidyr to accomplish this

gather() function takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed. You use gather() when you notice that you have columns that are not variables.

```
gather(data, key, value, ..., na.rm = FALSE, convert = FALSE,
  factor_key = FALSE)
```

The data argument gives the name of the original dataset. The key and value arguments give the column names for our tidy dataset.

```
> gather(students,sex,count,-grade)
```

The final argument, -grade, says that we want to gather all columns EXCEPT the grade column (since grade is already a proper column variable.)

```
grade    sex count
1      A   male     1
2      B   male     5
3      C   male     5
4      D   male     5
5      E   male     7
6      A female     5
7      B female     0
8      C female     2
9      D female     5
10     E female     4
```

Each row of the data now represents exactly one observation, characterized by av unique combination of the grade and sex variables. Each of our variables (grade, sex, and count) occupies exactly one column. That's tidy data!

- When multiple variables are stored in one column

```
grade male_1 female_1 male_2 female_2
1     A      3        4      3        4
2     B      6        4      3        5
3     C      7        4      3        8
4     D      4        0      8        1
5     E      1        1      2        7
```

This dataset is similar to the first, except now there are two separate classes, 1 and 2, and we have total counts for each sex within each class. students2 suffers from the same messy data problem of having column headers that are values (male_1, female_1, etc.) and not variable names (sex, class, and count). However, it also has multiple variables stored in each column (sex and class), which is another common symptom of messy data. Tidying this dataset will be a twostep process.

```
> res<-gather(students2,sex_class,count,-grade)
> separate(res,sex_class,into=c("sex", "class"))
```

```
   grade    sex class count
1      A   male     1     3
2      B   male     1     6
3      C   male     1     7
4      D   male     1     4
5      E   male     1     1
6      A female     1     4
7      B female     1     4
8      C female     1     4
9      D female     1     0
10     E female     1     1
11     A   male     2     3
12     B   male     2     3
13     C   male     2     3
14     D   male     2     8
15     E   male     2     2
16     A female     2     4
17     B female     2     5
18     C female     2     8
19     D female     2     1
20     E female     2     7
```

Tidying students2 required both gather() and separate(), causing us to save an intermediate result (res). However, just like with dplyr, you can use the %>% operator to chain multiple function calls together.

2nd way to do this:

```
students2 %>%
 gather(sex_class, count, -grade) %>%
 separate(sex_class, c("sex", "class")) %>%
 print
```

- A third symptom of messy data is when variables are stored in both rows and columns.

```
    name    test class1 class2 class3 class4 class5
1  Sally midterm      A   <NA>      B   <NA>   <NA>
2  Sally   final      C   <NA>      C   <NA>   <NA>
3   Jeff midterm   <NA>      D   <NA>      A   <NA>
4   Jeff   final   <NA>      E   <NA>      C   <NA>
5  Roger midterm   <NA>      C   <NA>   <NA>      B
6  Roger   final   <NA>      A   <NA>   <NA>      A
7  Karen midterm   <NA>   <NA>      C      A   <NA>
8  Karen   final   <NA>   <NA>      C      A   <NA>
9  Brian midterm      B   <NA>   <NA>   <NA>      A
10 Brian   final      B   <NA>   <NA>   <NA>      C
```

Solution:

```
students3 %>%
 gather(class, grade, class1:class5, na.rm = TRUE) %>%
 print
```

tidyr makes it easy to reference multiple adjacent columns with class1:class5, just like with sequences of numbers. Since each student is only enrolled in two of the five possible classes, there are lots of missing values (i.e. NAs). The argument na.rm = TRUE omit these values from the final result.

```
students3 %>%
 gather(class, grade, class1:class5, na.rm = TRUE) %>%
```

```
 spread(test, grade) %>%
 print
```

If we want the values in the class column to simply be 1, 2, ..., 5 and not class1, class2, ..., class5 we can use the parse_number() function from readr to accomplish this. parse_number ("class5") readr is required for certain data manipulations, such as `parse_number()

```
students3 %>%
 gather(class, grade, class1:class5, na.rm = TRUE) %>%
 spread(test, grade) %>%
 mutate(class = parse_number(class)) %>%
 print
```

- When multiple observational units are stored in the same table.

```
student_info <- students4 %>%
 select(id, name, sex) %>%
 unique %>%
 print
```

It's important to note that we leave the id column in all tables (new and old). In the world of relational databases, 'id' is called our 'primary key' since it allows us to connect data. Without a unique identifier, we might not know how the tables are related.

- When a single observational unit is stored in multiple tables

Use dplyr's mutate() to add a new column to the passed table. The column should be called status and the value, "passed" (a character string), should be the same for all students. 'Overwrite' the current version of passed with the new one.

```
> passed <-passed %>% mutate(status="passed")
```

Call bind_rows() with two arguments, passed and failed (in that order), to join the two tables.

```
> bind_rows(passed,failed)
```

```
sat %>%
  select(-contains("total")) %>%
  gather(part_sex, count, -score_range) %>%
  separate(part_sex, c("part", "sex")) %>%
  print
```

```
sat %>%
  select(-contains("total")) %>%
  gather(part_sex, count, -score_range) %>%
  separate(part_sex, c("part", "sex")) %>%
  group_by(part,sex) %>%
  mutate(total = sum(count),
         prop = count / total) %>% print
```

Dates and Times with lubridate

lubridate has a consistent, memorable syntax, that makes working with dates fun instead of frustrating, a major limitation of R.

```
> library(lubridate)
```

- The today() function returns today's date
- We can also get the day of the week from our dataset using the wday() function. It will be represented as a number, such that 1 = Sunday, 2 = Monday, 3 = Tuesday, etc. add a second argument, label = TRUE, to display the *name* of the weekday
- To parse the date

```
> ymd("1989 May 17")

[1] "1989-05-17"
```

- The update() function allows us to update one or more components of a date-time. For example, let's say the current time is 08:34:55 (hh:mm:ss). Update this_moment to the new time using the following command: update(this_moment, hours = 8, minutes = 34, seconds = 55).

```
this_moment <- update(this_moment, hours = 10, minutes = 16, seconds = 0)
```

- Now, pretend you are in New York City and you are planning to visit a friend in Hong Kong. You seem to have misplaced your itinerary, but you know that your flight departs New York at 17:34 (5:34pm) the day after tomorrow. You also know that your flight is scheduled to arrive in Hong Kong exactly 15 hours and 50 minutes after departure. Let's reconstruct your itinerary from what you can remember, starting with the full date and time of your departure. We will approach this by finding the current date in New York, adding 2 full days, then setting the time to 17:34.

```
> nyc<- now("America/New_York")
> depart <- nyc + days(2)
> depart <- update(depart, hours=17, minutes=34)
> arrive <-  depart + hours(15)+minute(50)
> arrive <-with_tz(arrive, "Asia/Hong_Kong")
```

Exploratory Data Analysis

Principles of Analytic Graphs and Exploratory Graphs

We use graphs in order to show

1. causality, mechanism, explanation
2. comparisons
3. multivariate data

Exploratory graphs serve mostly the same functions as graphs. They help us find patterns in data and understand its properties. They suggest modeling strategies and help to debug analyses. We DON'T use exploratory graphs to communicate results.

- Command to plot a blue boxplot

```
> boxplot(dataset, col="blue")
```

- Drew a horizontal line at 12

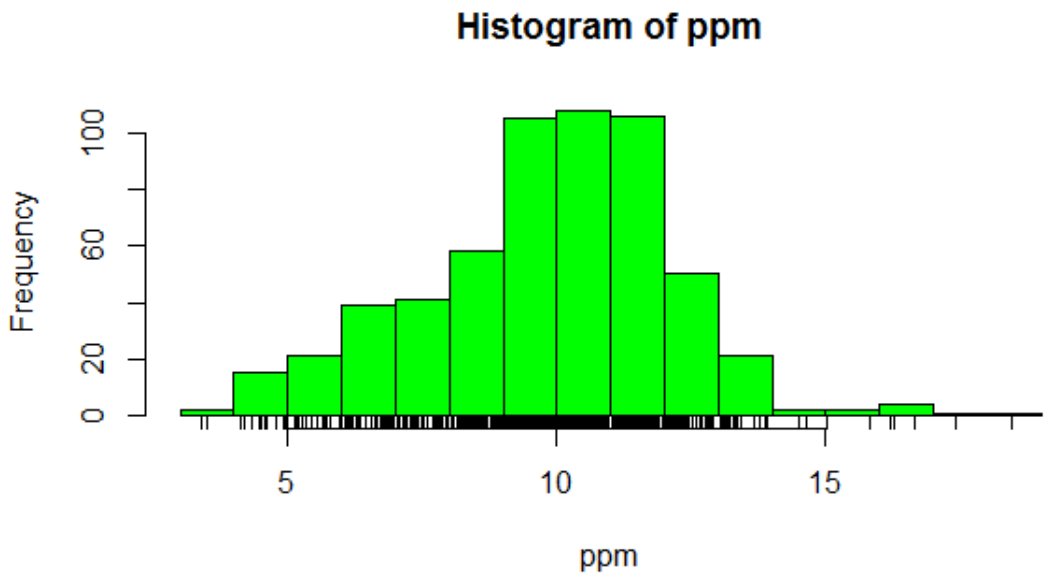
```
> abline(h=12)
```

- Command to plot a green histogram

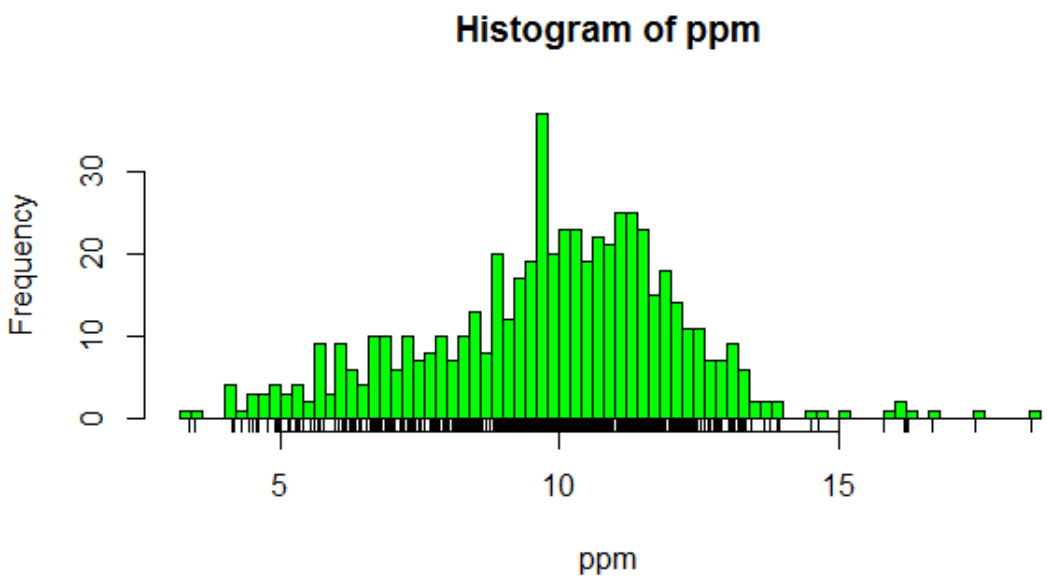
```
> hist(ppm, col="green")
```

- The one-dimensional plot, with its grayscale representation, gives you a little more detailed information about how many data points are in each bucket and where they lie within the bucket

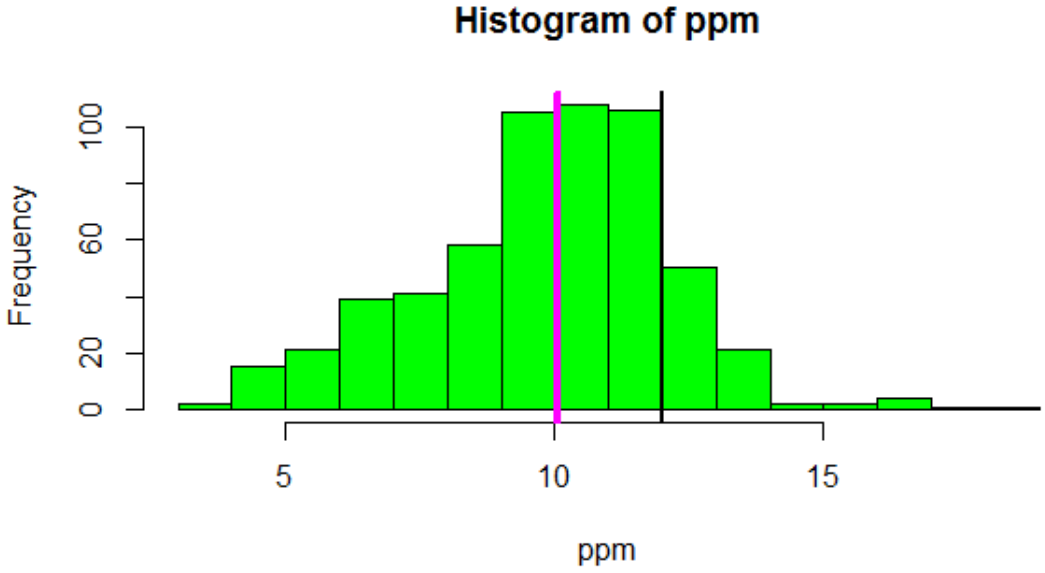
```
> rug(dataset)
```



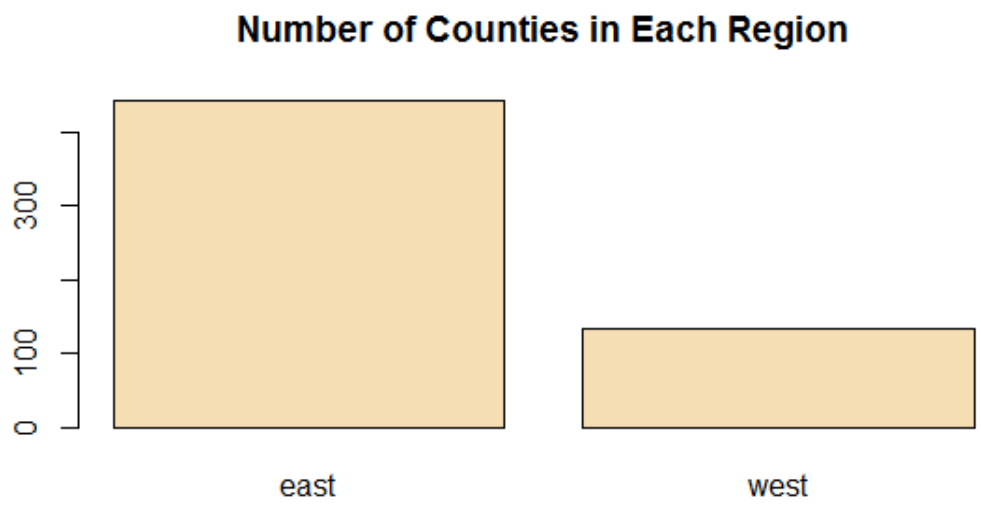

```
> hist(ppm, col = "green", breaks = 100)
```



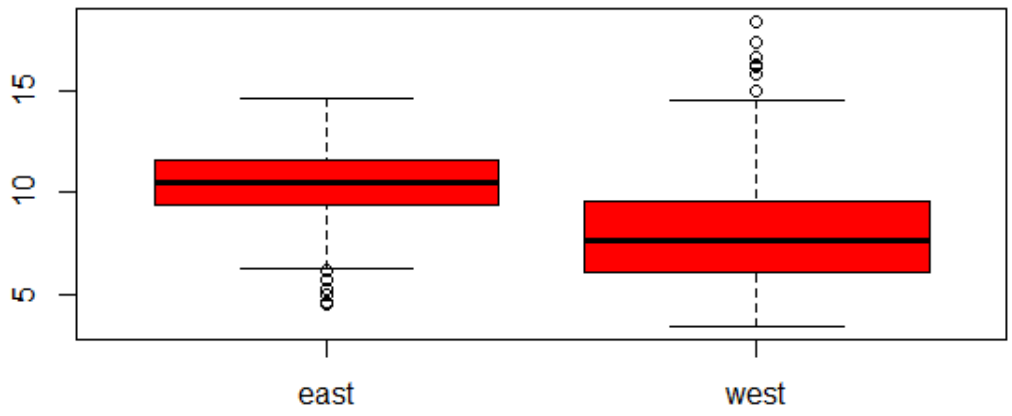

```
> abline(v=median(ppm), col="magenta", lwd=4)
```




```
> barplot(reg,col="wheat", main = "Title of the Plot")
```



- We use the R formula y ~ x to show that y (in this case pm25) depends on x (region). Since both come from the same data frame (pollution) we can specify a data argument set equal to pollution. By doing this, we don't have to type pollution$pm25 (or ppm) and pollution$region. We can just specify the formula pm25~region. Call boxplot now with this formula as its argument, data equal to pollution, and col equal to "red".

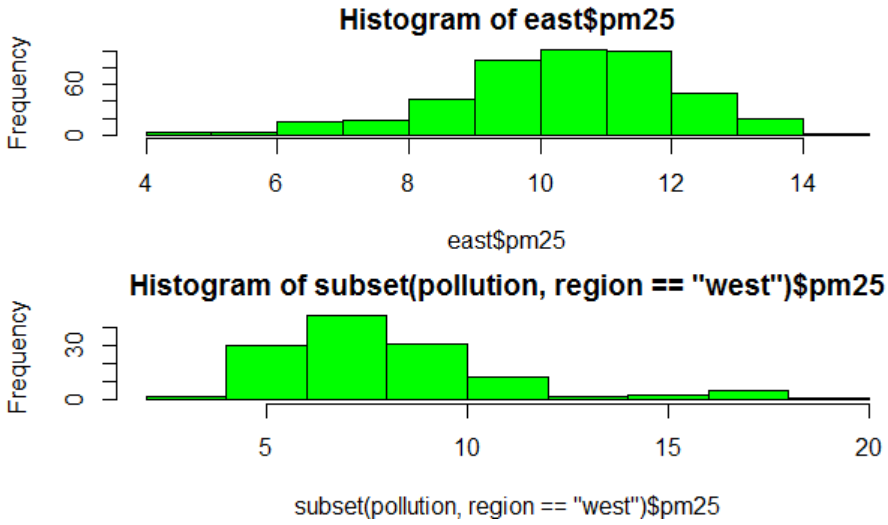



```
> boxplot(pm25~region, data = pollution, col="red")
```

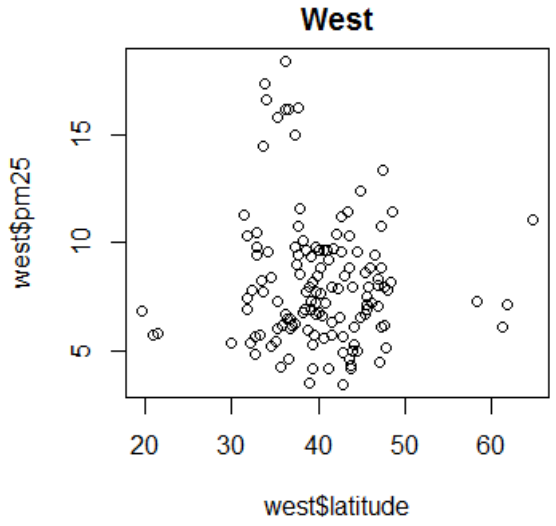
Similarly we can plot multiple histograms in one plot, though to do this we have to use more

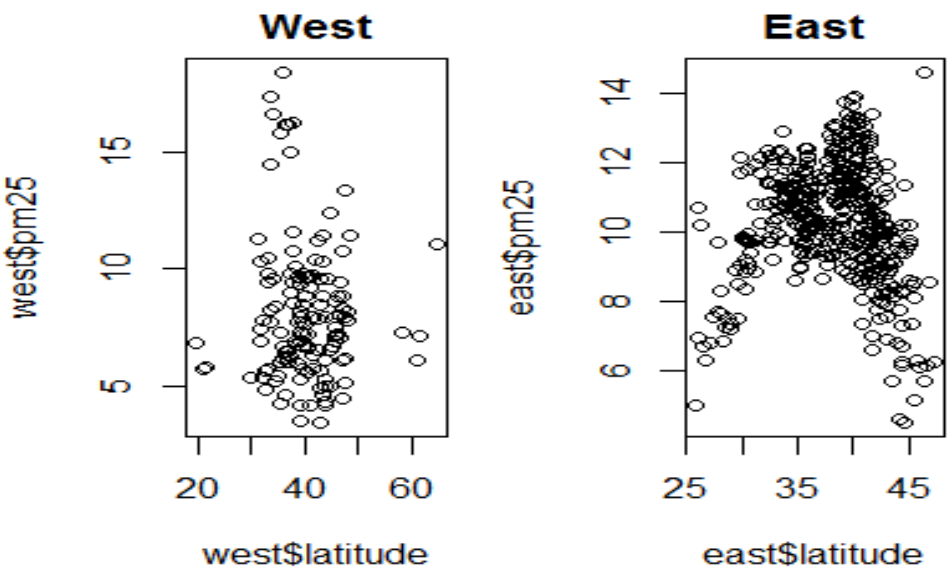
than one R command. First we have to set up the plot window with the R command par which specifies how we want to lay out the plots, say one above the other. We also use par to specify margins, a 4-long vector which indicates the number of lines for the bottom, left, top and right.

```
> par(mfrow=c(2,1),mar=c(4,4,2,1)) par told R we were going to have one column with 2 rows

> east <- subset(pollution, region=="east")
> hist(subset(pollution,region=="west")$pm25, col = "green")
```




```
> par(mfrow = c(1, 2), mar = c(5, 4, 2, 1))
> west <- subset(pollution, region=="west" )
> plot(west$latitude, west$pm25, main = "West" )
> plot(east$latitude, east$pm25, main = "East" )
```



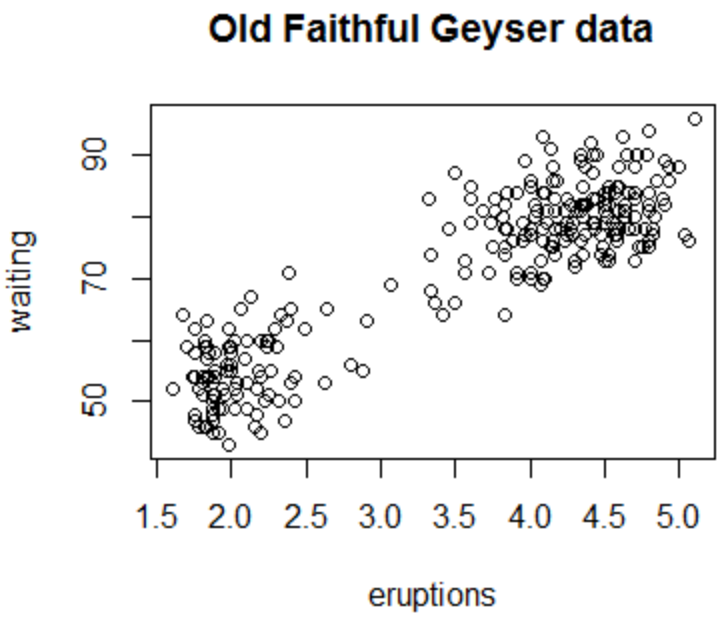


Graphics Devices in R

There are several different kinds of file devices with particular characteristics and hence uses. These include PDF, PNG, JPEG, SVG, and TIFF, To be clear, when you make a plot in R, it has to be "sent" to a specific graphics device. Usually this is the screen (the default device), especially when you're doing exploratory work. You'll send your plots to files when you're ready to publish a report, make a presentation, or send info to colleagues.

`with` is a generic function that evaluates `expr` in a local environment constructed from `data`

```
> with(faithful, plot(eruptions, waiting))
> title(main="Old Faithful Geyser data")
```



The second way to create a plot is to send it to a file device. Depending on the type of plot you're making, you explicitly launch a graphics device, e.g., a pdf file. Type the command pdf(file="myplot.pdf") to launch the file device. This will create the pdf file myplot.pdf in your working directory.

When plotting to a file device, you have to close the device with the command dev.off(). This is very important! After closing, you'll be able to view the pdf file on your computer.

There are two basic types of file devices, vector and bitmap devices. These use different formats and have different characteristics. Vector formats are good for line drawings and plots with solid colors using a modest number of points, while bitmap formats are good for plots with a large number of points, natural scenes or web-based plots.

4 specific vector formats. The first is pdf, which we've just used in our example. This is useful for line-type graphics and papers. It resizes well, is usually portable, but it is not efficient if a plot has many objects/points. The second is svg which is XML-based, scalable vector graphics. This supports animation and interactivity and is potentially useful for web-based plots. The last two vector formats are win.metafile, a Windows-only metafile format, and postscript (ps), an older format which also resizes well, is usually portable, and can be used to create encapsulated postscript files. Unfortunately, Windows systems often don’t have a postscript viewer.

4 different bitmap formats. The first is png (Portable Network Graphics) which is good for line drawings or images with solid colors. It uses lossless compression (like the old GIF format), and most web browsers can read this format natively. In addition, png is good for plots with many points, but it does not resize well. In contrast, jpeg files are good for photographs or natural

scenes. They use lossy compression, so they're good for plots with many points. Files in jpeg format don't resize well, but they can be read by almost any computer and any web browser. They're not great for line drawings.

Every open graphics device is assigned to an integer greater than or equal to 2. You can change the active graphics device with dev.set(<integer>) where <integer> is the number associated with the graphics device you want to switch to. For pdf the number is 4.

Plotting Systems

```
> text(mean(cars$speed),max(cars$dist), "SWIRL rules!" )
```

Lattice System comes in the package of the same name. Unlike the Base System, lattice plots are created with a single function call such as xyplot or bwplot. Margins and spacing are set automatically because the entire plot is specified at once. The lattice system is most useful for conditioning types of plots which display how y changes with x across levels of z. The variable z might be a categorical variable of your data. This system is also good for putting many plots on a screen at once.

- see how many categories there are and how many states are in a column

```
> table(state$region)
```

- see how big the dataset is

```
> dim(mpg)
```

For the last plotting system, ggplot2, which is a hybrid of the base and lattice systems. It automatically deals with spacing, text, titles (as Lattice does) but also allows you to annotate by "adding" to a plot (as Base does), so it's the best of both worlds.

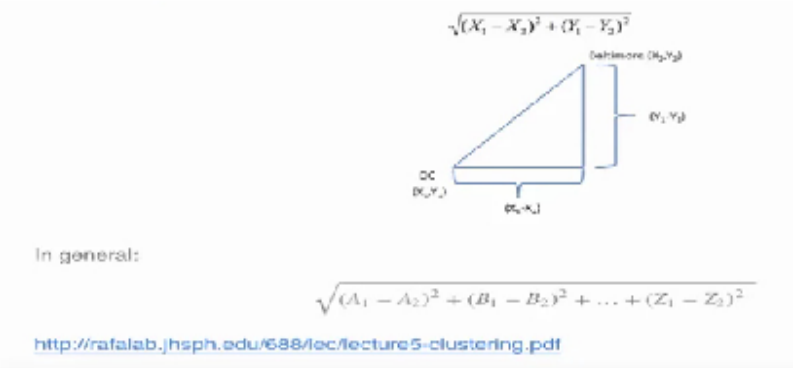
Hierarchical Clustering

Hierarchical clustering, a simple way of quickly examining and displaying multi-dimensional data. This technique is usually most useful in the early stages of analysis when you're trying to get an understanding of the data e.g., finding some pattern or relationship between different factors or variables. As the name suggests hierarchical clustering creates a hierarchy of clusters. In this method, each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. This means that we'll find the closest two points and put them together in one cluster, then find the next closest pair in the updated picture, and so forth.

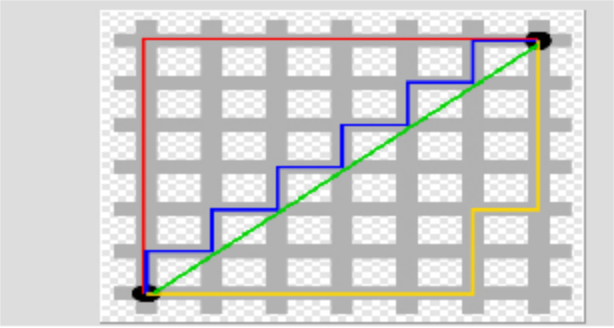
One can decide to stop clustering either when the clusters are too far apart to be merged (distance criterion) or when there is a sufficiently small number of clusters (number criterion).

There are several ways to measure distance or similarity. Euclidean distance and correlation similarity are continuous measures, while Manhattan distance is a binary measure.

Euclidean distance is what you learned about in high school algebra. Given two points on a plane, (x1,y1) and (x2,y2), the Euclidean distance is the square root of the sums of the squares of the distances between the two x-coordinates (x1-x2) and the two y-coordinates (y1-y2). This as an application of the Pythagorean theorem which yields the length of the hypotenuse of a right triangle.



Manhattan distance is the sum of the absolute values of the distances between each coordinate, so the distance between the points (x1,y1) and (x2,y2) is |x1-x2|+|y1-y2|. As with Euclidean distance, this too generalizes to more than 2 dimensions. You want to travel from the point at the lower left to the one on the top right. The shortest distance is the Euclidean (the green line), but you're limited to the grid, so you have to follow a path similar to those shown in red, blue, or yellow. These all have the same length (12) which is thenumber of small gray segments covered by their paths.



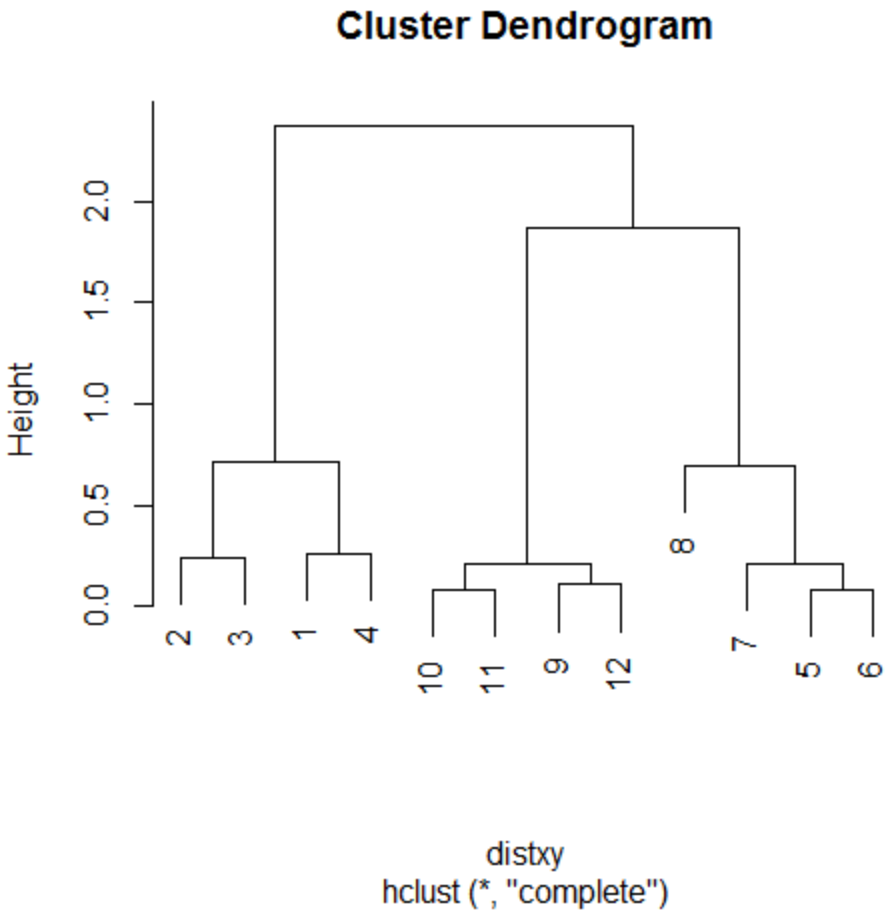
The R command dist with the argument dataFrame computes the (smallest) distances between all pairs of these points.

```
> dist(dataFrame)
```

R provides a simple function which you can call which creates a dendrogram for you. It's called hclust() and takes as an argument the pairwise distance matrix.

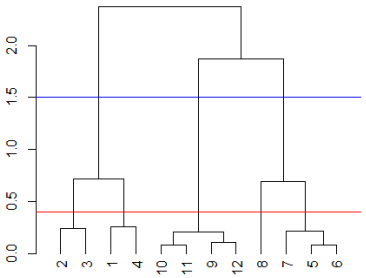
```
matrix in a variable called i.e. distxy
```

```
> hc <- hclust(distxy)
> plot(hc)
```



The number of clusters determined by counting the number of lines that are in the cut at the distance with the abline command (red line, 5 clusters).

```
> abline(h=0.4, col="red")
```

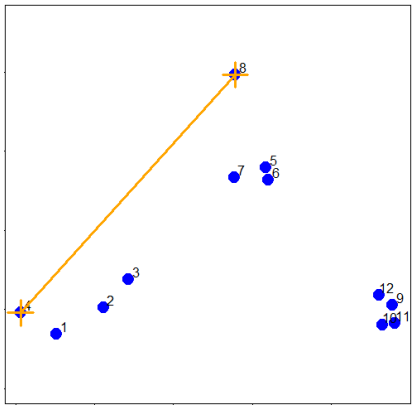


The shortest distance was around .08, so a distance smaller than that would make all the points their own private clusters. So the number of clusters in your data depends on where you draw the line.

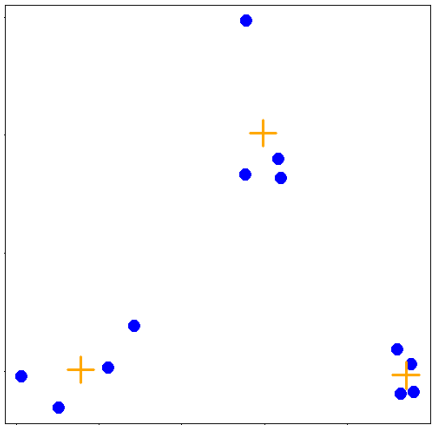
- How distances between clusters of points are measured ?

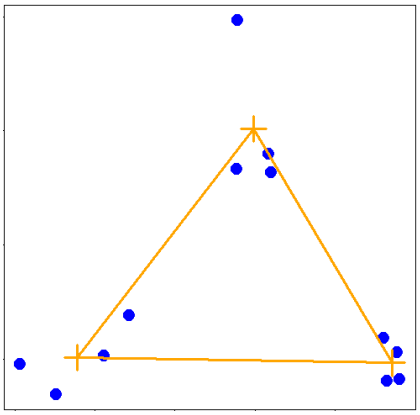
Complete linkage and it says that if you're trying to measure a distance between two clusters, take the greatest distance between the pairs of points in those two clusters. Obviously such pairs contain one point from each cluster.

So if we were measuring the distance between the two clusters of points (1 through 4) and (5 through 8), using complete linkage as the metric we would use the distance between points 4 and 8 as the measure since this is the largest distance between the pairs of those groups.



The second way to measure a distance between two clusters that we'll just mention is called average linkage. First you compute an "average" point in each cluster (think of it as the cluster's center of gravity). We do this by computing the mean (average) x and y coordinates of the points in the cluster. Then you compute the distances between each cluster average to compute the interclasses distance





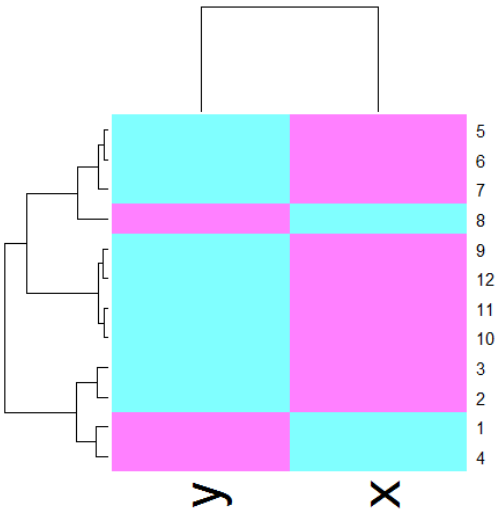
Heat Maps

A heat map is "a graphical representation of data where the individual values contained in a matrix are represented as colors. Heat maps originated in 2D displays of the values in a data matrix. Larger values were represented by small dark gray or black squares (pixels) and smaller values by lighter squares.

- Nice concise tutorial on creating heatmaps in R exists at http://sebastianraschka.com/Articles/heatmaps_in_r.html#clustering

```
> heatmap(dataMatrix, col=cm.colors(25))
```

The first argument is dataMatrix and the second is col set equal to cm.colors(25). This last is optional, but we like the colors better than the default ones.



Description:

This is a very simple heat map - simple because the data isn't very complex. The rows and columns are grouped together as shown by colors. The top rows (labeled 5, 6, and 7) seem to be in the same group (same colors) while 8 is next to them but colored differently. This matches the dendrogram shown on the left edge. Similarly, 9, 12, 11, and 10 are grouped together (row-wise) along with 3 and 2. These are followed by 1 and 4 which are in a separate group. Column data is treated independently of rows but is also grouped.

K_Means_Clustering

```
Package 'ggplot2', 'fields', 'jpeg', 'datasets'
```

As with hierarchical clustering, this technique is most useful in the early stages of analysis when we're trying to get an understanding of the data, e.g., finding some pattern or relationship between different factors or variables.

The k-means method "aims to partition the points into k groups such that the sum of squares from points to the assigned cluster centers is minimized. k-means is a portioning approach which requires that you first guess how many clusters you have (or want). Once you fix this number, you randomly create a "centroid" (a phantom point) for each cluster and assign each point or observation in your dataset to the centroid to which it is closest. Once each point is assigned a centroid, you readjust the centroid's position by making it the average of the points assigned to it. Once you have repositioned the centroids, you must recalculate the distance of the observations to the centroids and reassign any, if necessary, to the centroid closest to them. Again, once the reassignments are done, readjust the positions of the centroids based on the new cluster membership. The process stops once you reach an iteration in which no adjustments are made or when you've reached some predetermined maximum number of iterations. So k-means clustering requires some distance metric (say Euclidean), a hypothesized fixed number of clusters, and an initial guess as to cluster centroids.

Step 1: Two 3-long vectors, cx and cy. These respectively hold the x- and y- coordinates for 3 proposed centroids. For convenience, we've also stored them in a 2 by 3 matrix cmat. The x coordinates are in the first row and the y coordinates in the second.

```
     [,1] [,2] [,3]
[1,]    1  1.8  2.5
[2,]    2  1.0  1.5
```

The coordinates of these points are (1,2), (1.8,1) and (2.5,1.5). We'll add these centroids to the plot of our points. Do this by calling the R command points with 6 arguments. The first 2 are cx and cy, and the third is col set equal to the concatenation of 3 colors, "red", "orange", and "purple". The fourth argument is pch set equal to 3 (a plus sign), the fifth s cex set equal to 2 (expansion of character), and the final is lwd (line width) also set equal to 2.

```
> points(cx,cy,col=c("red","orange","purple"),pch=3,cex=2,lwd=2)
```

The function called mdist takes 4 arguments. The vectors of data points (x and y) are the first two and the two vectors of centroid coordinates (cx and cy) are the last two.

```
> mdist(x,y,cx,cy)
```

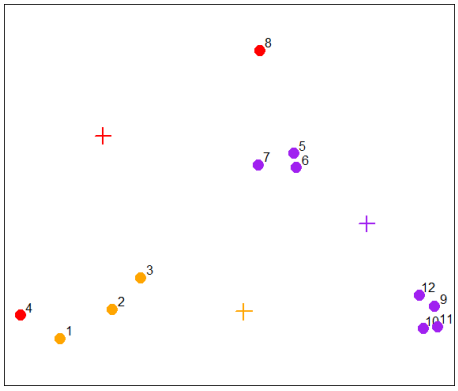
```
        [,1]      [,2]      [,3]     [,4]      [,5]      [,6]      [,7]
[,8]
[1,] 1.392885 0.9774614 0.7000680 1.264693 1.1894610 1.2458771
0.8113513 1.026750
[2,] 1.108644 0.5544675 0.3768445 1.611202 0.8877373 0.7594611
0.7003994 2.208006
[3,] 3.461873 2.3238956 1.7413021 4.150054 0.3297843 0.2600045
0.4887610 1.337896
          [,9]     [,10]     [,11]     [,12]
[1,] 4.5082665 4.5255617 4.8113368 4.0657750
[2,] 1.1825265 1.0540994 1.2278193 1.0090944
[3,] 0.3737554 0.4614472 0.5095428 0.2567247
```

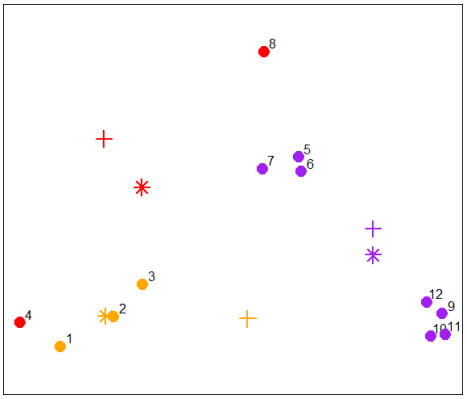
Which cluster would point 6 be assigned to? (Which row in column 6 has the lowest value)

```
> apply(distTmp, 2, which.min)
> points(x,y,pch=19,cex=2,col=cols1[newClust])
```

use tapply to find the x coordinate of the new centroid. Recall 3 arguments, x, newClust2, and mean.

```
> tapply(y, newClust, mean)

> tapply(x, newClust, mean)

> points(newCx,newCy, col=cols1,pch=8,cex=2,lwd=2)
 > mdist(x,y,newCx,newCy)
 > points(x,y,pch=19,cex=2,col=cols1[newClust2]) and the points are
reassigned to new clusters
```






```
> tapply(x, newClust2, mean)
> tapply(y, newClust2, mean)
```

Store off these coordinates for you in the variables finalCx and finalCy. Plot these new centroids using the points function with 6 arguments. The first 2 are finalCx and finalCy. The argument col should equal cols1, pch should equal 9, cex 2 and lwd 2.

```
> points(finalCx,finalCy,col=cols1,pch=9,cex=2,lwd=2)
```

Now that you've gone through an example step by step, you'll be relieved to hear that R provides a command to do all this work for you. Unsurprisingly it's called kmeans and, although it has several parameters, we'll just mention four. These are x, (the numeric matrix of data), centers, iter.max, and nstart. The second of these (centers) can be either a number of clusters or a set of initial centroids. The third, iter.max, specifies the maximum number of iterations to go through, and nstart is the number of random starts you want to try if you specify centers as a number.

```
> kmObj$iter
```

```
> plot(x,y,col=kmObj$cluster,pch=19,cex=2)
```

Now add the centroids which are stored in kmObj$centers. Use the points function with 5 arguments. The first two are kmObj$centers and col=c("black","red","green"). The last three, pch, cex, and lwd, should all equal 3.

```
> points(kmObj$centers, col=c("black","red","green"), pch=3, cex=3,
lwd=3)
```

We'll plot our data points several times and each time we'll just change the argument col which will show us how the R function kmeans is clustering them.

```
> plot(x,y,col=kmeans(dataFrame,6)$cluster, pch=19,cex=2)
```

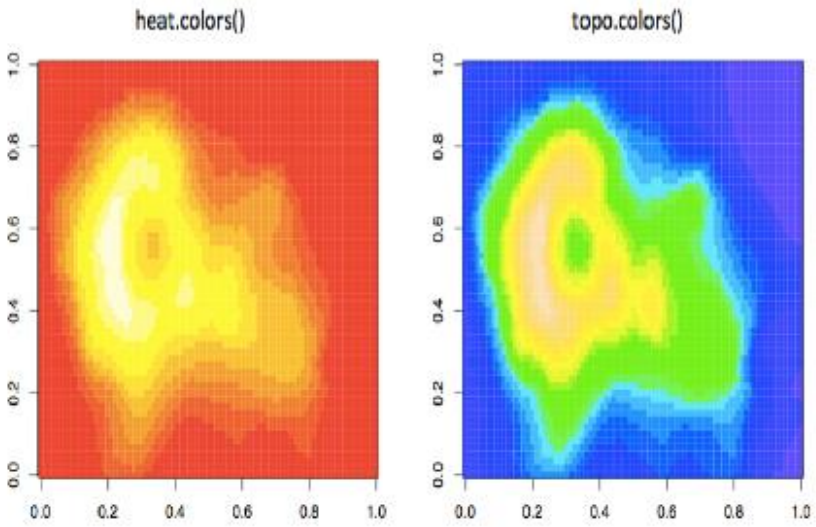
Rerun the last command

Working with colors

Package ‘jpeg’ , ‘RColorBrewer’ , ‘datasets’

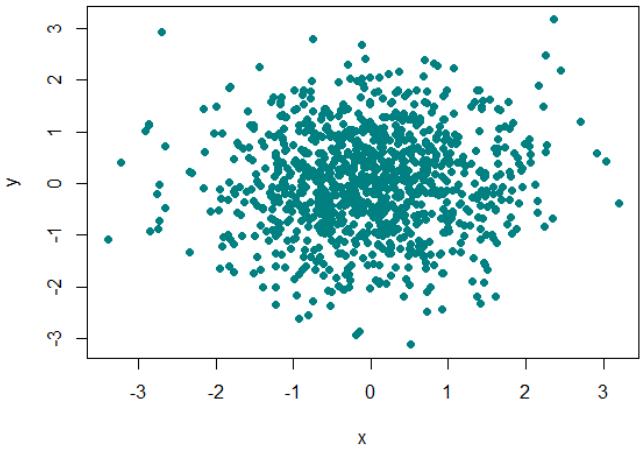
If you were plotting and just specified col=c(1:3) as one of your arguments for the basic colors (red, black, green)

Function heat.colors()



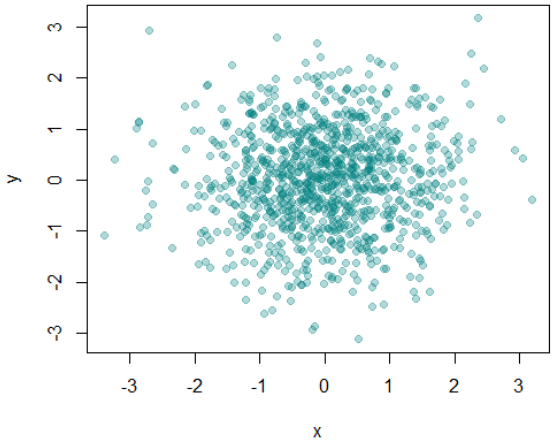
Alpha function: check if a sting variable is alphabetic or in coloring graphs alpha represents an opacity level.

- grDevices: colorRamp and colorRampPalette
- `> pal <- colorRamp(c("red", "blue"))`
- `plot(x,y,pch=19,col=rgb(0,.5,.5))`



The pch argument changes the dots' shape and takes numbers from 0 to 25

```
> plot(x,y,pch=19,col=rgb(0,.5,.5,.3))
```



Thee colorBrewer palettes can be used in conjunction with the colorRamp() and colorRampPalette() functions. You would use colors from a colorBrewer palette as your base palette,i.e., as arguments to colorRamp or colorRampPalette which would interpolate them to create new colors. As an example of this, create a new object, cols by calling the function brewer.pal with 2 arguments, 3 and "BuGn". The string "BuGn" is the second last palette in the sequential display. The 3 tells the function how many different colors we want.

ggplot2

ggplot2 (grammar of graphics) combines the best of base and lattice. It allows for multipanel (conditioning) plots (as lattice does) but also post facto annotation (as base does), so you can add titles and labels. It uses the low-level grid package (which comes with R) to draw the graphics. As part of its grammar philosophy, ggplot2 plots are composed of aesthetics (attributes such as size, shape, color) and geoms (points, lines, and bars), the geometric objects you see on the plot and FACETS are the panels used in conditional plots.

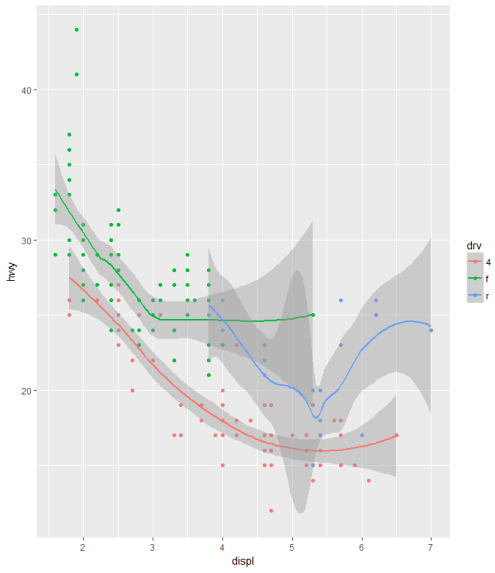
The ggplot2 package has 2 workhorse functions. The more basic workhorse function is qplot, (think quick plot), which works like the plot function in the base graphics system. It can produce many types of plots (scatter, histograms, box and whisker) while hiding tedious details from the user.

The more advanced workhorse function in the package is ggplot, which is more flexible and can be customized for doing things qplot cannot do.

```
> str(mpg)
```

(dataset=mpg, str from structure, an alternative to summary

GENERALLY the arguments have the form: variable (for countind ect), dataset, aesthetic (ie color, geom, fill)



The first argument is shown along the x-axis and the second along the y-axis. The negative trend (increasing displacement and lower gas mileage) is pretty clear. Now suppose we want to do the same plot but this time use different colors to distinguish between the 3 factors (subsets) of different types of drive (drv) in the data (front-wheel, rear-wheel, and 4-wheel). Again, qplot makes this very easy. We'll just add what ggplot2 calls an aesthetic, a fourth argument, color, and set it equal to drv

```
> qplot(displ,hwy,data=mpg,color=drv)
```

R function c(), set it equal to the concatenation of the two strings "point" and "smooth". The first refers to the data points and second to the trend lines we want plotted

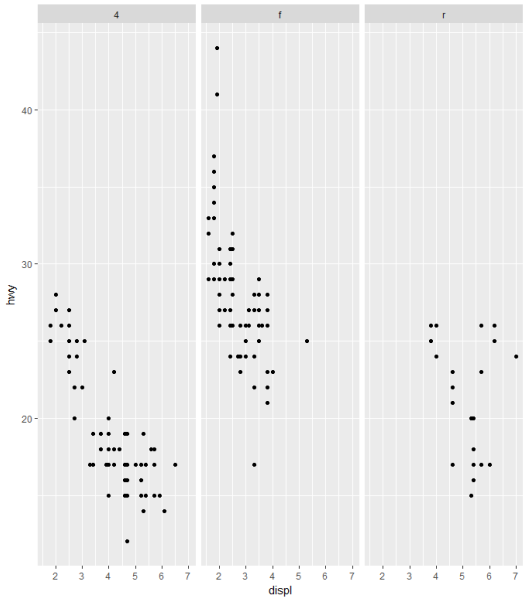
```
> qplot(displ,hwy,data=mpg,color=drv, geom = C("point","smooth"))
```

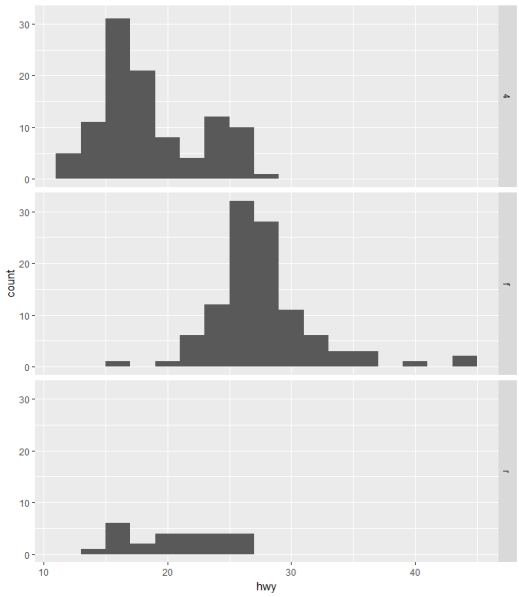
~ drv which is ggplot2's shorthand for number of rows (to the left of the ~) and the number of columns (to the right of the ~).

```
> qplot(displ, hwy, data=mpg, facets =. ~ drv)
```

~drv which is ggplot2's shorthand for a number of rows (to the left of the ~) and the number of columns (to the right of the ~). Here the . indicates a single row and drv implies 3, since there are 3 distinct drive factors

```
> qplot(hwy, data=mpg, facets = drv~ ., binwidth=2)
```





There are 3 more. STATS are statistical transformations such as binning, quantiles, and smoothing which ggplot2 applies to the data. SCALES show what coding an aesthetic map uses (for example, male = red, female = blue). Finally, the plots are depicted on a COORDINATE SYSTEM. When you use qplot these were taken care of for you. With ggplot2, the plots are built up in layers, maybe in several steps. You can plot the data, then overlay a summary (for instance, a regression line or smoother) and then add any metadata and annotations you need.

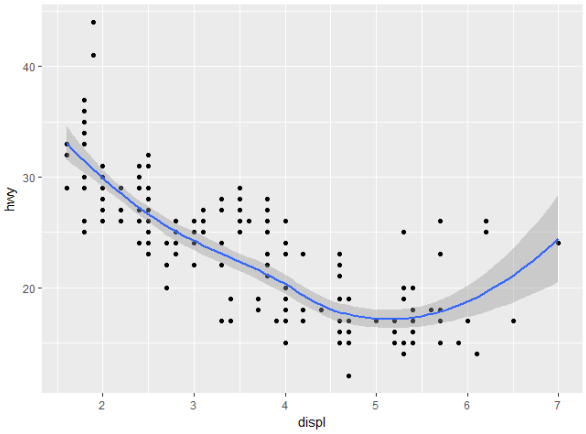
- Bulding a graph in steps

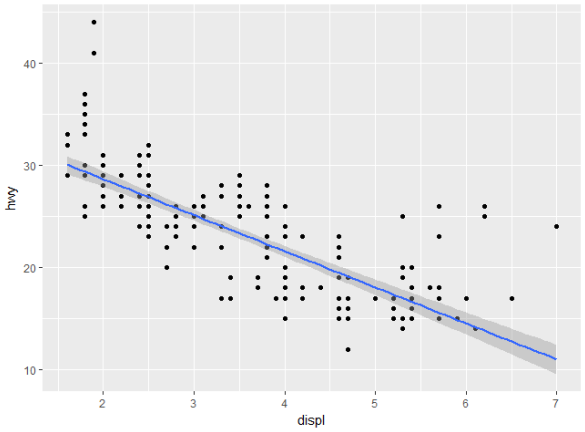
First we'll create a variable g by assigning to it the output of a call to ggplot with 2 arguments. The first is mpg (our dataset) and the second will tell ggplot what we want to plot, in this case, displ and hwy. These are what we want our aesthetics to represent so we enclose these as two arguments to the function aes.

```
> g <-ggplot(mpg, aes(displ,hwy))
> summary(g)
> g+geom_point()  without argument it just prints the graph
> g+geom_point()+geom_smooth()  to add mean and confident band
```

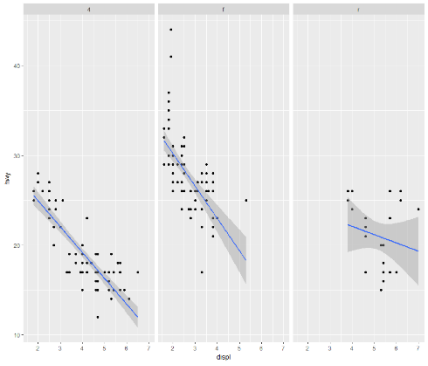
By changing the smoothing function to "lm" (linear model) ggplot2 generated a regression line through the data

```
> g+geom_point()+geom_smooth(method = "lm")
```





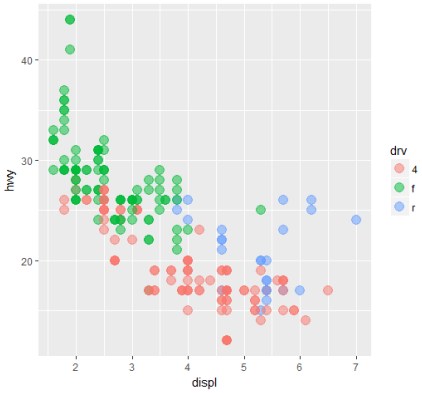

```
> g+geom_point()+geom_smooth(method = "lm")+facet_grid(. ~ drv)
```



Labels

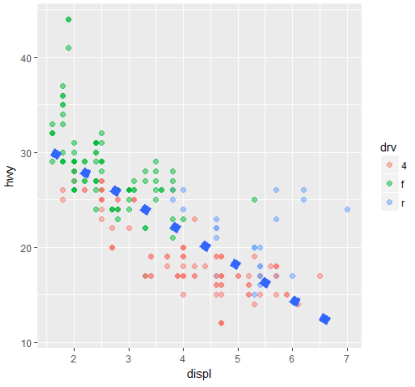
xlab(), ylab(), and ggtitle(). In addition, the function labs() is more general and can be used to label either or both axes as well as provide a title

Each of the “geom” functions (e.g., _point and _smooth) has options to modify it. Also, the function theme() can be used to modify aspects of the entire plot, e.g. the position of the legend. Two standard appearance themes are included in ggplot. These are theme_gray() which is the default theme (gray background with white grid lines) and theme_bw() which is a plainer (black and white) color scheme.




```
> g+geom_point(color="pink", size=4, alpha=1/2)
```

Shades of pink? That's the result of the alpha aesthetic. This aesthetic tells ggplot how transparent the points should be. Darker circles indicate values hit by multiple data points.



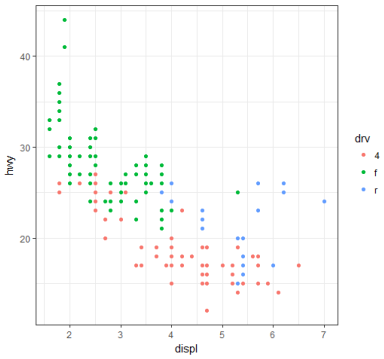

```
> g+ geom_point(size=4, alpha=1/2,
aes(color=drv))
```

MUST use the function aes since the color of the points is data dependent and not a constant as it was in the previous example.

```
> g + geom_point(aes(color = drv)) +
labs(title="Swirl Rules!") +
labs(x="Displacement",y="Hwy Mileage")
```

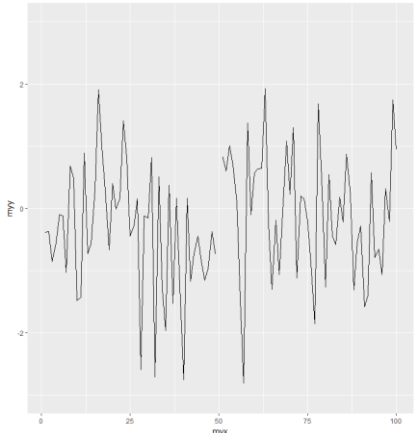
```
> g+ geom_point(size=2, alpha=1/2,
aes(color=drv))+geom_smooth(size=4, linetype=3,
method = "lm", se=FALSE)
```

The method specified a linear regression (note the negative slope indicating that the bigger the displacement the lower the gas mileage), the linetype specified that it should be dashed (not continuous), the size made the dashes big, and the se flag told ggplot to turn off the gray shadows indicating standard errors (confidence intervals).



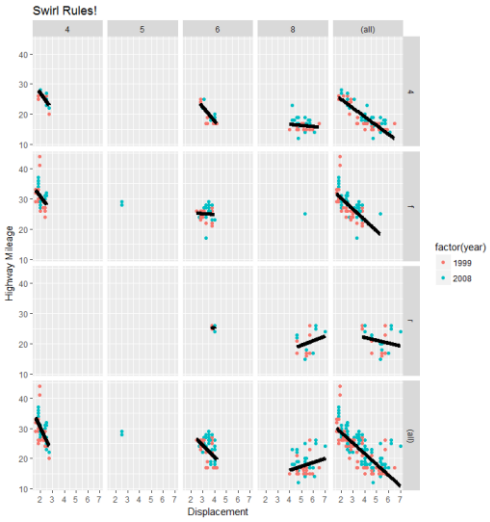

```
g + geom_point(aes(color = drv)) +
theme_bw(base_family="Times")
> plot(myx,myy, type = "l", ylim =c(-3,3))
```

The type="l" tells plot you want to display the data as a line instead of as a scatterplot



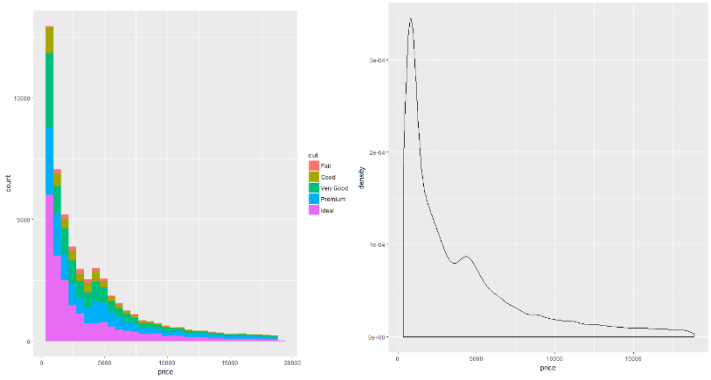

```
g <- ggplot(testdat, aes(x = myx, y = myy))
g + geom_line()
>g + geom_line() + ylim(-3,3)  to exclude an outlier

> g + geom_line() + coord_cartesian(ylim = c(-3,3))
```

The margins argument tells ggplot to display the marginal totals over each row and column, so instead of seeing 3 rows (the number of drv factors) and 4 columns (the number of cyl factors) we see a 4 by 5 display

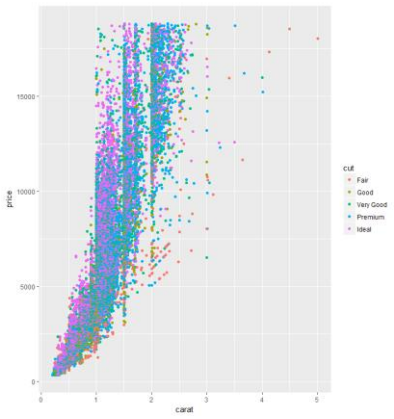



```
> g+geom_point()+facet_grid(drv~cyl, margins =
TRUE)
> g+geom_point()+facet_grid(drv~cyl, margins =
TRUE) + geom_smooth(method="lm", se=FALSE, size=2,
color="black")+labs(x="Displacement", y="Highway
Mileage", title="Swirl Rules!")
```

Range returns the minimum and maximum prices

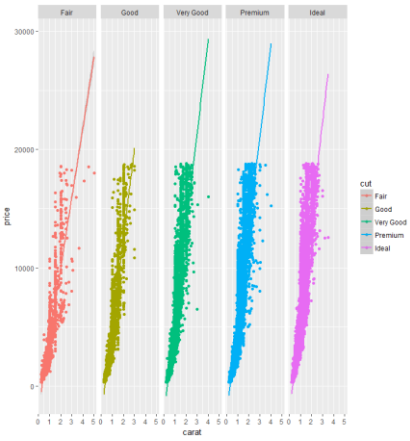



```
> qplot(price,
data=diamonds,
binwidth=18497/30,
fill=cut)
```




```
> qplot(price, data=diamonds, geom = "density")
> qplot(carat,price,data=diamonds) (scatterplot)
> qplot(carat,price,data=diamonds, shape=cut)
> qplot(carat,price,data=diamonds, color=cut)
```

```
> qplot(carat,price,data=diamonds,
color=cut)+geom_smooth(method="lm")
```
add regression lines



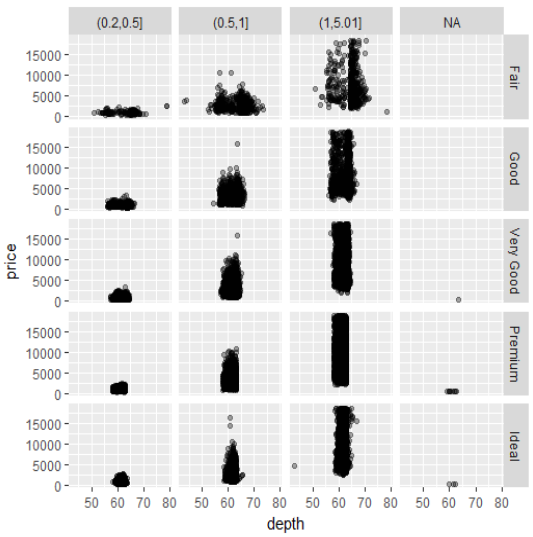

```
> qplot(carat,price,data=diamonds, color=cut,
facets = .~cut)+geom_smooth(method="lm")
```

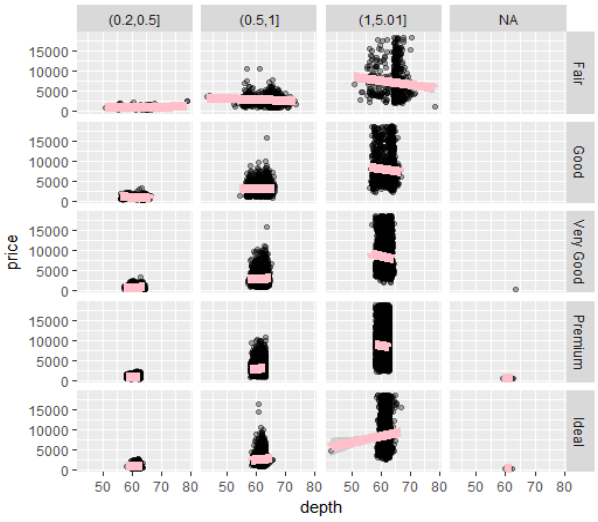
```
> cutpoints <-quantile(diamonds$carat, seq(0,1,length=4), na.rm = TRUE)
> cutpoints
      0% 33.33333% 66.66667%      100%
    0.20       0.50       1.00       5.01
```

We see a 4-long vector (explaining why length was set equal to 4). We also see that .2 is the smallest carat size in the dataset and 5.01 is the largest. One third of the diamonds are between .2 and .5 carats and another third are between .5 and 1 carat in size. The remaining third are between 1 and 5.01 carats.

Use the R command cut to label each of the 53940 diamonds in the dataset as belonging to one of these 3 factors.

```
> diamonds$car2 <-cut(diamonds$carat, cutpoints)
> g <-ggplot(diamonds, aes(depth, price))
> g +geom_point(alpha=1/3)+facet_grid(cut~car2)
> ggplot(diamonds, aes(carat,price))+geom_boxplot()+facet_grid(. ~
cut)(for boxplot)
> diamonds[myd,]  only myd rows, all columns
> g+geom_point(alpha=1/3)+facet_grid(cut~car2)+geom_smooth(method =
"lm", size=3, color="pink")
```





Principal component analysis (PCA)

This entails processes which finding subsets of variables in datasets that contain their essences.

```
> source("addPatt.R", local = TRUE)
```

Suppose we have 1000's of multivariate variables X_1, ... ,X_n. By multivariate we mean that each X_i contains many components, i.e., X_i = (X_{i1} ... , X_{im}. However, these variables (observations) and their components might be correlated to one another. As data scientists, we'd like to find a smaller set of multivariate variables that are uncorrelated AND explain as much variance (or variability) of the data as possible. This is a statistical approach.

In other words, we'd like to find the best matrix created with fewer variables (that is, a lower rank matrix) that explains the original data. This is related to data compression.

Two related solutions to these problems are PCA which stands for Principal Component Analysis and SVD, Singular Value Decomposition. This latter simply means that we express a matrix X of observations (rows) and variables (columns) as the product of 3 other matrices, i.e., X=UDV^t. This last term (V^t) represents the transpose of the matrix V.

```
http://arxiv.org/pdf/1404.1100.pdf. The paper by Jonathon Shlens of
Google Research is
| called, A Tutorial on Principal Component Analysis
> svd(scale(mat))
> prcomp(scale(mat))
```