THE LONDON SCHOOL OF ECONOMICS AND POLITICAL SCIENCE

Jon Danielsson
2020

Support notebook for *Financial Risk Forecasting*

This should serve to accompany the Book Code from Jón Daníelsson's *Financial Risk Forecasting* and the Seminar notebooks for the class. All the code used is in R.

Thanks to Alvaro Aguirre for developing the notebook and material, and to Jia Rong Fan for the information on financial data sources.

References

If you want further material, the following list provides information on various topics:

- A very short introduction to R: Concise tutorial on the basics of R
- The official introduction to R: Official documentation of the language which covers its usage
- R Programming tutorial by freeCodeCamp.org: Complete video for learning R
- R Markdown tutorial: Using R to create documents
- R Markdown cheat sheet: Practical guide to use R Markdown
- Dplyr cheat sheet: Guide on data wrangling using dplyr
- ggplot2 tutorial: Complete tutorial on ggplot2

Table of Contents

1 | Financial data sources

While financial markets generate a vast amount of data (sometimes with new observations every nanosecond), it is generally difficult and costly to get this data. Most data providers are commercial with complicated interfaces, and free data providers tend to have erratic access and errors in data.

Online data providers generally provide an *application programming interface* (API), an interface which allows users to submit data requests and receive responses. For most languages, there are often open-source or proprietary packages available which allow end-users to retrieve data using a relatively simple function call.

Today, many universities maintain subscriptions to data providers — this makes obtaining complete and accurate data much easier. Below we discuss some of the data providers that can be of use.

While this information is accurate at the time of writing (July 2020), given the rapid developments in the space, it is likely to become quickly outdated.

While we have used each of these data sources below, we have no specific recommendation.

Common Sources of Financial Data

Bloomberg

One of the most ubiquitous data sources in finance is the Bloomberg Terminal. Due to its pervasiveness throughout the industry, there are numerous packages in practically every language to access its APIs.

LSE students have access to a number of Bloomberg terminals in the library and the Masters students common rooms.

Wind

The Wind Financial Terminal (WFT) also provides market data like the Bloomberg Terminal, but with a specific focus on the Chinese financial markets. It supports APIs for MATLAB, R, C++ and Python, among others.

WRDS

The Wharton business school at University of Pennsylvania provides service called Wharton Research Data Services (WRDS) that many universities subscribe to. This provides a common interface to a number of databases, including CRSP and TAC high-frequency data. WRDS and many of its databases are available to LSE students and staff via wrds.wharton.upenn.edu.

CRSP

For historical US stock market data, one of the major sources is The Center for Research in Security Prices (CRSP, pronounced "crisp"), headquartered at the University of Chicago. LSE students can access CRSP via WRDS.

Yahoo Finance

The go-to place for many researchers requiring financial data has been finance.yahoo.com. This data can be automatically downloaded into many software packages, including Matlab, R and Python.

There are three problems with Yahoo Finance.

1. Yahoo occasionally changes how the API works, requiring updates to software;
2. It often is unavailable for days or weeks;
3. There are errors in the data. For example, UK prices, quoted in pence by convention, sometimes appear in pounds for one or two days, reverting back to pence. On other occasions, numbers are simply wrong.

EOD Historical Data

End of Day Historical Data provides fundamental data API, live and end of day historical prices for stocks, ETFs and mutual funds from exchanges all around the world.

DBnomics

DBnomics is an open source data aggregator that provides access to dozens of data providers, including several countries' statistics.

Federal Reserve Economic Data (FRED)

The [FRED Economic Reseracj] (https://fred.stlouisfed.org) is a good source for macroeconomic data including data for unemployment, GDP, interest rates, the money supply, etc. It can be accessed from DBnomics.

IEX

IEX provides access to US equity data via https://iextrading.com/developer/.

ECB FX

The European Central Bank Statistical Data Warehouse and its corresponding SDMX interface allow for retrieval of daily Euro FX data.

The entire dataset is here http://www.ecb.europa.eu/stats/eurofxref/eurofxref-hist.zip, and it can be accessed in R using

```
wget http://www.ecb.europa.eu/stats/eurofxref/eurofxref-hist.zip -O eurofxref-hist.zip
```

Alpha Vantage

Alpha Vantage provides free daily and realtime stock price data, and API access is available in both R and Python. Its data source appears to be the same as Yahoo Finance and hence subject to the same errors.

Quandl

Quandl provides a common API access for R and Python to a large number of commercial databases, and some that are free. While comprehensive, one may need to subscribe to data from a number of providers.

Fama-French Data Library:

The Fama-French Data Library provides a large amount of historical data compiled by Eugene Fama and Kenneth French. The data is updated regularly, and the Fama-French 3-factor data is especially useful for analyzing fund and portfolio performance.

Preliminary Considerations

Downloading and importing data

Data from this sources can either be downloaded directly, as a .csv or .xls file, or in most cases be imported into R using an API or package.

Symbols and Names

All stocks that are trading in the market are associated with a *ticker symbol*, serving as an identifier for the specific security. These are often specific to the particular exchange or a country of listing. This can often lead to confusion or ambiguity when a company has cross-listings — for instance, the Japanese car manufacturer Toyota is listed as `7203` on the Tokyo Stock Exchange, `TYT` on the London Stock Exchange, and `TM` on the New York Stock Exchange.

Some identical securities have different names. Depending on the source, searching for data on the S&P-500 (a major American stock market index) might require the ticker names `SPX` (Bloomberg), `^GSPC` (Yahoo Finance), `INX` (Google Finance) and so on.

When searching for a particular security from the data source, it is good practice to verify that the ticker symbol used indeed corresponds to the correct data series by checking the data description.

Tickers, ISIN and PERMNO

It is important to understand that Tickers can change over time (for example due to a merger or name change) and be recycled. This can create problems if not taken into consideration when querying financial data. There are other securities identifiers that do not change over time. The *ISIN* is the International Securities Identification Number, which is an internationally recognized 12-characters code that is unique for each stock. Unlike Tickers, the ISIN for a stock is the same regardless of the market. Another important identifier is the *PERMNO*, which is the permanent issue identifier of the CRSP dataset.

Dates

A date generally has three components, year (YYYY), month (MM) and day (DD). Depending on the granularity of data, there may also be a time component: hours, minutes, seconds and fractions of a second. Software packages often define some arbitrary date as origin. For example, R's origin date/time is set as January 1 1970 00:00:00, and all dates are relative to that.

Dates often present problems when using multiple languages or programs to carry out numerical work. Excel has two different conventions for dates depending on version — the origin year can either be 1900 or 1904 — this requires verification before use. Furthermore, Excel does not allow dates before 1900.

A date can be represented numerically in many different ways. Consider the date 13 September 2018:

Format	Example
DD-MM-YYYY	13-09-2018
MM-DD-YYYY	09-13-2018
YYYY-MM-DD	2018-09-13
YYYYMMDD	20180913

The best way is to use the YYYYMMDD convention for two reasons:

1. It can be represented as an integer, not as a string, making data handling more convenient
2. It sorts naturally (in chronological order) even as integers

Best practice in data downloading

Given the need to ensure that accurate data is used, it may be best to have a separate step in downloading and running the data. Get data from a provider, save it as a CSV (or Excel) file and then have a separate run for processing the data.

There are a few reasons why this is preferable:

1. Limits for API calls:
   - Many data providers place limits on the amount/speed of data downloads
2. Minimize time spent importing data
   - Especially true for large datasets, since CSV/xlsx reading is relatively quick
3. Quick checking for data omissions
   - Useful to have a pre-analysis check of data quality using plots and tables

Never manipulate your data in Excel. Perform any analysis in R or the programming language of your choice. This will make it tractable and can prevent mistakes.

2 | Control Flow in R

In many cases we will want to repeat a piece of code several times, or execute part of our code only if some conditions are met. Programming languages allow us to do that quite easily with loops and conditionals, which are specific statements that give instructions to computers. In this section we will cover the most important loops and conditionals in R.

For more details you can check the official introduction to R (Section

For statements

A *for loop* evaluates a piece of code for different values. Usually, we think of *for loops* in two types, depending on the values we are evaluating: Loops over sequences, and loops over elements.

Looping over sequences

Probably the most common way to build a *for* statement is to loop over a numeric sequence. For example, if we wanted to `print()` the integers from 1 to 5:

In [1]:
```r
for (i in 1:5) {
    print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

In line 1, we are telling R that we want to create an integer sequence from 1 to 5, and use the name `i` to refer to values from this sequence. Then, line 2 is executed for every possible value specified in line 1.

Basically we are telling R:

1. Create a sequence from a to b
2. Take the first element of the sequence and assign it to "i"
3. Execute the code between { } using the current value of "i"
4. When done, check if there are more elements in the sequence
   4.1 If there are, take the next element and assign that value to "i", then go back to step 3
   4.2 If there are not, exit the loop

Recall the syntax `1:5` means "Create a sequence from 1 to 5", and if we don't specify the step, it takes 1 by default:

In [2]:
```r
1:5
```

1 2 3 4 5

This is equivalent to doing:

In [3]:
```r
seq(1,5,1)
```

1 2 3 4 5

So our *for loop* could also be written as:

In [4]:
```r
for (i in seq(1,5,1)) {
    print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

It is a convention to create integer sequences using `first_element:last_element`

The main usage of this type of loops is working with **indexes**. For example, consider we have a vector of 500 elements filled with `NA`:

In [5]:
```r
vec <- rep(NA, 500)
```

Let's say we want to replace all odd elements with a 1, and keep the even elements as `NA`. We could build a *for loop* over the sequence `seq(1, 500, by = 2)`

In [6]:
```r
for (i in seq(1, 500, by = 2)) {     # Loop over i = 1, 3, 5, ...
    vec[i] <- 1                      # Replace the i-th element in vec for 1
}
```

An interesting application for using *for loops* over sequences is constructing the Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21, ...

Recall the j-th term of the Fibonacci sequence is the sum of the j-1-th and j-2-th terms.

Let's say we want to create a vector with the first 50 terms of the Fibonacci sequence:

In [7]:
```r
fibonacci <- rep(NA, 50)                    # Create the empty vector
fibonacci[1:2] <- 1                          # The first two values are 1
for (i in 3:50){                        # Loop over the indexes 3 to 100
    fibonacci[i] <- fibonacci[i-1] + fibonacci[i-2]  # Replace the i-th term by the sum of the two prev
ious terms
}
```

In [8]:
```r
fibonacci
```

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 4807526976 7778742049 12586269025

We could put this in to a function that gives us the i-th fibonacci number:

In [9]:
```r
fibonacci <- function(N) {
    fibo <- rep(NA, N)
    fibo[1:2] <- 1
    for (i in 3:N) {
        fibo[i] <- fibo[i-1] + fibo[i-2]
    }
    paste0("The ", N, "-th Fibonacci number is: ", fibo[N])
}

fibonacci(10)
fibonacci(50)
```

'The 10-th Fibonacci number is: 55'

'The 50-th Fibonacci number is: 12586269025'

Or, for example, if you have a vector with different model names:

In [10]:
```r
model <- c("GARCH", "EWMA", "HS")
```

And another vector with a value associated to each model respectively:

In [11]:
```r
violation_ratio <- c(1.06, 0.95, 0.88)
```

You can output this as:

In [12]:
```r
for (i in 1:3){
    cat("The value for the", model[i], "model is:", violation_ratio[i], "\n")
}
```

```
The value for the GARCH model is: 1.06
The value for the EWMA model is: 0.95
The value for the HS model is: 0.88
```

Looping over elements

The other common way to think of *for loops* although technically analogous, is to loop over elements of a vector, which don't necessarily have to be numbers. The syntax is similar.

For example, consider the `model` vector defined above:

In [13]:
```r
model
```

'GARCH' 'EWMA' 'HS'

You can loop over the vector `model` by doing `for (element in model)`. Note that `element` is just a placeholder, it can be anything else. Let's say you want to say hello to everyone on the list:

In [14]:
```r
for (model_type in model){
    cat("My favorite model is", model_type, "\n")
}
```

```
My favorite model is GARCH
My favorite model is EWMA
My favorite model is HS
```

Another example:

In [15]:
```r
estimation_windows = c(300, 500, 1000, 2000)   # Vector of different "windows"
S = 5000
rand <- rnorm(S)                              # Create a vector of random numbers of length 5k
for (window in estimation_windows){             # Loop over estimation windows
    # Get last window-length elements of rand
    tmp <- rand[(S-window):S]
    cat("Average of last", window, "elements: ", mean(tmp), "\n")
}
```

```
Average of last 300 elements:  -0.05571649
Average of last 500 elements:  -0.03766916
Average of last 1000 elements:  0.01084457
Average of last 2000 elements:  0.01729325
```

If statements

A conditional, or *if*, statement, evaluates a piece of code that can be either `TRUE` or `FALSE` and takes a different action depending on the value. If the condition is `TRUE`, then it will execute the piece of code inside the brackets { }, if it is `FALSE`, it will not. You can then specify a different action to be performed if the condition is `FALSE`, which would go in an *else* statement. You can even include *else if* statements to evaluate another condition if the first one was not met.

A classic example:

In [16]:
```r
X = 5

if (X > 0) {
    print("Positive")
} else if (X < 0) {
    print("Negative")
} else {
    print("Zero")
}
```

```
[1] "Positive"
```

With this we can easily define a function that checks if a number is even:

In [17]:
```r
is_even <- function(x){
    if (x %% 2 == 0) {
        cat(x, "is even", "\n")
    } else {
        cat(x, "is odd", "\n")
    }
}

is_even(4)
is_even(7)
```

```
4 is even
7 is odd
```

While loops

A *while* loop will evaluate a condition and as long as it is `TRUE`, it will execute a piece of code. **NOTE:** you need to make sure that the loop will not run indefinitely. To do this, at some point in the loop the condition needs to be set to `FALSE`.

A basic usage is:

In [18]:
```r
a <- 1

while (a < 6) {
    print(a)
    a = a + 1
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Normally, you use *while* loops when the number of times the loop has to be executed is not pre-determined. For example, you can create a simple function that uses a *while* loop to tell you how many times you need to halve a number before its absolute value is less than 0.5:

In [19]:
```r
less_half <- function(x){
    steps <- 0
    while (abs(x) > 0.5) {
        x = x/2
        steps = steps + 1
    }
    cat("Number of steps:", steps, "\n")
}

less_half(10)
less_half(0.2)
less_half(-500)
```

```
Number of steps: 5
Number of steps: 0
Number of steps: 10
```

3 | Data frames handling

Objects in R can be of different *classes*. For example, we can have a vector, which is an ordered array of observations of data of the same type (numbers, characters, logicals). We can also have a matrix, which is a rectangular arrangement of elements of the same data type. You can check what *class* an object is by running `class(object)`.

A data frame is a two-dimensional structure in which each column contains values of one variable and each rows contains one set of values, or "observation" from each column. It is perhaps the most common way of storing data in R and the one we will use the most.

One of the main advantaged of a data frame in comparison to a matrix, is that each column can have a different data type. For example, you can have one column with numbers, one with text, one with dates, and one with logicals, whereas a matrix limits you to only one data type. Keep in mind that a data frame needs all its columns to be of the same length.

Creating Data Frames

There are different ways to create a data frame. We will focus on three:

1. Reading a file

In finance it is very common to have a `.csv` file with the data you want to analyze. For example, this data could be the output of any of the financial data sources we mentioned above. We can import directly a `.csv` file into R as a `data.frame` object using the function `read.csv()`.

The file *stocks.csv* holds daily returns for 6 different companies for the years 2015 to 2019. It has been downloaded from the CRSP database. Let's import it:

In [20]:
```r
stocks <- read.csv("stocks.csv")    # Importing data to a dataframe
```

After importing a file, it is recommended to see how it looks like before starting to perform any type of analysis. For this, we can use the function `head()` to see the first elements of our object:

In [21]:
```r
head(stocks)
```

MSFT	XOM	GE	JPM	INTC	C	date
0.006651827	0.004101577	-0.0083447205	0.004941769	0.001927142	0.002768165	2015-01-02
-0.009346543	-0.027743311	-0.0185265618	-0.031537108	-0.011340056	-0.032022284	2015-01-05
-0.014678200	-0.005330180	-0.0217804820	-0.026271083	-0.018812857	-0.035839635	2015-01-06
0.012624969	0.010082005	0.0004149139	0.001524837	0.020758053	0.009227297	2015-01-07
0.028993594	0.016507990	0.0119710601	0.022099986	0.018430117	0.014935902	2015-01-08
-0.008440521	-0.001410995	-0.0140502440	-0.017539929	0.001906182	-0.022586158	2015-01-09

It is **very** important to check the documentation of the functions we are using to make sure we are doing things right. For example, we need to be careful if:

- The CSV file has headers or not - A *header* is the title of a column. We can have a csv file where the first row includes the column titles, or a csv with no titles
- The separator is a comma (,), a semicolon (;), a tab (\t) or another symbol
- We are getting data from a country where the decimal symbol is a comma (,) instead of a period (.)

By checking the documentation of a function, we will see how to deal with each of these specific cases and more. To do so, run `?read.csv`

In [22]:
```r
?read.csv
```

Now let's check that our variable `df` is actually a data frame:

In [23]:
```r
class(stocks)
```

'data.frame'

We can check the *structure* of a data frame using the function `str()`. This can be quite useful to see what every column of the data frame holds, and what type of data it has:

In [24]:
```r
str(stocks)
```

```
'data.frame':   1258 obs. of  7 variables:
 $ MSFT: num  0.00665 -0.00935 -0.01468 0.01262 0.02899 ...
 $ XOM : num  0.0041 -0.02774 -0.00533 0.01008 0.01651 ...
 $ GE  : num  -0.008345 -0.018527 -0.02178 0.000415 0.011971 ...
 $ JPM : num  0.00494 -0.03154 -0.02627 0.00152 0.0221 ...
 $ INTC: num  0.00193 -0.01134 -0.01881 0.02076 0.01843 ...
 $ C   : num  0.00277 -0.03202 -0.03584 0.00923 0.01494 ...
 $ date: Factor w/ 1258 levels "2015-01-02","2015-01-05",..: 1 2 3 4 5 6 7 8 9 10 ...
```

We see the data frame has 8 columns and 1258 rows. The *structure* function shows us the names of each column, along with the data type it holds, and some observations. We see there is a column called X that serves as an index, with variable type integer, a column for every stock, which holds numeric data, and finally a column for dates, which is of type factor.

We can also check parts of the structure with different functions:

In [25]:
```r
dim(stocks)
nrow(stocks)
ncol(stocks)
colnames(stocks)
```

1258 7

1258

7

'MSFT' 'XOM' 'GE' 'JPM' 'INTC' 'C' 'date'

2. Creating a Data Frame from scratch

In some cases you might want to create a data frame from a list of vectors. This can easily be done with the `data.frame()` function:

In [26]:
```r
df <- data.frame(col1 = 1:3,
                 col2 = c("A", "B", "C"),
                 col3 = c(TRUE, TRUE, FALSE),
                 col4 = c(1.0, 2.2, 3.3))
```

You have to specify the name of each column, and what goes inside it. Note that all vectors need to be the same length. We can now check the structure:

In [27]:
```r
str(df)

dim(df)
colnames(df)
```

```
'data.frame':   3 obs. of  4 variables:
 $ col1: int  1 2 3
 $ col2: Factor w/ 3 levels "A","B","C": 1 2 3
 $ col3: logi  TRUE TRUE FALSE
 $ col4: num  1 2.2 3.3
```

3 4

'col1' 'col2' 'col3' 'col4'

Note that the elements of `col2` are considered `Factors`, since it is the default in `data.frame()`. We can check the documentation and use an option to keep these are `chr`:

In [28]:
```r
df <- data.frame(col1 = 1:3,
                 col2 = c("A", "B", "C"),
                 col3 = c(TRUE, TRUE, FALSE),
                 col4 = c(1.0, 2.2, 3.3),
                stringsAsFactors = FALSE)

str(df)
```

```
'data.frame':   3 obs. of  4 variables:
 $ col1: int  1 2 3
 $ col2: chr  "A" "B" "C"
 $ col3: logi  TRUE TRUE FALSE
 $ col4: num  1 2.2 3.3
```

3. Transforming a different object into a Data Frame

You might want to transform a matrix into a data frame (or viceversa). For example, you need an object to be a matrix to perform linear algebra operations, but you would like to keep the results as a data frame after the operations. You can easily switch from matrix to data frame using `as.data.frame()` (and analogously, from data frame to matrix with `as.matrix()`, however remember all columns need to have the same data type to be a matrix).

For example, let's say we have the matrix:

In [29]:
```r
my_matrix <- matrix(1:10, nrow = 5, ncol = 2, byrow = TRUE)
class(my_matrix)
my_matrix
```

'matrix'

1	2
3	4
5	6
7	8
9	10

We can now transform it into a data frame:

In [30]:
```r
df <- as.data.frame(my_matrix)
class(df)
df
str(df)
```

'data.frame'

V1	V2
1	2
3	4
5	6
7	8
9	10

```
'data.frame':   5 obs. of  2 variables:
 $ V1: int  1 3 5 7 9
 $ V2: int  2 4 6 8 10
```

And we can change the column names:

In [31]:
```r
colnames(df) <- c("Odd", "Even")
str(df)
```

```
'data.frame':   5 obs. of  2 variables:
 $ Odd : int  1 3 5 7 9
 $ Even: int  2 4 6 8 10
```

Subsetting Data Frames

There are many different ways to subset a data frame and access its elements. We can subset a data frame by its column/row position, by names, using logicals, etc. Each way can be useful for a different situation:

In [32]:
```r
head(stocks)
```

MSFT	XOM	GE	JPM	INTC	C	date
0.006651827	0.004101577	-0.0083447205	0.004941769	0.001927142	0.002768165	2015-01-02
-0.009346543	-0.027743311	-0.0185265618	-0.031537108	-0.011340056	-0.032022284	2015-01-05
-0.014678200	-0.005330180	-0.0217804820	-0.026271083	-0.018812857	-0.035839635	2015-01-06
0.012624969	0.010082005	0.0004149139	0.001524837	0.020758053	0.009227297	2015-01-07
0.028993594	0.016507990	0.0119710601	0.022099986	0.018430117	0.014935902	2015-01-08
-0.008440521	-0.001410995	-0.0140502440	-0.017539929	0.001906182	-0.022586158	2015-01-09

In [33]:
```r
# Subsetting by row numbers
stocks[2:3,]
```

	MSFT	XOM	GE	JPM	INTC	C	date
2	-0.009346543	-0.02774331	-0.01852656	-0.03153711	-0.01134006	-0.03202228	2015-01-05
3	-0.014678200	-0.00533018	-0.02178048	-0.02627108	-0.01881286	-0.03583964	2015-01-06

In [34]:
```r
# Subsetting by column number
head(stocks[, 4])
```

0.00494176931922958 -0.0315371077982067 -0.0262710827191262 0.00152483684516575 0.0220999863448632 -0.0175399291295306

In [35]:
```r
# Subsetting by col names
head(stocks[c("MSFT", "GE")])
```

MSFT	GE
0.006651827	-0.0083447205
-0.009346543	-0.0185265618
-0.014678200	-0.0217804820
0.012624969	0.0004149139
0.028993594	0.0119710601
-0.008440521	-0.0140502440

In [36]:
```r
# Subsetting using $
head(stocks$MSFT)
```

0.00665182746034677 -0.00934654316964306 -0.0146781996320895 0.0126249686402503 0.028993593997765 -0.00844052118960295

In [37]:
```r
# Subsetting both rows and columns
stocks[1:2, c("JPM", "C")]
```

JPM	C
0.004941769	0.002768165
-0.031537108	-0.032022284

In [38]:
```r
# Subsetting using a logical
head(stocks[stocks$JPM > 0, c("JPM", "date")])
```

	JPM	date
1	0.0049417693	2015-01-02
4	0.0015248368	2015-01-07
5	0.0220999863	2015-01-08
8	0.0001699856	2015-01-13
11	0.0169495415	2015-01-16
13	0.0032257915	2015-01-21

In [39]:
```r
# Subsetting using a vector
v <- c(1, 2, 4, 8, 16)
stocks[v,]
```

	MSFT	XOM	GE	JPM	INTC	C	date
1	0.006651827	0.004101577	-0.0083447205	0.0049417693	0.001927142	0.002768165	2015-01-02
2	-0.009346543	-0.027743311	-0.0185265618	-0.0315371078	-0.011340056	-0.032022284	2015-01-05
4	0.012624969	0.010082005	0.0004149139	0.0015248368	0.020758053	0.009227297	2015-01-07
8	-0.005270867	-0.003659688	-0.0050165619	0.0001699856	-0.002735739	-0.007171655	2015-01-13
16	-0.003609506	0.009526479	0.0044829366	0.0015867405	-0.017717024	0.005537639	2015-01-26

Handling

Let's store the stock returns for JP Morgan and dates in a different variable:

```r
# ARCH(1) MLE

LL.ARCH1 <- function(par, x) {

    omega <- par[1] # First element of par
    alpha <- par[2] # Second element of par

    T <- length(x)       # Number of rows
    loglikelihood <- -(T-1)/2 * log(2*pi)

    # Loop (create the sum) and add every instance fo loglikelihood
    for(i in 2:T){
        loglikelihood = loglikelihood - 1/2 * (log(omega + alpha*x[i-1]^2)+x[i]^2/(omega + alpha*x[i-1]
^2))
    }

    return(-loglikelihood) # output the log likelihood
}

res <- optim(c(0.1,0.5) , LL.ARCH1, gr = NULL, ret, hessian = T) # Optimization

cat("Omega:", res$par[1], "\n",
   "Alpha:", res$par[2])
```

```
Omega: 0.9956736
 Alpha: 0.3486696
```

GARCH(1,1)

In a GARCH(1,1) model, the conditional volatility follows:

$$\sigma_t^2 = \omega + \alpha Y_{t-1}^2 + \beta\sigma_{t-1}^2$$

The conditional density is:

$$f(y_2|y_1) = \frac{1}{\sqrt{2\pi(\omega + \alpha y_1^2 + \beta\hat{\sigma}_1^2)}} exp\left(-\frac{1}{2}\frac{y_2^2}{\omega + \alpha y_1^2 + \beta\hat{\sigma}_1^2}\right)$$

So the log-likelihood function is:

$$\log\mathcal{L} = -\frac{T-1}{2}\log(2\pi) - \frac{1}{2}\sum_{t=2}^{T}\left(\log(\omega + \alpha y_{t-1}^2 + \beta\hat{\sigma}_{t-1}^2) + \frac{y_t^2}{\omega + \alpha y_{t-1}^2 + \beta\hat{\sigma}_{t-1}^2}\right)$$

Writing the function in R:

```r
# GARCH(1,1) MLE

LL.GARCH1_1 <- function(par, x) {

    omega <- par[1] # First element of par
    alpha <- par[2] # Second element of par
    beta <- par[3]  # Third element of par

    T <- length(x)       # Number of rows
    loglikelihood <- -(T-1)/2 * log(2*pi)
    sigma_2 <- rep(NA, T)
    sigma_2[1] <- var(x) # Initialize the sigma with unconditional volatility

    # Loop (create the sum) and add every instance to loglikelihood
    for(i in 2:T){
        loglikelihood = loglikelihood - 1/2 * (log(omega + alpha*x[i-1]^2 + beta*sigma_2[i-1])
                                               + x[i]^2/(omega + alpha*x[i-1]^2 + beta*sigma_2[i-1]))

        sigma_2[i] <- omega + alpha*x[i-1]^2 + beta*sigma_2[i-1]     # Update sigma
    }

    return(-loglikelihood) # output the log likelihood
}

res <- optim(c(0.1, 0, 0) , LL.GARCH1_1, gr = NULL, ret, hessian = T,
             method = 'L-BFGS-B', lower = c(0,0,0)) # Optimization

cat("Omega:", res$par[1], "\n",
   "Alpha:", res$par[2], "\n",
   "Beta:", res$par[3])

## Note that the values obtained in GARCH ML will depend on the optimization method used
```

```
Omega: 0.01810722
 Alpha: 0.1030158
 Beta: 0.883244
```

Some important components of the optimization process are:

- Value of σ_1^2: We need to assign a value to σ_1^2, the first iteration of the conditional volatility. General, we choose the unconditional variance of the data for this
- Optimization method: We have chosen the L-BFGS-B algorithm, since it allows us to place bounds on the parameters. We have specified non-negativity by setting `lower = c(0,0,0)`, since this is desired in GARCH
- Initial values of the parameters: Optimization works iteratively. You need to assign values to initialize the parameters, making sure these are compliant with the restrictions of the function and the parameter bounds. This will have a small effect on the parameter values, but can have a large effect in computing time

Optimization packages

In this section we used the function `optim()` to fit the maximum likelihood estimation. This was enough for our purposes, but if you have to carry on more complex optimization problems, `optim()` can fail. We recommend the package *nloptr*, which currently is the state of the art for minimizing functions in non-linear settings.

7 | *rugarch* and *rmgarch* usage

The packages that we use in the course for estimating GARCH models are *rugarch* for univariate models, and *rmgarch* for multivariate models. The complete documentation of the packages is:

- rugarch: https://cran.r-project.org/web/packages/rugarch/rugarch.pdf
- rmgarch: https://cran.r-project.org/web/packages/rmgarch/rmgarch.pdf

Both packages were developed by Alexios Galanos and are constantly mantained and updated. The development code can be found here: https://bitbucket.org/alexiosg/ If you are curious on learning more programming, you are encourage to browse the page to see how the package was created.

Univariate GARCH models with *rugarch*

To estimate a univariate GARCH model you need to follow two steps:

1. Create an object of the `uGARCHspec` class which is the specification of the model you want to estimate. This includes the type of model (standard, asymmetric, power, etc), the GARCH order, the distribution, and model for the mean
2. Fit the specified model to the data

1. ugarchspec( )

First let's take a look at `ugarchspec()`, which is the function to create an instance of the `uGARCHspec` class. The complete syntax with its default values is:

```
ugarchspec(variance.model = list(model = "sGARCH", garchOrder = c(1, 1),
submodel = NULL, external.regressors = NULL, variance.targeting = FALSE),
mean.model = list(armaOrder = c(1, 1), include.mean = TRUE, archm = FALSE,
archpow = 1, arfima = FALSE, external.regressors = NULL, archex = FALSE),
distribution.model = "norm", start.pars = list(), fixed.pars = list(), ...)
```

You can check the details of this function and what every argument means by running `?ugarchspec`.

For the purposes of our course, we are going to focus on three arguments of the `ugarchspec()` function:

`variance.model`

This argument takes a `list` with the specifications of the GARCH model. Its most important components are:

- `model`: Specify the type of model, currently implemented models are “sGARCH”, “fGARCH”, “eGARCH”, “gjrGARCH”, “apARCH” and “iGARCH” and “csGARCH”
- `garchOrder`: Specify the ARCH(q) and GARCH(p) orders

Other components includes options to do `variance.targeting` or specify `external regressors`.

`mean.model`

This argument takes a `list` with the specifications of the mean model, if assumed to have one. A traditional assumption is that the model has zero mean, in which case we specify `armaOrder = c(0,0), include.mean = FALSE`. However, it is also common to assume that the mean follows a certain ARMA process.

`distribution.model`

Here you can specify the conditional density to use for innovations. The default is a normal distribution, but we can specify the use of a Student-t distribution by setting the value to `std`.

2. ugarchfit( )

Once the specification has been created, you can *fit* this to the data using `ugarchfit( spec = ..., data = ...)`. The result will be an object of the class `uGARCHfit`, which is a list that contains useful information which will be shown below.

GARCH(1,1)

Let's use *rugarch* to estimate a GARCH(1,1) model.

First, let's load the library, and the S&P500 data:

```r
library(rugarch)
sp <- read.csv("sp500.csv")
ret <- diff(log(sp$price))*100
ret <- ret[2:length(ret)]
```

```
Loading required package: parallel

Registered S3 method overwritten by 'xts':
  method     from
  as.zoo.xts zoo


Attaching package: 'rugarch'


The following object is masked from 'package:stats':

    sigma
```

Now let's use `ugarchspec()` to specify the type of model we want. In this case, we want to use a GARCH(1,1), so the `garchOrder` in `variance.model` will be set as `c(1,1)`. Note that even if we only have one component, we need to put it inside a `list`.

We will assume zero mean, so we need to include this into `mean.model`:

```r
spec1 <- ugarchspec(
    variance.model = list(garchOrder = c(1,1)),
    mean.model = list(armaOrder = c(0,0), include.mean = FALSE)
)
```

We can call `spec1` to see what is inside:

```r
spec1
```

```
*---------------------------------*
*       GARCH Model Spec          *
*---------------------------------*

Conditional Variance Dynamics
------------------------------------
GARCH Model             : sGARCH(1,1)
Variance Targeting      : FALSE

Conditional Mean Dynamics
------------------------------------
Mean Model              : ARFIMA(0,0,0)
Include Mean            : FALSE
GARCH-in-Mean           : FALSE

Conditional Distribution
------------------------------------
Distribution    :  norm
Includes Skew   :  FALSE
Includes Shape  :  FALSE
Includes Lambda :  FALSE
```

Now we can fit the specified model to our data using the `ugarchfit()` function:

```r
garch1_1 <- ugarchfit(spec = spec1, data = ret)
```

We can check the class of this new object:

```r
class(garch1_1)
```

'uGARCHfit'

Objects from the `uGARCHfit` class have two *slots*. You can think of every slot as a list of components. The two slots are:

1. `@model`
2. `@fit`

To access a slot you need to use the syntax: `object@slot`

Let's explore these to understand better:

@model

The *model* slot includes all the information that was needed to estimate the GARCH model. This includes both the model specification and the data. To see every that is included in this slot, you can use the `names()` function:

```r
names(garch1_1@model)
```

'modelinc' 'modeldesc' 'modeldata' 'pars' 'start.pars' 'fixed.pars' 'maxOrder' 'pos.matrix' 'fmodel' 'pidx' 'n.start'

And you can access each element with the `$` sign. For example, let's see what is inside `pars`:

```r
garch1_1@model$pars
```

	Level	Fixed	Include	Estimate	LB	UB
mu	0.00000000	0	0	0	NA	NA
ar	0.00000000	0	0	0	NA	NA
ma	0.00000000	0	0	0	NA	NA
arfima	0.00000000	0	0	0	NA	NA
archm	0.00000000	0	0	0	NA	NA
mxreg	0.00000000	0	0	0	NA	NA
omega	0.01802185	0	1	1	2.220446e-16	1454.914
alpha1	0.10279939	0	1	1	0.000000e+00	1.000
beta1	0.88352744	0	1	1	0.000000e+00	1.000
gamma	0.00000000	0	0	0	NA	NA
eta1	0.00000000	0	0	0	NA	NA
eta2	0.00000000	0	0	0	NA	NA
delta	0.00000000	0	0	0	NA	NA
lambda	0.00000000	0	0	0	NA	NA
vxreg	0.00000000	0	0	0	NA	NA
skew	0.00000000	0	0	0	NA	NA
shape	0.00000000	0	0	0	NA	NA
ghlambda	0.00000000	0	0	0	NA	NA
xi	0.00000000	0	0	0	NA	NA

We have a matrix with all the possible parameters to be used in fitting a GARCH model. We see there is a value of 1 for the parameters that were included in our GARCH specification (omega, alpha1 and beta1). The matrix also includes what are the lower and upper bounds for these parameters, which has nothing to do with our particular data, only with what is expected from the model.

We can also see the model description:

```r
garch1_1@model$modeldesc
```

$distribution
'norm'
$distno
1
$vmodel
'sGARCH'

And inside `garch1_1@model@modeldata` we will find the data vector we have used when fitting the model. You can think of the `@model` slot as everything that R needs to know in order to be able to estimate the model.

@fit

Inside the *fit* slot we will find the *estimated* model, including the coefficients, likelihood, fitted conditional variance, and more. Let's check everything that is included:

```r
names(garch1_1@fit)
```

'hessian' 'cvar' 'var' 'sigma' 'condH' 'z' 'LLH' 'log.likelihoods' 'residuals' 'coef' 'robust.cvar' 'A' 'B' 'scores' 'se.coef' 'tval' 'matcoef' 'robust.se.coef' 'robust.tval' 'robust.matcoef' 'fitted.values' 'convergence' 'kappa' 'persistence' 'timer' 'ipars' 'solver'

Let's see the coefficients, along with their standard errors, in `matcoef`:

Note: You will see these values are very similar to what we found using Maximum Likelihood. Since it is an optimization problem, we can expect to see some small differences

```r
garch1_1@fit$matcoef
```

	Estimate	Std. Error	t value	Pr(>|t|)
omega	0.01802185	0.002790007	6.459428	1.050997e-10
alpha1	0.10279939	0.009098806	11.298119	0.000000e+00
beta1	0.88352744	0.009724544	90.855407	0.000000e+00

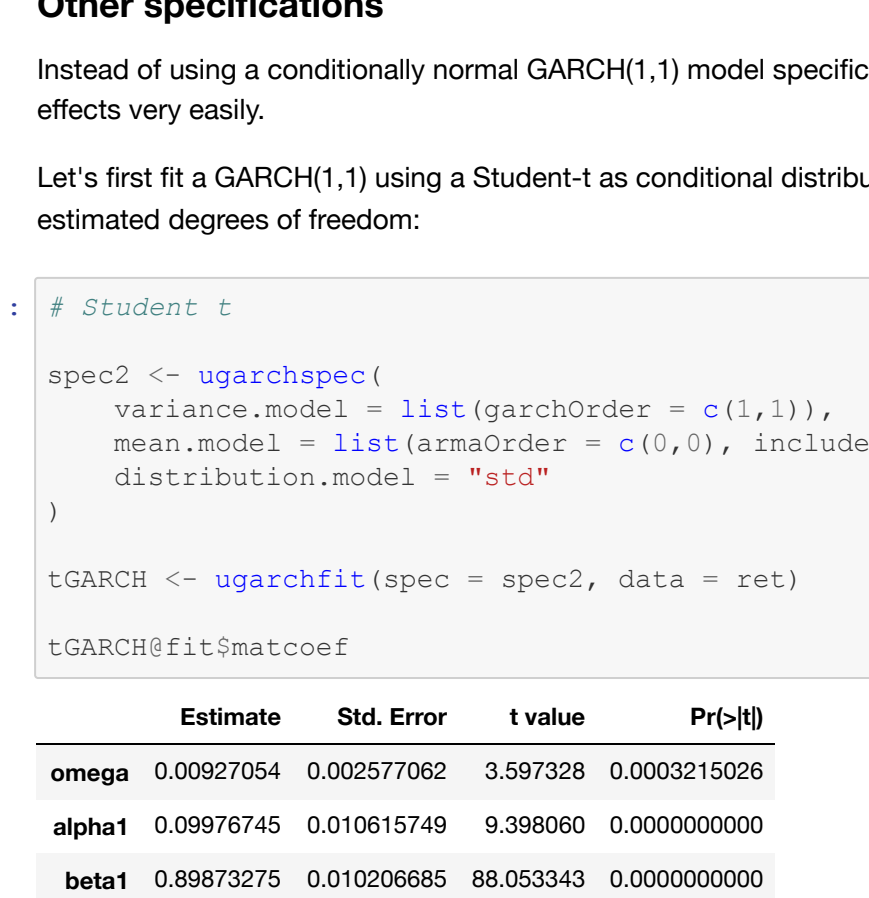
We can also see the log-likelihood:

```r
garch1_1@fit$LLH
```

-6542.99177744059

This makes it easy to extract the estimated conditional volatility:

```r
plot(garch1_1@fit$sigma, type = "l")
```



We can also access the coefficients are likelihood using the following functions:

```r
coef(garch1_1)
```

omega	0.0180218480714706
alpha1	0.102799391800216
beta1	0.883527438351233

```r
likelihood(garch1_1)
```

-6542.99177744059

Other specifications

Instead of using a conditionally normal GARCH(1,1) model specification, we can use a t-distribution or introduce asymmetric and power effects very easily.

Let's first fit a GARCH(1,1) using a Student-t as conditional distribution. We can see that the coefficients now include `shape`, which is the estimated degrees of freedom:

```r
# Student t

spec2 <- ugarchspec(
    variance.model = list(garchOrder = c(1,1)),
    mean.model = list(armaOrder = c(0,0), include.mean = FALSE),
    distribution.model = "std"
)

tGARCH <- ugarchfit(spec = spec2, data = ret)

tGARCH@fit$matcoef
```

	Estimate	Std. Error	t value	Pr(>|t|)
omega	0.00927054	0.002577062	3.597328	0.0003215026
alpha1	0.09976745	0.010615749	9.398060	0.0000000000
beta1	0.89873275	0.010206685	88.053343	0.0000000000
shape	6.65154269	0.646320576	10.291399	0.0000000000

Now let's fit an apARCH model with fat tails. The paramter list will now include `gamma1` and `delta`, which are part of the apARCH model specification:

```r
# tapARCH model
spec3 <- ugarchspec(
    variance.model = list(model = "apARCH"),
    mean.model = list(armaOrder = c(0,0), include.mean = FALSE),
    distribution.model = "std"
)

tapARCH <- ugarchfit(spec = spec3, data = ret)

tapARCH@fit$matcoef
```

	Estimate	Std. Error	t value	Pr(>|t|)
omega	0.02231419	0.0015974472	13.96866	0
alpha1	0.08930565	0.0043439698	20.55853	0
beta1	0.90967493	0.0052901005	171.95797	0
gamma1	0.99999971	0.0003113951	3211.35291	0
delta	1.04334904	0.0924854533	11.28122	0
shape	7.72471637	0.8248865782	9.36458	0

Issues

Sample size

It is important to consider that the more parameter a model has, the more likely it is to run into estimation problems, and the more data we need. For example, as a rule of thumb, for a GARCH(1,1) we need at least 500 observations, while for a Student-t GARCH that minimum number is around 3,000 observations. In general, the more parameters are estimated, the more data is needed to be confident in the estimation.

Optimization failure

`ugarchfit()` essentially performs Maximum Likelihood to fit the specified model to our data. This optimization problem required a `solver`, which is the numerical method used to maximize the likelihood function. In some case, the default solver in `ugarchfit()` can fail to converge. We recommend using the option `solver = "hybrid"` when fitting the model, since this will try different optimization methods in case the default one does not converge.

Multivariate models with *rmgarch*

The package *rmgarch* allows to easily estimate models with multiple assets in the same fashion as *rugarch*. The procedure is analogous, we first need to specify the model we want to use, and then fit it to the data. However, specifying the model has extra steps. To explain this, we will focus on the estimation of a DCC model.

In a DCC model, we assume that each individual asset follows some type of univariate model, usually a GARCH. And then we model the correlation between the assets using an ARMA-like process.

The process for fitting a DCC model using *rmgarch* is then:

1. Specify the univariate volatility model each asset follows using `ugarchspec()`
2. Use the `multispec()` function to create a multivariate specification. This is essentially a list of univariate specifications. If we are going to use the same for every asset, we can use `replicate()`
3. Then we need to create a `dccspec()` object, which takes in the list of univariate specifications for every asset, and the additional DCC joint specifications, like `dccOrder` and `distribution`
4. Fit the specification to the data

We will use the returns for JPM and Citigroup from the file `Y.RData` to go through this process:

```r
library(rmgarch)
load("RET.RData")
y <- RET[c("JPM", "C")]
```

We will assume a simple GARCH(1,1) with mean zero for each stock. We will create a single univariate specification, and then replicate it into `multispec()`:

```r
# Create the univariate specification
uni_spec <- ugarchspec(
    variance.model = list(garchOrder = c(1,1)),
    mean.model = list(armaOrder = c(0,0), include.mean = FALSE)
)

# Replicate it into a multispec element
mspec <- multispec(replicate(2, uni_spec))
```

Let's take a look inside the `mspec` object. We will see there are two univariate specifications, one per asset, and they are equal:

```r
mspec
```

```
*-----------------------------*
*     GARCH Multi-Spec        *
*-----------------------------*
Multiple Specifications : 2
Multi-Spec Type                  : equal
```

You can check the specifications with `mspec@spec`

Now we proceed to create the specification for the DCC model using `dccspec()`:

```r
spec <- dccspec(
                # Univariate specifications - Needs to be multispec
                uspec = mspec,

                # DCC specification. We will assume an ARMA(1,1)-like process
                dccOrder = c(1,1),

                # Distribution, here multivariate normal
                distribution = "mvnorm"
)
```

We can call `spec` to see what is inside:

```r
spec
```

```
*------------------------------*
*       DCC GARCH Spec         *
*------------------------------*
Model          :  DCC(1,1)
Estimation     :  2-step
Distribution   :  mvnorm
No. Parameters :  9
No. Series     :  2
```

We can again see more details in `spec@model` and `spec@umodel`.

Now we can proceed to fit the specification to the data:

```r
res <- dccfit(spec, data = y)
res
```

```
*---------------------------------*
*          DCC GARCH Fit          *
*---------------------------------*

Distribution         :  mvnorm
Model                :  DCC(1,1)
No. Parameters       :  9
[VAR GARCH DCC UncQ] : [0+6+2+1]
No. Series           :  2
No. Obs.             :  1258
Log-Likelihood       :  8156.696
Av.Log-Likelihood    :  6.48

Optimal Parameters
-----------------------------------
               Estimate  Std. Error  t value Pr(>|t|)
[JPM].omega    0.000029    0.000016   1.8689 0.061633
[JPM].alpha1   0.141068    0.062791   2.2466 0.024663
[JPM].beta1    0.691315    0.135507   5.1017 0.000000
[C].omega      0.000021    0.000013   1.6703 0.094864
[C].alpha1     0.118458    0.056954   2.0799 0.037535
[C].beta1      0.795242    0.098281   8.0915 0.000000
[Joint]dcca1   0.100491    0.028630   3.5100 0.000448
[Joint]dccb1   0.685564    0.090675   7.5606 0.000000

Information Criteria
---------------------

Akaike       -12.953
Bayes        -12.917
Shibata      -12.954
Hannan-Quinn -12.940


Elapsed time : 0.933454
```

We can check what slots are inside:

```r
names(res@model)
names(res@mfit)
```

'modelinc' 'modeldesc' 'modeldata' 'varmodel' 'pars' 'start.pars' 'fixed.pars' 'maxgarchOrder' 'maxdccOrder' 'pos.matrix' 'pidx' 'DCC' 'mu' 'residuals' 'sigma' 'mpars' 'ipars' 'midx' 'eidx' 'umodel'

'coef' 'matcoef' 'garchnames' 'dccnames' 'cvar' 'scores' 'R' 'H' 'Q' 'stdresid' 'llh' 'log.likelihoods' 'timer' 'convergence' 'Nbar' 'Qbar' 'plik'

```r
# Coefficient matrix
res@mfit$matcoef
```

	Estimate	Std. Error	t value	Pr(>|t|)
[JPM].omega	2.930068e-05	1.567784e-05	1.868924	6.163342e-02
[JPM].alpha1	1.410681e-01	6.279070e-02	2.246641	2.466300e-02
[JPM].beta1	6.913155e-01	1.355069e-01	5.101699	3.366172e-07
[C].omega	2.118301e-05	1.268232e-05	1.670279	9.486425e-02
[C].alpha1	1.184580e-01	5.695379e-02	2.079896	3.753505e-02
[C].beta1	7.952415e-01	9.828105e-02	8.091504	6.661338e-16
[Joint]dcca1	1.004908e-01	2.862958e-02	3.510034	4.480492e-04
[Joint]dccb1	6.855639e-01	9.067528e-02	7.560648	4.019007e-14

```r
# Log likelihood
res@mfit$llh
```

8156.69553672898

The matrix H inside `res@mfit` includes the covariances. It is 3-dimensional, since it includes the 2x2 covariance matrix for each of the T time periods:

```r
H <- res@mfit$H
dim(H)
```

2 2 1258

```r
# First period's covariances
H[,,1]
```

0.0001754697	0.0001841418
0.0001841418	0.0002449290

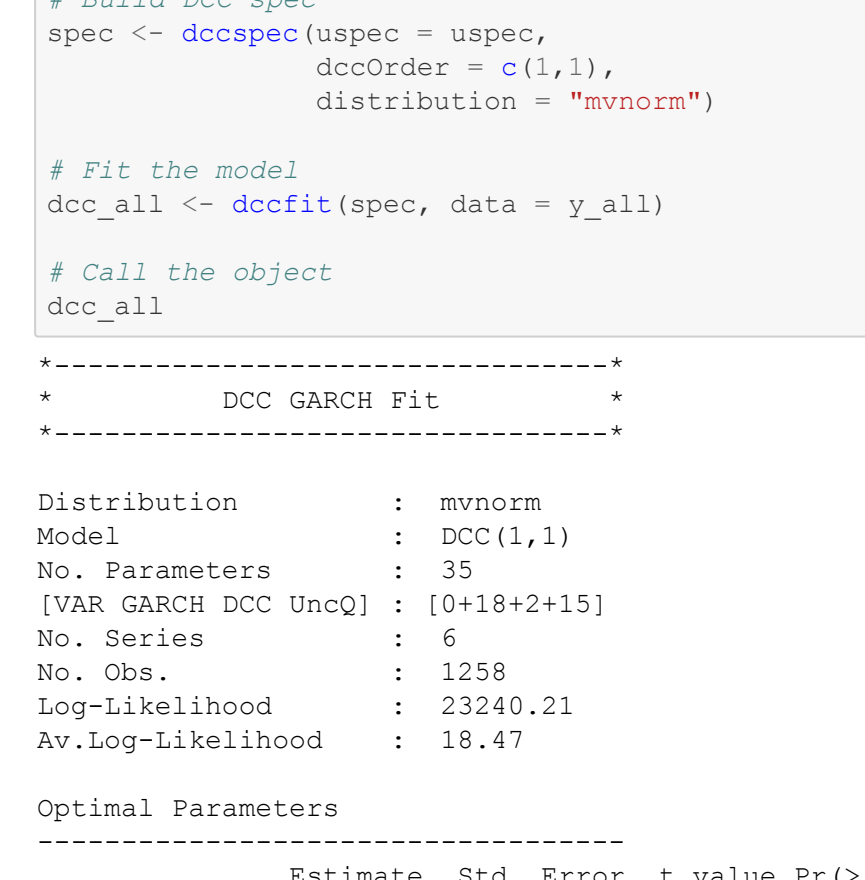
We can extract the conditional correlation in two ways. One is computing it from H:

```r
# Initializing the vector
rhoDCC <- vector(length = dim(y)[1])

# Populate with the correlations
rhoDCC <- H[1,2,] / sqrt(H[1,1,]*H[2,2,])
```

Or we can directly extract it from `res@mfit`:

```r
# Initializing the vector
rhoDCC2 <- vector(length = dim(y)[1])

for (i in 1:dim(y)[1]) {
    rhoDCC2[i] <- rcor(res)[1,2,i]
}
```

```r
plot(rhoDCC2, type = "l")
```



Several assets

We can extend this procedure to several assets just by replicating the individual GARCH specification more times. Here is an example for 6 assets:

```r
y_all <- subset(RET, select = -c(date))
y_all <- as.matrix(y_all)

# Univariate spec
xspec <- ugarchspec(mean.model = list(armaOrder = c(0,0), include.mean = FALSE))

# Replicate for each asset
uspec <- multispec(replicate(dim(y_all)[2], xspec))

# Build DCC spec
spec <- dccspec(uspec = uspec,
                dccOrder = c(1,1),
                distribution = "mvnorm")

# Fit the model
dcc_all <- dccfit(spec, data = y_all)

# Call the object
dcc_all
```

```
*---------------------------------*
*          DCC GARCH Fit          *
*---------------------------------*

Distribution         :  mvnorm
Model                :  DCC(1,1)
No. Parameters       :  35
[VAR GARCH DCC UncQ] : [0+18+2+15]
No. Series           :  6
No. Obs.             :  1258
Log-Likelihood       :  23240.21
Av.Log-Likelihood    :  18.47

Optimal Parameters
-----------------------------------
                Estimate  Std. Error  t value Pr(>|t|)
[MSFT].omega    0.000033    0.000013  2.49435 0.012619
[MSFT].alpha1   0.223677    0.083369  2.68297 0.007297
[MSFT].beta1    0.654439    0.084555  7.73983 0.000000
[XOM].omega     0.000010    0.000001  7.93856 0.000000
[XOM].alpha1    0.180048    0.025082  7.17846 0.000000
[XOM].beta1     0.756208    0.032562 23.22399 0.000000
[GE].omega      0.000006    0.000011  0.56401 0.572748
[GE].alpha1     0.145713    0.056479  2.57995 0.009881
[GE].beta1      0.853287    0.062404 13.67368 0.000000
[JPM].omega     0.000029    0.000015  1.92622 0.054076
[JPM].alpha1    0.141068    0.059964  2.35256 0.018645
[JPM].beta1     0.691315    0.130500  5.29743 0.000000
[INTC].omega    0.000002    0.000007  0.29527 0.767786
[INTC].alpha1   0.021260    0.025515  0.83322 0.404720
[INTC].beta1    0.971191    0.032324 30.04537 0.000000
[C].omega       0.000021    0.000012  1.69795 0.089518
[C].alpha1      0.118458    0.055894  2.11932 0.034063
[C].beta1       0.795242    0.096387  8.25052 0.000000
[Joint]dcca1    0.032177    0.008970  3.58724 0.000334
[Joint]dccb1    0.826159    0.061671 13.39633 0.000000

Information Criteria
---------------------

Akaike       -36.892
Bayes        -36.749
Shibata      -36.894
Hannan-Quinn -36.839


Elapsed time : 2.448151
```

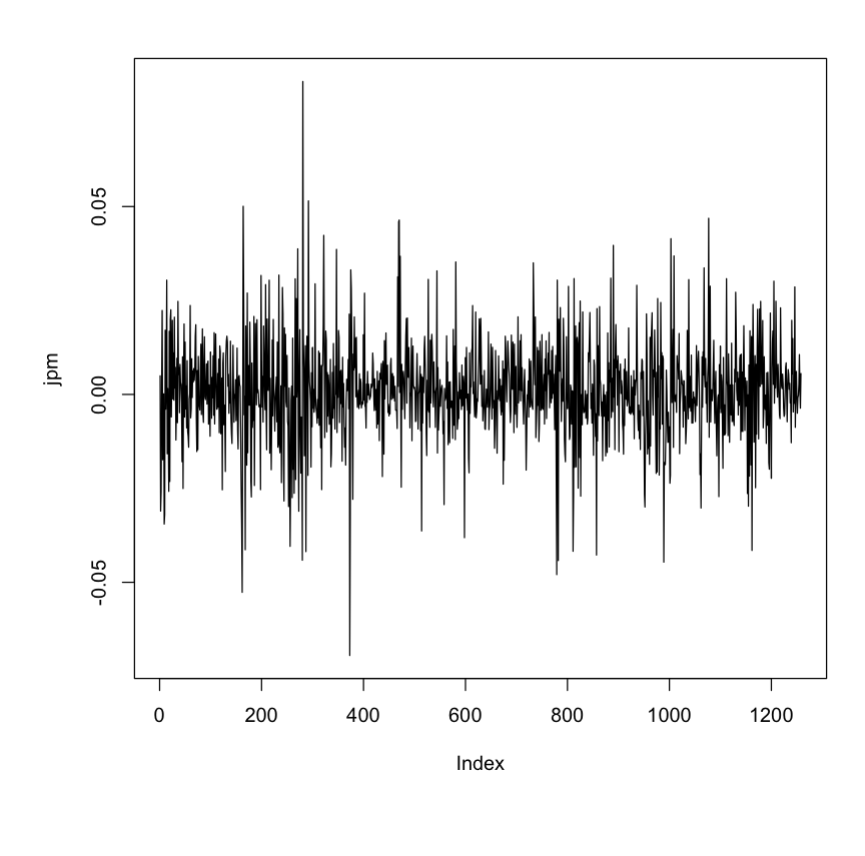
8 | VaR probabilities

In this section we will discuss an important topic in VaR estimation using Historical Simulation. Let's say we are interested in performing a HS VaR(5%) for JPM. Let's first load the data:

```r
load("RET.RData")
jpm <- RET$JPM
head(jpm)
```

0.004954 -0.031045 -0.025929 0.001526 0.022346 -0.017387

```r
# Plot the returns
plot(jpm, type = "l")
```

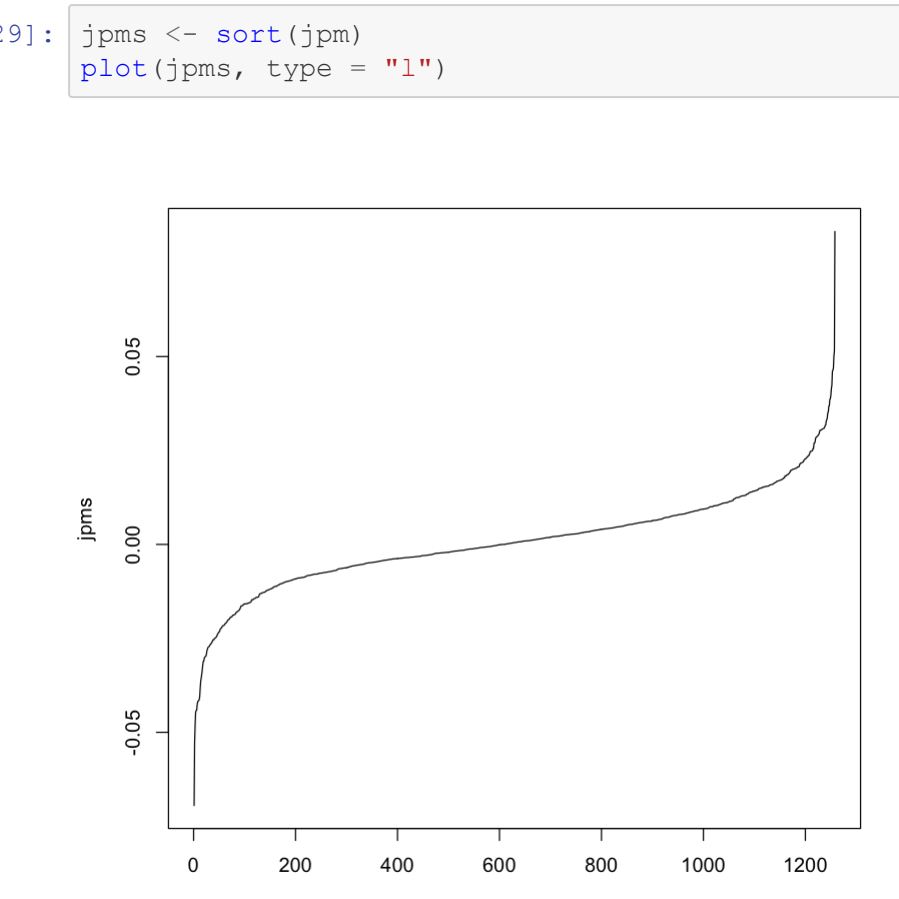


Now let's specify the VaR probability:

In [128]:
```
p <- 0.05
```

Let's sort the values using `sort()`:

In [129]:
```
jpms <- sort(jpm)
plot(jpms, type = "l")
```



Historical Simulation is a method that assumes that history repeats itself. It uses the empirical distribution of the returns to find the p-th quantile. To do this, we figure out which observation of the sorted dataset p-th one. Practically, this involves multiplying the length of the dataset by the VaR probability.

Let's try this:

In [130]:
```
n <- length(jpms)

n*p
```

62.9

The value we obtain is not an integer. So what do we do? Do we round this number to the closest integer, always round down, always round up, or get a weighted average of the previous and posterior observations?

There is no consensus on this, but we will provide two options depending on your assumption on the empirical distribution of the returns.

Discrete returns - Always round up

You might want to assume that the empirical distribution of the returns is discrete. Meaning that you don't want to average observations. In other words, you want to use one of the past returns as the VaR estimation. In this case, you always need to **round up**. Imagine a case where `n*p = 80.1`. Intuitevely, it seems obvious that you should round down to 80 and use the 80th observation. However, this would not be technically correct, because the quantile of the 80th observation is actually less than the required VaR probability p. So you would not be meeting the required probability. In many cases, VaR is computed as part of the financial regulation system, so it is quite important to meet the required probability. Using the 80th observation in this case might have given us the 4.9% VaR, whereas using the 81st observation might have given us the 5.6% VaR, but the latter does meet the required probability, while the first does not.

Continuous returns - Weighted average

If you want to treat the empirical distribution of returns as continuous, you can computed the weighted average between the previous and posterior observations. For example, if `n*p = 80.1`, you would use the `0.9*(80th observation) + 0.1*(81st observation)`.

For our purposes, we believe it is more intuitive to strictly use a past observation as the VaR estimation, so we will always round up. This can be easily done using the `ceiling()` function. Completing the HS estimation in our case:

In [131]:
```
ceiling(n*p)
```

63

In [132]:
```
# VaR(5%)

jpms[ceiling(n*p)]
```

-0.020952

9 | Simulated VaR with simple and continuous returns

When working with simulations, we can choose if we want to use simple or continuous returns to simulate future prices.

Simple returns

If we want to work with simple returns, we calculate one-day future prices in the following way:

1. Simulate one-day return: $R_{t+1} \sim \mathcal{N}(0, \sigma^2)$
2. Calculate one-day future price: $P_{t+1} = P_t \times (1 + R_{t+1})$

Continuous returns

If we want to work with continuous returns, we calculate one-day future prices in the following way:

1. Simulate one-day return: $y_{t+1} \sim \mathcal{N}(0, \sigma^2)$
2. Calculate one-day future price: $P_{t+1} = P_t e^{r(1/365)} \times e^{y_{t+1}} \times e^{-0.5\sigma^2}$

Comparison for VaR 5%

In [133]:
```
# Simple returns
p <- 0.05
S <- 1e5
P <- 100
sigma <- 0.01
ret <- rnorm(S, 0, sigma^2)
Psim_simple <- P*(1+ret)

# VaR 5%
Ps_simple <- sort(Psim_simple - P)
Ps_simple[p*S]
```

-0.0163162893227877

In [134]:
```
# Compound returns
r <- 0.03   # Assuming risk free rate
Psim_comp <- P*exp(r*(1/365))*exp(ret)*exp(-0.5*sigma^2)
# VaR 5%
Ps_comp <- sort(Psim_comp - P)
Ps_comp[p*S]
```

-0.0130962536064345

10 | Statistical Testing

Jarque-Bera

The Jarque-Bera (JB) test uses the third and fourth central moments of the sample data to check if it has a skewness and kurtosis matching a normal distribution. If the sample data comes from a normal distribution, the JB statistic has an asymptotic chi-square distribution with two degrees of freedom. So:

$$H_0: \text{The data is normally distributed}$$
$$H_a: \text{The data is NOT normally distributed}$$

We will generally use the JB test to check if our data follows a normal distribution. The function `jarque.bera.test()` from the `tseries` package will evaluate the data to compute the JB statistic and p-value. A small p-value will mean there is enough evidence to reject the null hypothesis that the third and fourth central moments of the sample data resemble the ones of a normal distribution.

Let's do a test on the normality of the S&P500 returns:

In [135]:
```
library(tseries)
sp <- read.csv("sp500.csv")
ret  <- diff(log(sp$price))
jarque.bera.test(ret)
```

```
Registered S3 method overwritten by 'quantmod':
  method            from
  as.zoo.data.frame zoo
```

```
        Jarque Bera Test

data:  ret
X-squared = 14521, df = 2, p-value < 2.2e-16
```

The p-value presented is the inverse of the JB statistic under the CDF of the asymptotic distribution. The p-value of this test is virtually zero, which means that we have enough evidence to reject the null hypothesis that our data has been drawn from a Normal distribution.

Ljung-Box

The Ljung-Box (LB) test checks if the presented data exhibit serial correlation. It has as null-hypothesis that the data are independently distributed. The test statistic includes the sample size, and a sum of sample autocorrelations squared up to a specified lag, scaled by the sample size minus the lag. Under the null hypothesis, the statistic asymptotically follows a chi-square distribution with degrees of freedom equal to the number of lags being tested.

$$H_0: \text{The data exhibits serial correlation}$$
$$H_a: \text{The data does NOT exhibit serial correlation}$$

This type of test is commonly used for the residuals of ARMA models (Note: not used for the actuals values of the ARMA series). For our purposes, we normally use the LB test on returns and returns squared. Remember that it makes sense to do GARCH modelling when volatility exhibits autocorrelation, so we expect to see a rejection of the LB null hypothesis when applying the test to returns squared, even if we cannot reject the null when applied to returns.

We can implement the test using the `Box.test()` function and specifying the `type = "Ljung-Box"`:

In [136]:
```
Box.test(ret, type = "Ljung-Box")
Box.test(ret^2, type = "Ljung-Box")
```

```
        Box-Ljung test

data:  ret
X-squared = 26.212, df = 1, p-value = 3.059e-07

        Box-Ljung test

data:  ret^2
X-squared = 212.66, df = 1, p-value < 2.2e-16
```

Likelihood Ratio test

The Likelihood-Ratio (LR) test compares two models based on the ratio of their likelihoods, where one of the models was fitted over a restricted parameter space compared to the other one, so we can say it is *nested*. When we are working with log-likelihoods, we can naturally express the test statistic as a difference instead of a ratio.

Intuitively, you can think of the LR test as a check if a restriction imposed on a model makes a significant difference or not. The null hypothesis is that there is essentially no difference between the models. The LR statistic is asymptotically distributed as a chi-square with degrees of freedom qual to the difference in dimensionality in the parameter spaces that each model used in their maximum likelihood estimation.

$$H_0: \text{The two models are the same (the parameters excluded are jointly equal to zero)}$$
$$H_a: \text{The two models are not the same}$$

Let's give an application of this to make this clearer. Assume you want to build a model for the returns of the S&P500, and you are considering an ARCH(1) and a GARCH(1,1) model. Essentially, an ARCH(1) is a specific case of the GARCH family, in particular, it is a GARCH(1,0). Consider the maximum likelihood estimation of a GARCH(1,1) process, we are maximizing the likelihood in the parameters ω, α, and β. We can see the ARCH maximization process as exactly the same, except that we are narrowing down the parameter space, since we are forcing $\beta = 0$. Hence, there is **one less dimension** in the ARCH parameter space MLE than in the GARCH MLE.

We can then perform a LR test to see if there is a gain in using a GARCH(1,1) instead of an ARCH(1) model.

Let's fit both models and build the LR test ourselves:

In [137]:
```
library(rugarch)

# ARCH
spec_arch <- ugarchspec(
    variance.model = list(garchOrder = c(1,0)),
    mean.model = list(armaOrder = c(0,0), include.mean = FALSE)
)

ARCH <- ugarchfit(spec = spec_arch, data = ret, solver = "hybrid")

# GARCH
spec_garch <- ugarchspec(
    variance.model = list(garchOrder = c(1,1)),
    mean.model = list(armaOrder = c(0,0), include.mean = FALSE)
)

GARCH <- ugarchfit(spec = spec_garch, data = ret, solver = "hybrid")

# Perform the LR test
LR_statistic <- 2*(likelihood(GARCH)-likelihood(ARCH))
p_value  <- 1 - pchisq(LR_statistic, df = 1)

cat(" Likelihood of ARCH: ", round(likelihood(ARCH),2), "\n",
    "Likelihood of GARCH: ", round(likelihood(GARCH),2), "\n",
   "2 * (Lu - Lr): ", round(LR_statistic,2), "\n",
   "p-value:", p_value)
```

```
 Likelihood of ARCH:  14588.73
 Likelihood of GARCH:  15449.95
 2 * (Lu - Lr):  1722.44
 p-value: 0
```

We find a p-value of 0, meaning that we have enough evidence to reject the null hypothesis that the two models are the same. The GARCH model has a significantly larger likelihood than the ARCH model.

11 | The Apply family of functions

The *apply* family of functions in R allows you to repetitively perform an action on multiple chunks of data. It is good practice to get familiar with these functions since they are a direct way to perform operations that otherwise would require you to set up a loop. They are usually much faster too. The reason it is a *family of functions* instead of a single function, is that each one is specific to some data type and will provide the output in a different format.

Before getting started, let's take a look at a simple example, imagine you have the following dataset:

In [138]:
```
load("RET.RData")
y <- subset(RET, select = -c(date))
head(y)
```

MSFT	XOM	GE	JPM	INTC	C
0.006674	0.004110	-0.008310	0.004954	0.001929	0.002772
-0.009303	-0.027362	-0.018356	-0.031045	-0.011276	-0.031515
-0.014571	-0.005316	-0.021545	-0.025929	-0.018637	-0.035205
0.012705	0.010133	0.000415	0.001526	0.020975	0.009270
0.029418	0.016645	0.012043	0.022346	0.018601	0.015048
-0.008405	-0.001410	-0.013952	-0.017387	0.001908	-0.022333

And you want to get the variance for each row. The first thing that comes to mind is either writing a line of code for each row:

In [139]:
```
print(var(y$MSFT))
print(var(y$XOM))
print(var(y$GE))
print(var(y$JPM))
print(var(y$INTC))
print(var(y$C))
```

```
[1] 0.0002155043
[1] 0.0001436519
[1] 0.0003526265
[1] 0.0001749218
[1] 0.0002597558
[1] 0.0002448805
```

Or write a loop:

In [140]:
```
for (stock in names(y)) {
    print(var(y[c(stock)]))
}
```

```
             MSFT
MSFT 0.0002155043
             XOM
XOM 0.0001436519
             GE
GE 0.0003526265
             JPM
JPM 0.0001749218
             INTC
INTC 0.0002597558
             C
C 0.0002448805
```

The first approach is inefficient, especially with a large number of columns. The second approach is useful but a loop can seem unnecessary. We can use the apply family of function for this:

In [141]:
```
apply(y, 2, var)
```

MSFT	0.000215504300024893
XOM	0.000143651860649984
GE	0.00035262645651305
JPM	0.000174921788784438
INTC	0.000259755847669449
C	0.000244880468210118

The apply function

The basic apply function has the syntax: `apply(X, MARGIN, FUN)` where:

- X is an array, matrix, or dataframe (the data you will be performing the function on
- MARGIN specifies if you are going to perfrom the function accross rows (1) or columns (2)
- FUN is the function you want to use, which can include options

For example, let's create a matrix:

In [142]:
```
a <- matrix(rnorm(20), nrow = 5, ncol = 4)
a
```

0.9720144	1.0263764	-0.24445109	-0.34984960
0.8848046	-0.5175984	-0.87864295	-0.21970682
1.7020800	1.2469390	-0.54733383	1.22877162
-1.5881660	-0.1629312	-0.24592251	0.09574781
0.5411785	2.0624779	-0.04103685	1.04744810

Let's find the row and column sums using MARGIN 1 and 2 respectively:

In [143]:
```
# Row sums
apply(a, 1, sum)
```

1.40409013122431   -0.731143612430116   3.6304568031668   -1.90127188696659   3.6100676568428

In [144]:
```
# Column sums
apply(a, 2, sum)
```

2.51191152274534   3.6552636957194   -1.95738723320548   1.80241110657793

Let's find the largest element of every row:

In [145]:
```
# Max of every row
apply(a, 1, max)
```

1.02637644719398   0.884804587966169   1.70208003179984   0.0957478125732051   2.062477906833

However, `apply()` will not work on vectors:

In [146]:
```
vec <- 1:20
apply(vec, 1, sum)
```

```
Error in apply(vec, 1, sum): dim(X) must have a positive length
Traceback:

1. apply(vec, 1, sum)
2. stop("dim(X) must have a positive length")
```

lapply and sapply

These are variations of `apply()` that work with other data types.

lapply( )

This variation can be used for other objects like dataframes, lists or vectors. The output return is a **list**, which has the same number of elements as the object passed to it.

For example, let's create a list from three matrices:

In [147]:
```
A <- matrix(1:9, nrow = 3, ncol = 3)
B <- matrix(1:4, nrow = 2, ncol =2)
C <- matrix(rep(1, 100), nrow = 10, ncol = 10)

my_list <- list(A,B,C)

# We can extract the 2nd column from every element using the selector "["
lapply(my_list, "[", , 2)
```

1. 4   5   6
2. 3   4
3. 1   1   1   1   1   1   1   1   1   1

In [148]:
```
# Or the first row of all elements
lapply(my_list, "[", 1, )
```

1. 1   4   7
2. 1   3
3. 1   1   1   1   1   1   1   1   1   1

Note that the outputs are **lists**. To subset a list you need to use double brackets:

In [149]:
```
a <- lapply(my_list, "[", 1, )
a[[1]]
```

1   4   7

We can use `lapply()` on a vector. Let's get the square root of every element from 1 to 10:

In [150]:
```
vec = 1:10
a <- lapply(vec, sqrt)
a
class(a)
```

1. 1
2. 1.4142135623731
3. 1.73205080756888
4. 2
5. 2.23606797749979
6. 2.44948974278318
7. 2.64575131106459
8. 2.82842712474619
9. 3
10. 3.16227766016838

'list'

sapply

The `sapply()` functions works very similarly to `lapply()`, with the difference that it simplies the output. While `lapply()` provides a list as output, `sapply()` usually provides vectors, which are easier to work with:

In [151]:
```
vec = 1:10
b <- sapply(vec, sqrt)
b
class(b)
```

1   1.4142135623731   1.73205080756888   2   2.23606797749979   2.44948974278318   2.64575131106459   2.82842712474619
3   3.16227766016838

'numeric'

12 | Introduction to dplyr

The *dplyr* package is a fast, consistent tool for working with data frame like objects. It is the most common tool for data manipulation, and can make your life easier. Here we will provide a very brief introduction to it. It is the next iteration of the *plyr* package (remember to load *plyr* before *dplyr*!).

A complete *cheat sheet* on how to use *dplyr* can be found here: https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

Some of its most useful functions are:

filter

filter keeps rows meeting a condition. The syntax is `filter(data, ..., preserve)`, where:

- data = The data frame containing the rows you want to filter
- ... = The conditions to be evaluated. To use multiple conditions, use the logical operators & (AND), | (OR)
- preserve = F to recalculate the grouping structure based on the resulting data

select

select keeps only the specified variables. Syntax is `select(data,...)`, where:

- data = The data frame containing the variables you want to select
- ... = one or more unquoted expressions separated by commas; you can treat variable names like they are positions, so you can use expressions like x:y to select range of variables; positive values select variables, and negative values drop variables

mutate

The *mutate* function add new variables and preserves existing ones. Syntax is `mutate(date, newvar = ...)` where:

- data = The data object you want to add the variable to
- ... = The function you want to use to create the new variable

summarize

summarize creates one or more scalar variables summarizing the variables of an existing data obeject. Syntax is `summarize(data, newvar = ...)` where:

- data = The data object containing the variable you want to summarize
- ... = The function you want to use to create the new variable

group_by

group_by converts a data object into a grouped data object where operations are perform 'by group'. *ungroup( )* removes grouping. Syntax is `group_by(data, ..., add, drop)` where:

- data = The data object containing the variable you want to group by
- ... = The variables you want to group by
- add = F to override existing group (default)
- drop = T to drop empty groups

join

Used to join data objects together. Syntax is `*_join(x, y, by)` where:

- x, y = The data objects to join
- by = A character vector of variables to join by

There are different types of join:

- _inner*join* returns all rows from x where there are matching values in y, and all columns from x and y
- _left*join* returns all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values
- _right*join* similar to _left*join*
- _outer*join* returns all rows and all columns from both x and y

Pipes

dplyr allows to perform several operations at the same time by connecting different functions with the symbol `%>%`.

It is particularly useful if you want to execute a function by group.

Tidyverse

Dplyr is one of the packages included in *tidyverse*, which is a collection of R packages designed for data science.

Examples

We will use mpg, one of the existing datasets in the *tidyverse* to demonstrate how *dplyr* works:

In [152]:
```
# Load in this order
library(plyr)
library(dplyr)

# Tidyverse
library(tidyverse)
```

```
Attaching package: 'plyr'


The following object is masked from 'package:lubridate':

    here



Attaching package: 'dplyr'


The following objects are masked from 'package:plyr':

    arrange, count, desc, failwith, id, mutate, rename, summarise,
    summarize


The following objects are masked from 'package:rmgarch':

    first, last


The following objects are masked from 'package:lubridate':

    intersect, setdiff, union


The following objects are masked from 'package:stats':

    filter, lag


The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union


── Attaching packages ─────────────────────────────────── tidyverse 1.2.1 ──

✔ ggplot2 3.2.1     ✔ readr   1.3.1
✔ tibble  2.1.3     ✔ purrr   0.3.3
✔ tidyr   1.0.0     ✔ stringr 1.4.0
✔ ggplot2 3.2.1     ✔ forcats 0.4.0

── Conflicts ────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::arrange()          masks plyr::arrange()
✖ lubridate::as.difftime() masks base::as.difftime()
✖ purrr::compact()          masks plyr::compact()
✖ dplyr::count()            masks plyr::count()
✖ lubridate::date()         masks base::date()
✖ dplyr::failwith()         masks plyr::failwith()
✖ dplyr::filter()           masks stats::filter()
✖ dplyr::first()            masks rmgarch::first()
✖ plyr::here()              masks lubridate::here()
✖ dplyr::id()               masks plyr::id()
✖ lubridate::intersect()    masks base::intersect()
✖ dplyr::lag()              masks stats::lag()
✖ dplyr::last()             masks rmgarch::last()
✖ dplyr::mutate()           masks plyr::mutate()
✖ purrr::reduce()           masks rugarch::reduce()
✖ dplyr::rename()           masks plyr::rename()
✖ lubridate::setdiff()      masks base::setdiff()
✖ dplyr::summarise()        masks plyr::summarise()
✖ dplyr::summarize()        masks plyr::summarize()
✖ lubridate::union()        masks base::union()
```

In [153]:
```
# First look at the dataset
data <- mpg
head(mpg)
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
audi	a4	1.8	1999	4	auto(l5)	f	18	29	p	compact
audi	a4	1.8	1999	4	manual(m5)	f	21	29	p	compact
audi	a4	2.0	2008	4	manual(m6)	f	20	31	p	compact
audi	a4	2.0	2008	4	auto(av)	f	21	30	p	compact
audi	a4	2.8	1999	6	auto(l5)	f	16	26	p	compact
audi	a4	2.8	1999	6	manual(m5)	f	18	26	p	compact

In [154]:
```
# Select only manufacturer, model and class
data_select <- select(data, manufacturer, model, class)
head(data_select)
```

manufacturer	model	class
audi	a4	compact
audi	a4	compact
audi	a4	compact
audi	a4	compact
audi	a4	compact
audi	a4	compact

In [155]:
```
# Filter only the cars that are pickups
data_filter <- filter(data, class == "pickup")
head(data_filter)
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
dodge	dakota pickup 4wd	3.7	2008	6	manual(m6)	4	15	19	r	pickup
dodge	dakota pickup 4wd	3.7	2008	6	auto(l4)	4	14	18	r	pickup
dodge	dakota pickup 4wd	3.9	1999	6	auto(l4)	4	13	17	r	pickup
dodge	dakota pickup 4wd	3.9	1999	6	manual(m5)	4	14	17	r	pickup
dodge	dakota pickup 4wd	4.7	2008	8	auto(l5)	4	14	19	r	pickup
dodge	dakota pickup 4wd	4.7	2008	8	auto(l5)	4	14	19	r	pickup

In [156]:
```
# Select only the manufacturer model and class, and then filter
# Note that with pipes you don't have to specify "data" again
data_pipes <- data %>% select(manufacturer, model, class) %>% filter(class == "pickup")
head(data_pipes)
```

manufacturer	model	class
dodge	dakota pickup 4wd	pickup
dodge	dakota pickup 4wd	pickup
dodge	dakota pickup 4wd	pickup
dodge	dakota pickup 4wd	pickup
dodge	dakota pickup 4wd	pickup
dodge	dakota pickup 4wd	pickup

In [157]:
```
# Create a new column that is a concatenation of manufacturer and model
data <- data %>% mutate(car = paste(manufacturer, model, sep = " - "))
head(data)
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class	car
audi	a4	1.8	1999	4	auto(l5)	f	18	29	p	compact	audi - a4
audi	a4	1.8	1999	4	manual(m5)	f	21	29	p	compact	audi - a4
audi	a4	2.0	2008	4	manual(m6)	f	20	31	p	compact	audi - a4
audi	a4	2.0	2008	4	auto(av)	f	21	30	p	compact	audi - a4
audi	a4	2.8	1999	6	auto(l5)	f	16	26	p	compact	audi - a4
audi	a4	2.8	1999	6	manual(m5)	f	18	26	p	compact	audi - a4

In [158]:
```
# Get the mean for the hwy variable
avg_hwy <- data %>% summarize(avg_hwy = mean(hwy))
head(avg_hwy)
```

avg_hwy
23.44017

In [159]:
```
# Get the mean of highway mileage for every manufacturer
avg_hwy2 <- data %>% group_by(manufacturer) %>% summarize(avg_hwy = mean(hwy))
head(avg_hwy2)
```

manufacturer	avg_hwy
audi	26.44444
chevrolet	21.89474
dodge	17.94595
ford	19.36000
honda	32.55556
hyundai	26.85714

In [160]:
```
# For every manufacturer, get the average highway and city mileage
avg_hwy3 <- data %>% group_by(manufacturer) %>% summarize(avg_hwy = mean(hwy),
                                                          avg_cty = mean(cty))
head(avg_hwy3)
```

manufacturer	avg_hwy	avg_cty
audi	26.44444	17.61111
chevrolet	21.89474	15.00000
dodge	17.94595	13.13514
ford	19.36000	14.00000
honda	32.55556	24.44444
hyundai	26.85714	18.64286

In [161]:
```
# Add the number of observations
avg_hwy3 <- data %>% group_by(manufacturer) %>% summarize(avg_hwy = mean(hwy),
                                                          avg_cty = mean(cty),
                                                          count = n())
head(avg_hwy3)
```

manufacturer	avg_hwy	avg_cty	count
audi	26.44444	17.61111	18
chevrolet	21.89474	15.00000	19
dodge	17.94595	13.13514	37
ford	19.36000	14.00000	25
honda	32.55556	24.44444	9
hyundai	26.85714	18.64286	14

In [162]:
```
# Get a list of the unique manufacturers
manufacturer <- unique(data$manufacturer)

# Create a dataframe with manufacturers and their countries:
countries <- data.frame(manufacturer,
                        country = c("GER","USA","USA","USA","JAP","KOR","USA","UK","USA","USA","JAP",
"USA","JAP","JAP","GER"),
                       stringsAsFactors = FALSE)

# Use a left join to add this to our data
# R will be smart enough to find the common variable if unspecified
data_all <- left_join(mpg, countries)
head(data_all)
```

```
Joining, by = "manufacturer"
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class	country
audi	a4	1.8	1999	4	auto(l5)	f	18	29	p	compact	GER
audi	a4	1.8	1999	4	manual(m5)	f	21	29	p	compact	GER
audi	a4	2.0	2008	4	manual(m6)	f	20	31	p	compact	GER
audi	a4	2.0	2008	4	auto(av)	f	21	30	p	compact	GER
audi	a4	2.8	1999	6	auto(l5)	f	16	26	p	compact	GER
audi	a4	2.8	1999	6	manual(m5)	f	18	26	p	compact	GER

13 | ggplot2

ggplot2 is a very popular graphics framework for R. It is widely used in research and industry, from data scientists at top tech companies to journalist at the New York times and university academics. Although the base plots in R can be highly customizable and it is what we mostly use for FM442, we will present a basic introduction to *ggplot2*. For more information, you can see this tutorial: http://r-statistics.co/Complete-Ggplot2-Tutorial-Part1-With-R-Code.html

ggplot2 is also part of the *tidyverse*

```
In [163]: library(ggplot2)
          library(tidyverse)
```

We will use the `mpg` dataset:

```
In [164]: mpg <- mpg
          head(mpg)
```

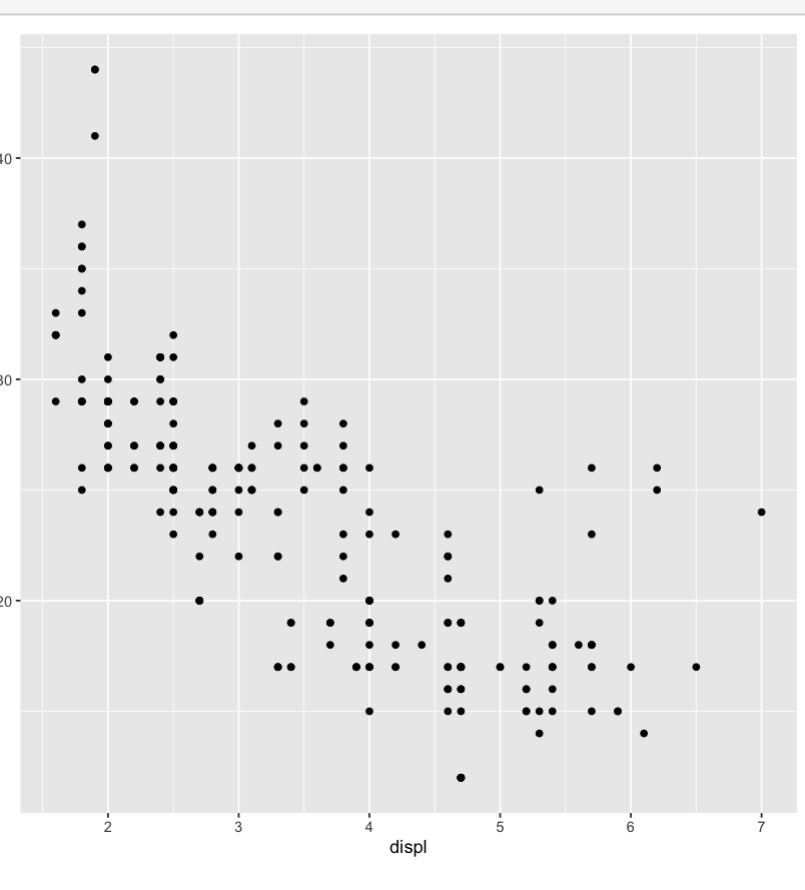
manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
audi	a4	1.8	1999	4	auto(l5)	f	18	29	p	compact
audi	a4	1.8	1999	4	manual(m5)	f	21	29	p	compact
audi	a4	2.0	2008	4	manual(m6)	f	20	31	p	compact
audi	a4	2.0	2008	4	auto(av)	f	21	30	p	compact
audi	a4	2.8	1999	6	auto(l5)	f	16	26	p	compact
audi	a4	2.8	1999	6	manual(m5)	f	18	26	p	compact

To create a *ggplot2* graph, we begin by using `ggplot(data = ..)` to specify the data that will feed the plot. On top of that we add the elements of the plot, like `geom_point(mapping = aes(x = .., y = ..)` to add points. `aes()` stands for aesthetics, and it holds the values in the x and y axis. It allows for several options that we will explore:

Scatter plot

We will build a scatter plot with engine displacement in the x-axis and highway consumption in the y-axis.

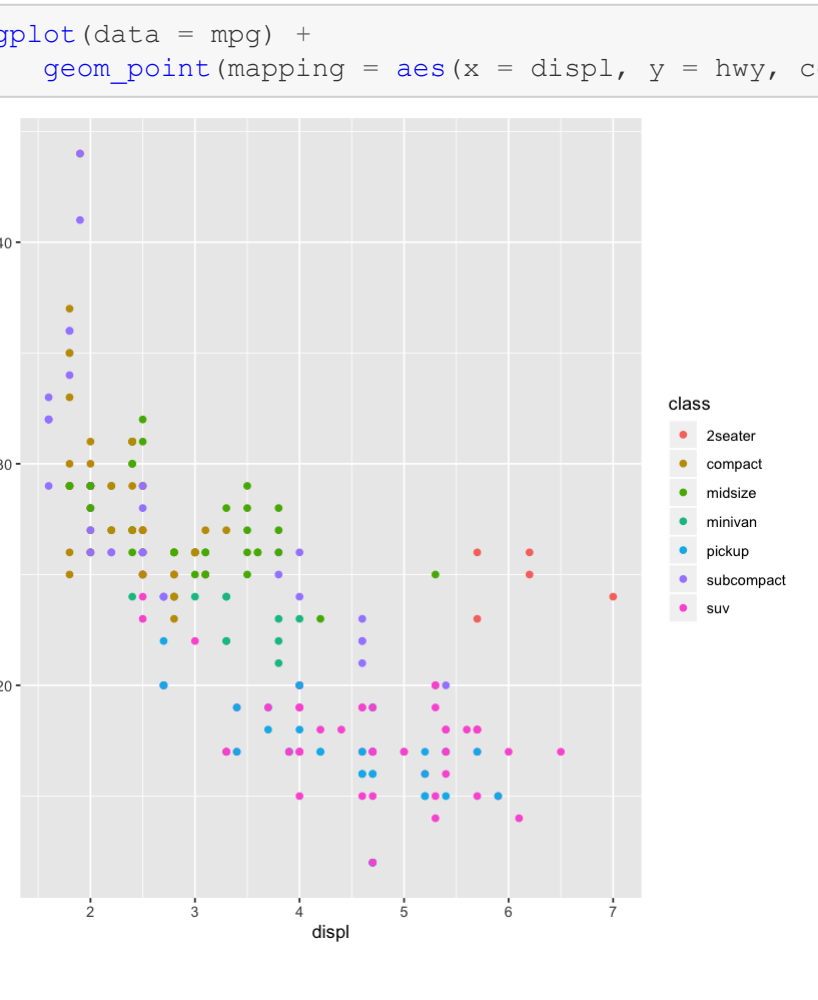
```
In [165]: # Scatter plot
          ggplot(data = mpg) +        # Specify the data
              geom_point(mapping = aes(x = displ, y = hwy))      # Add points
```



Colors

We can use `color = ...` to use a third variable to color the points plotted. This also creates a legend automatically:

```
In [166]: ggplot(data = mpg) +
              geom_point(mapping = aes(x = displ, y = hwy, color = class))
```

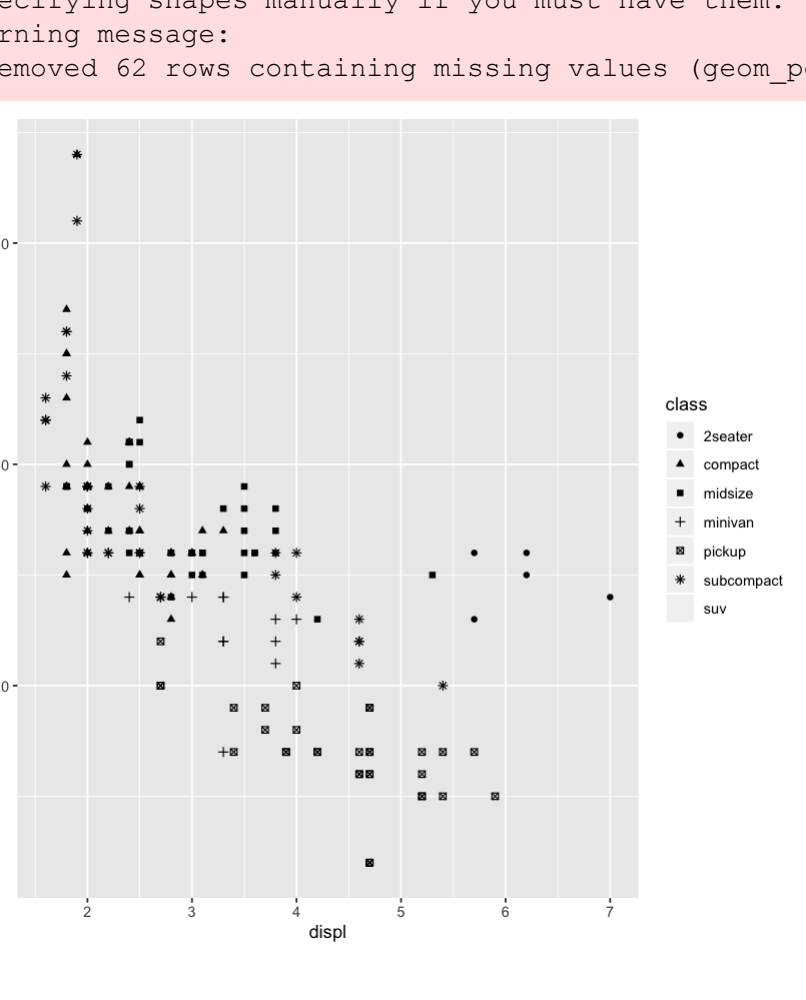


Shape

We give each point a different shape based on a third variable:

```
In [167]: ggplot(data = mpg) +
              geom_point(mapping = aes(x = displ, y = hwy, shape = class))
```

```
Warning message:
"The shape palette can deal with a maximum of 6 discrete values because
more than 6 becomes difficult to discriminate; you have 7. Consider
specifying shapes manually if you must have them."
Warning message:
"Removed 62 rows containing missing values (geom_point)."
```

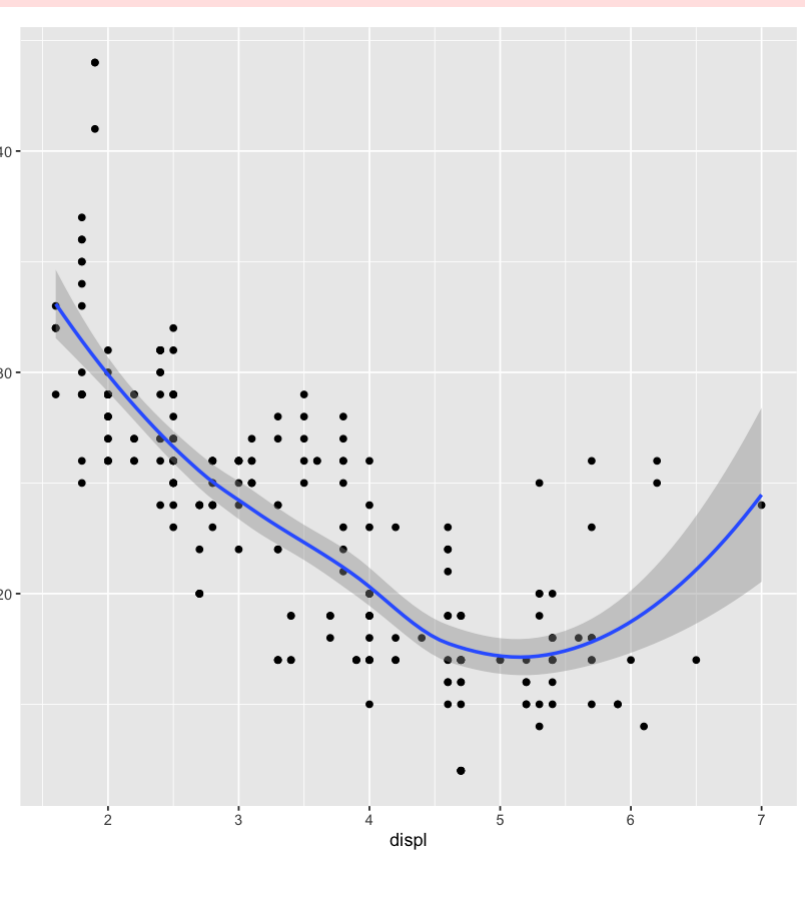


Smooth line

You can add a smooth line with confidence interval to the scatter plot using `geom_smooth()`:

```
In [168]: ggplot(data = mpg) +
              geom_point(mapping = aes(x = displ, y = hwy)) +
              geom_smooth(mapping = aes(x = displ, y = hwy))
```

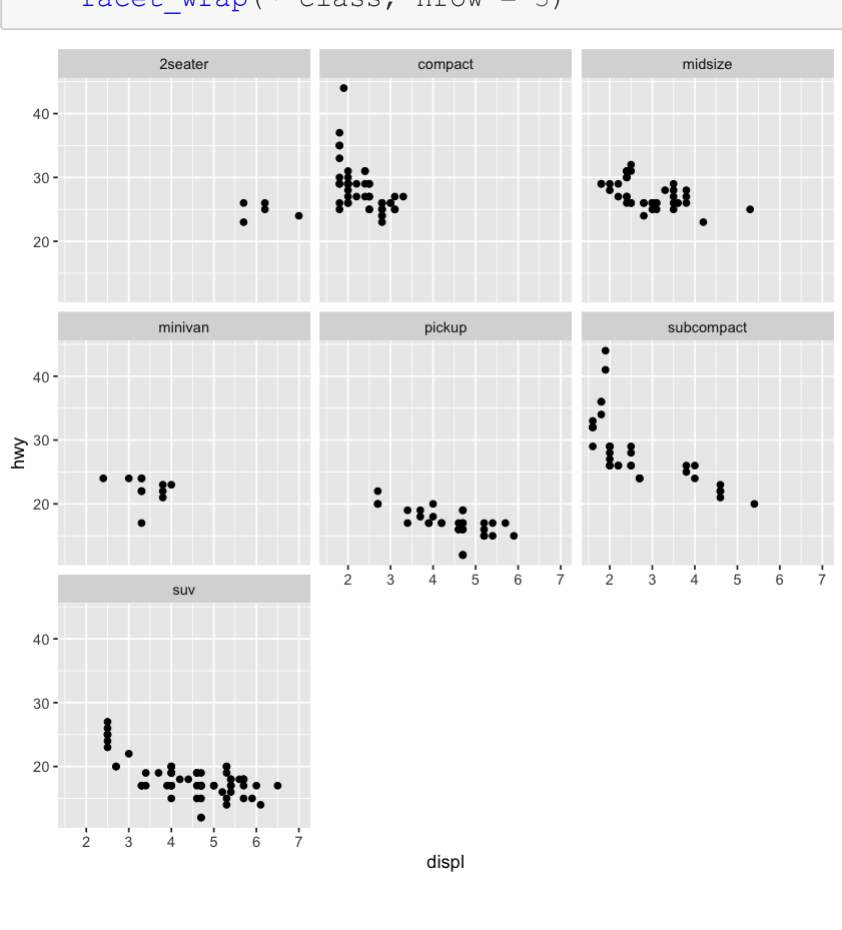
```
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Separate plots by a variable

If you want to make a different plot for each element of a variable, you can do so with `facet_wrap`:

```
In [169]: ggplot(data = mpg) +
              geom_point(mapping = aes(x = displ, y = hwy)) +
              facet_wrap(~ class, nrow = 3)
```

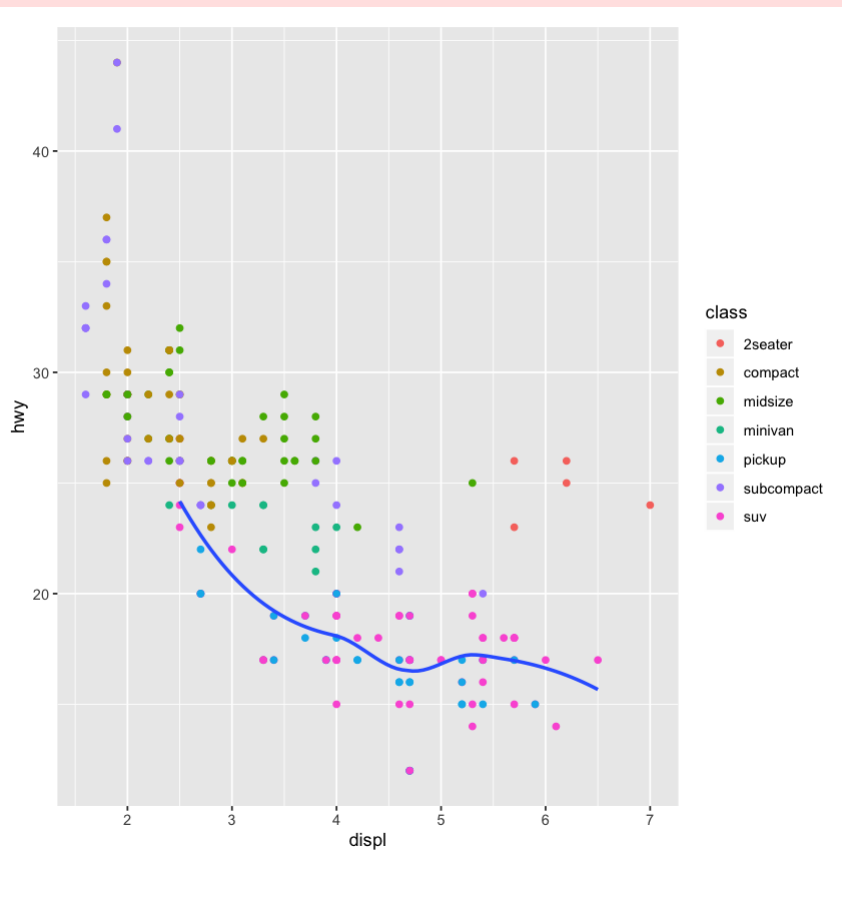


Filtering

You can also customize the plot for a specific value of a variable. For example, let's add a smooth line only for SUVs:

```
In [170]: ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
              geom_point(mapping = aes(color = class)) +
              geom_smooth(data = filter(mpg, class == "suv"), se = F)
```

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



Histograms and bar plots

Let's create a dataframe with information on sex and height to build some more plots:

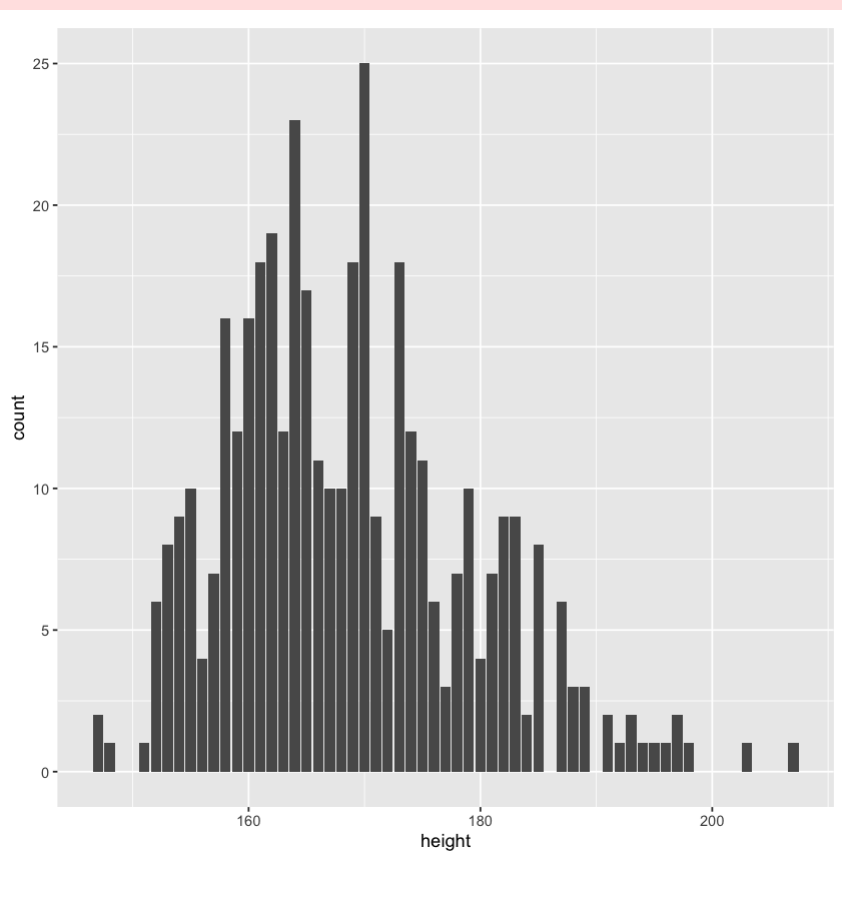
```
In [171]: height <- data.frame(
              sex = factor(rep(c("F", "M"), each=200)),
              height = round(c(rnorm(200, mean=161.5, sd=6), rnorm(200, mean=175.4, sd=9)))
          )
```

Bar Plot

We can use `geom_histogram` and select `stat = "count"` to build a simple bar plot. Notice that you can store the `ggplot` object in a variable.

```
In [172]: p1 <- ggplot(data=height, aes(x=height)) +
                geom_histogram(stat ="count")
          p1
```

```
Warning message:
"Ignoring unknown parameters: binwidth, bins, pad"
```

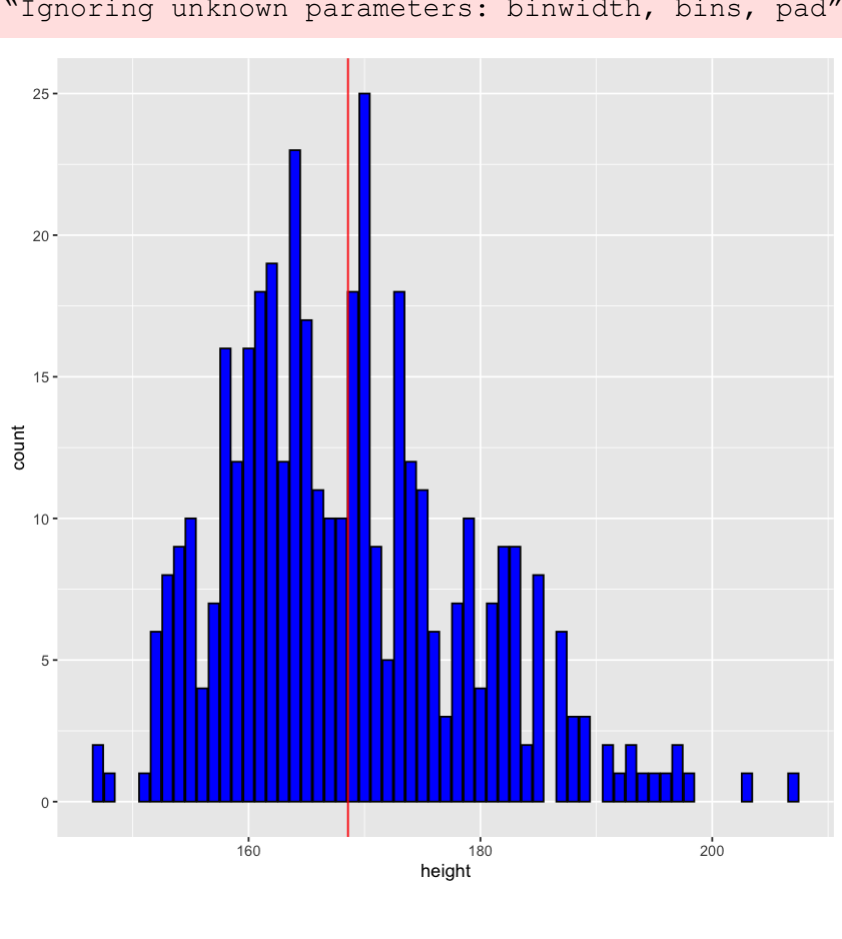


Adding a vertical line

We can easily add a vertical line using `geom_vline()`:

```
In [173]: p2 <- ggplot(data=height, aes(x=height)) +
                geom_histogram(stat ="count", color = "black", fill="blue") +
                geom_vline(aes(xintercept = mean(height)), color = "red")
          p2
```

```
Warning message:
"Ignoring unknown parameters: binwidth, bins, pad"
```

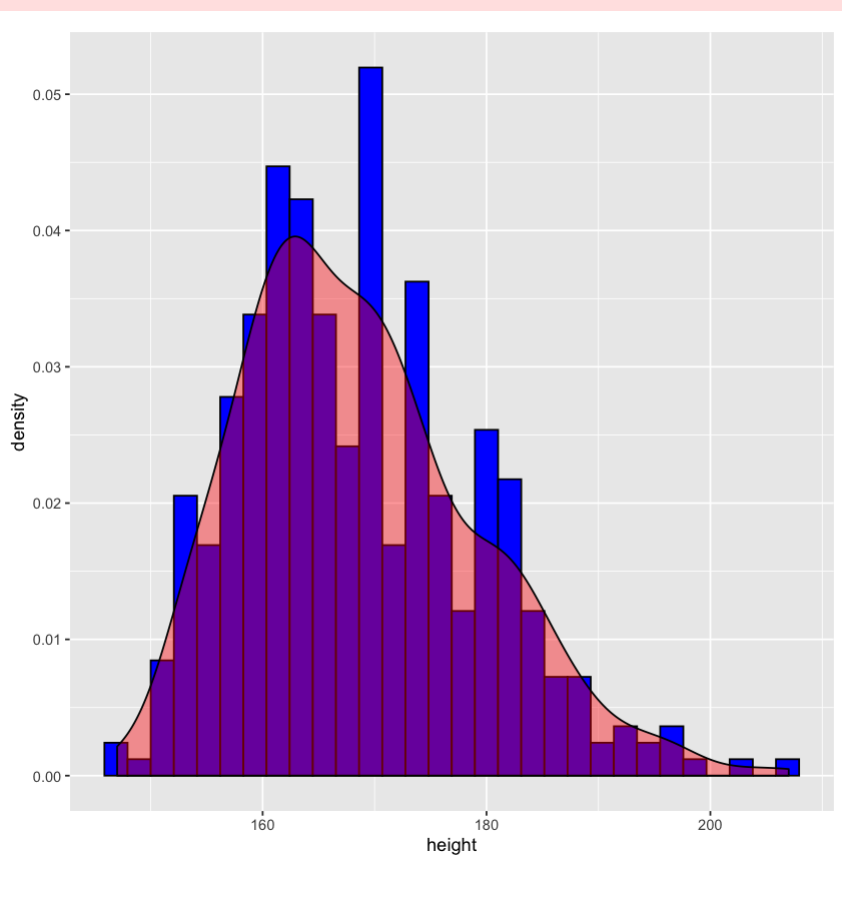


Density

We can overlay a transparent plot with the estimated kernel density with `geom_density()`

```
In [174]: p3 <- ggplot(data=height, aes(x=height)) +
                geom_histogram(aes(y=..density..), color = "black", fill="blue") +
                geom_density(alpha=.4, fill = "red") # alpha determines how transparent it is
          p3
```

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

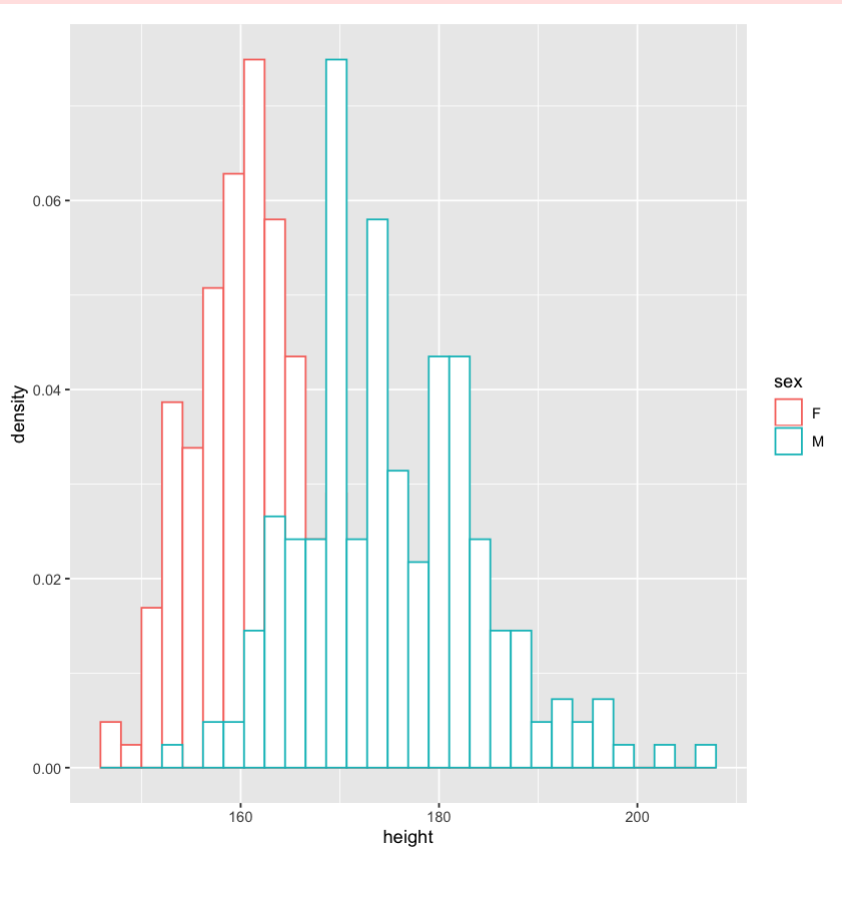


Overlapping histograms

With the option `position = identity` we can plot two histograms, one in front of the other:

```
In [175]: p4 <- ggplot(data=height, aes(x=height, color = sex)) +
                geom_histogram(aes(y=..density..), fill = "white", position="identity")        # Position identity
           for overlapping histograms
          p4
```

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

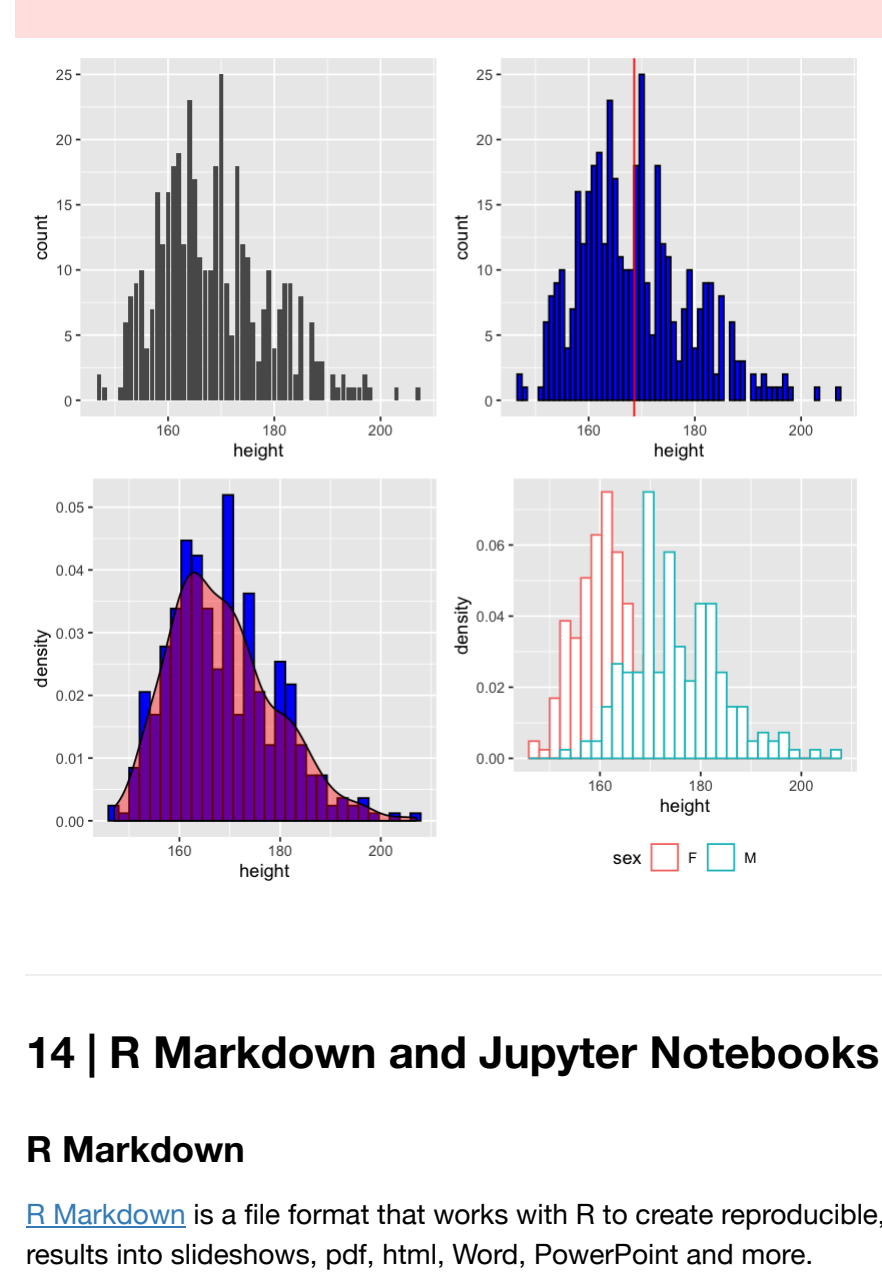


Arranging plots

Since we have stored each plot in a variable, we can use the `ggarrange()` function from the `ggpubr` package to plot each in a subplot:

```
In [176]: library(ggpubr)
          p <- ggarrange(p1, p2, p3, p4, ncol=2, nrow=2, common.legend = F, legend="bottom")
          p
```

```
Loading required package: magrittr


Attaching package: 'magrittr'


The following object is masked from 'package:purrr':

    set_names


The following object is masked from 'package:tidyr':

    extract



Attaching package: 'ggpubr'


The following object is masked from 'package:plyr':

    mutate


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

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



14 | R Markdown and Jupyter Notebooks

R Markdown

R Markdown is a file format that works with R to create reproducible, dynamic reports. You can use it to easily embed your code and results into slideshows, pdf, html, Word, PowerPoint and more.

You can create a Markdown file from RStudio by choosing **File > New File > R Markdown...**. You will be asked what type of output you would like to generate (html, pdf, word). Then you can start writing your document following a markdown syntax (check the *cheat sheet* in *Other references* for guidance). R Markdown allows you to embed code in the report that will be ran when the report is rendered and include the results.

Jupyter notebooks

Jupyter notebooks are file types with extension `.ipynb` produced by Project Jupyter which contain both computer code (e.g. R, Python, Julia), and rich text elements (paragraphs, equations, images, links, etc). They are growing in popularity and are a great way to elaborate reports and documents. They can be edited on a web server and exported as html, pdf via LaTeX, or other file formats. They are interactive and you can independently run pieces of code. As an example, this set of notes has been created on a Jupyter notebook.