Financial Risk Forecasting
Seminar 2

Jon Danielsson
London School of Economics

Version 4.0 August 2024

2 Data analysis

2.1 The plan for this week

1. Install and load packages

2. Basic data handling
3. Save data frames
4. Create, customise and export plots

2.2 Libraries / packages

One of the most useful features of R is its large library of packages. Although the base R language is very powerful, there are thousands of community built packages to perform specific tasks that can save us the effort of programming some things from scratch. The complete list of available R packages can be found on CRAN.

To install a package, type in the console: `install.packages("name_of_package")`, and after installed, you can load it with: `library(name_of_package)`.

It is considered a good practice to start your .R script with loading all the libraries that will be used in your program.

In the *Information* pane of RStudio, you can access the list of packages that are installed in your R environment:

By clicking on a package's name, you will see all the information it contains

2.3 Links from the R notebook

Processing data.

Plotting.

2.4 Installing some packages we will need for this course

Make sure you have the following packages installed:

```r
install.packages("zoo")
install.packages("lubridate")
install.packages("tseries")
install.packages("rugarch")
install.packages("rmgarch")
install.packages("reshape2")
install.packages("car")
```

2.5 Loading the packages needed for today

```r
library(reshape2)
library(lubridate)
```

2.6 Data Handling

The first thing to do when working with new data is cleaning it. Let's load the data downloaded in the previous class and take a look:

```r
data <- read.csv("crsp.csv")
head(data)
class(data)
```

We are interested in having a data frame that holds the price of each stock over time, and another one that holds the returns. We will use the current `TICKER` of the stocks as the identifying name for each company.

Before building our data frame, we need to adjust the prices with the Cumulative Factor to Adjust Prices, or `CFACPR`. The reason is that the `PRC` variable does not take into account stock splits, which can lead us to believe that the price of a stock halved in a day, when the reason for this drop is no more than a stock split. To adjust for this, we will divide the column `PRC` by `CFACPR`. For comparison, we will keep the unadjusted prices in a `UnAdjusted_Prices`.

```r
data$Unadjusted_Prices <- data$PRC
data$Adjusted_Prices <- data$PRC / data$CFACPR
head(data)
```

2.7 There are (at least) two ways to select prices into a data frame

2.7.1 The bad way

Now that we have the correct prices, we will select the date and price columns for each stock and put them into a variable with the ticker name. Afterwards, we have to rename the price column to the ticker name. For example, for MSFT:

```r
AAPL <- data[data$PERMNO == 14593, c("date", "Adjusted_Prices")]
names(AAPL)[2] <- "AAPL"
```

```r
MSFT <- data[data$PERMNO == 10107, c("date", "Adjusted_Prices")]
names(MSFT)[2] <- "MSFT"
XOM <- data[data$PERMNO==11850, c("date", "Adjusted_Prices")]
names(XOM)[2] <- "XOM"
GE <- data[data$PERMNO==12060, c("date", "Adjusted_Prices")]
names(GE)[2] <- "GE"
JPM <- data[data$PERMNO==47896, c("date", "Adjusted_Prices")]
names(JPM)[2] <- "JPM"
MCD <- data[data$PERMNO==43449, c("date", "Adjusted_Prices")]
names(MCD)[2] <- "MCD"
C <- data[data$PERMNO==70519, c("date", "Adjusted_Prices")]
names(C)[2] <- "C"
PRC <- merge(MSFT, XOM)
PRC <- merge(PRC, GE)
PRC <- merge(PRC, JPM)
PRC <- merge(PRC, MCD)
PRC <- merge(PRC, AAPL)
PRC <- merge(PRC, C)
head(PRC)
```

We got the output we wanted, but it involved several lines of basically copy-pasting the same code.

It also really easy to make mistakes.

As a challenge, you can try to replicate the process using a `for` loop.

2.7.2 The good way

Also, we could have saved us this trouble by using a package. R has thousands of packages with functions that can help us easily get the output we are looking for. We are going to create another table using the `dcast` function from the `reshape2` package.

We need to ensure the tickers map to the correct PERMNO.

```r
Tickers= dcast(data, date ~ PERMNO, value.var = "TICKER")
Tickers=tail(Tickers,1)
Tickers
Prices = dcast(data, date ~ PERMNO, value.var = "Adjusted_Prices")
names(Prices) <- Tickers
names(Prices)[1]="date"
head(Prices)
dim(Prices)
```

Can you explain how these commands work?

```r
UnAdjustedPrices = dcast(data, date ~ PERMNO, value.var = "PRC")
names(UnAdjustedPrices) <- Tickers
names(UnAdjustedPrices)[1]="date"
head(UnAdjustedPrices)
dim(UnAdjustedPrices)
```

We can now directly create the data frame for returns:

```
simpleReturns <- dcast(data, date ~ PERMNO, value.var = "RET")
names(simpleReturns) <- Tickers
names(simpleReturns)[1]="date"
head(simpleReturns)
```

2.8 Transforming simple returns to compound

The returns in our dataset are simple returns. To transform them into compound returns, we use the `log()` function:

```
Returns <- log(1 + simpleReturns[,2:dim(simpleReturns)[2]])
Returns$date <- simpleReturns$date
head(Returns)
```

Now the date is the last column, OK, but better for consistency if its the first. We can fix that by:

```
Returns =Returns[,c(dim(Returns)[2],1:(dim(Returns)[2]-1))]
head(Returns)
```

Can you explain how this command works?

2.9 Date formats

We have a column for the dates of the observations. R has a special variable type for working with dates called `Date`, which will make our lives easier when trying to do plots and some analyses. However, by default the date column in our dataset is not in this format:

```
class(Returns$date)
```

There are several ways to transform data into the `Date` type. We will use the package `lubridate` that we installed earlier. In particular, the function `ymd()` that stands for year-month-day. It is a powerful function that will turn any character in that order into a `Date` format. For example, it can handle:

```
ymd("20101005")
ymd("2010-10-05")
ymd("2010/10/05")
```

Likewise, you could also use `dmy()` or `mdy()` for different formats.

```
date.ts <- ymd(Returns$date)
class(date.ts)
```

2.10 Saving data frames

After handling data, we want to make sure we do not have to do the same procedure every time we open our program. For this, we can easily save the data frame we have created as an `.RData` object that can be loaded the next time we open R. To do so, we need to make sure we are at the Directory where we want to save the data, and use the `save()` function:

```
save(simpleReturns, file = "simpleReturns.RData")
save(Returns, file = "Returns.RData")
save(Prices, file = "Prices.RData")
save(UnAdjustedPrices, file = "UnAdjustedPrices.RData")
```

To load a `.RData` file, we need to use the function `load`:

```
load("Prices.RData")
head(Prices)
```

2.10.1 Saving as .csv

We can also easily save our data frames as a .csv file using the `write.csv()` function:

```
write.csv(Prices, file = "Prices.csv")
```

2.11 Plotting

The base R has an easy to use plot function called `plot()`. In this section we will learn how to customize our plots, add titles, change axis labels, add legends, select types of lines, vary the colors, and create subplots. See the plot chapter in the risk forecasting notebook

```
plot(Returns$JPM)
plot(Returns$JPM, type = "l")
```

```
plot(Returns$JPM,
    type = "l",
    main = "Compound returns for JP Morgan",
    ylab = "Returns",
    xlab = "Observation",
    col = "red",
    las=1
)
```

We would like to have the dates in the x axis to understand our data better:

```
plot(date.ts, Returns$JPM,
    type = "l",
    main = "Compound returns for JP Morgan",
    ylab = "Returns",
    xlab = "Date",
    col = "red",
    las = 1
)
```

2.11.1 Multiple graphs

What if we wanted to plot the prices for JP Morgan and Citi, the two major banks in our dataset? eated, or using the function `matplot()`:

```r
plot(date.ts, Prices$JPM, type = "l", main = "Prices for JP Morgan and Citi",
    ylab = "Price", xlab = "Date", col = "red")
lines(date.ts, Prices$C, col = "blue")
legend("bottomright",legend = c("JPM", "C"), col = c("red", "blue"), lty=1,bty='n')
```

There is something wrong with this plot. Since we first plotted the prices for JPM, the limits of the plot were determined to fit this data. However, when plotting the prices for C, we see some go below the limits of our plot. We can fix this with the `ylim()` option:

```r
plot(date.ts, Prices$JPM,
    type = "l",
    main = "Prices for JP Morgan and Citi",
    ylab = "Price",
    xlab = "Date",
    col = "red",
    ylim = c(0, 600)
)
lines(date.ts, Prices$C, col = "blue")
legend("topright",
    legend = c("JPM", "C"),
    col = c("red", "blue"),
    lty=1
)
```

Or just plot `C` first.

Or even

```r
matplot(date.ts, Prices[,2:dim(Prices)[2]],
    type = "l",
    lty=1,
    main = "Prices ",
    ylab = "Price",
    xlab = "Date",
    col = 1:9,
    las=1
)
```

Now that we know how to visualise two time series in the same plot, we can do a visual comparison of the adjusted and unadjusted prices for a stock:

```r
matplot(date.ts,
    cbind(Prices$MSFT,UnAdjustedPrices$MSFT),
    type = "l",
    lty=1,col = 2:3,
    main = "Adjusted and unadjusted prices for MSFT",
    ylab = "USD"
)
```

In some cases, plotting many time series in the same space can be messy, like if we wanted to plot all returns:

```r
matplot(date.ts,Returns[,2:dim(Returns)[2]],
    type = "l",
    main = "Returns for our stocks",
    ylab = "Returns",
    lty = 1,
    las=1
)
legend("bottomright",
    legend = names(Returns[,2:dim(Returns)[2]]),
    col = c(1:6),
    lty=1,
    bty='n'
)
```

For a better visualization, we will use `par(mfrow = c(a,b))`. This option divides the plotting area into a grid with `a` rows and `b` columns, each which will hold a subplot:

```r
par(mfrow = c(4,2))
for (i in 2:dim(Returns)[2]) {
    plot(date.ts, Returns[,i],
        type = "l",
        ylab = "Returns",
        xlab = "Date",
        main = paste("Returns for", names(Prices)[i])
    )
}
```

2.12 Exporting a plot

For this course you will have to create reports that include plots from R. The best way to do that, is to export the plots from RStudio. You can easily do that by clicking on `Export` in the Plot Viewer, and either save the plot, or copy it to your clipboard to paste it in a document. Or you can use functions such as `svg()`, `pdf()`, `bmp()` and `pdf()` for saving the figure automatically.

The easiest way might be to use quarto, which we discuss in a later seminar.

2.13 Recap

In this seminar we have covered:

- How to install and load packages
- Create a different column for each stock manually and using a package
- Working with date formats
- Saving data as .RData and .csv
- Customizing plots with titles, axis labels, colors and axis limits
- Adding legends to plots
- Plotting several time series in the same space
- Creating subplots
- Exporting plots

Some new functions used:

- `library()`
- `install.packages()`
- `class()`
- `dim()`
- `names()`
- `merge()`
- `dcast()`
- `log()`
- `ymd()`
- `save()`
- `load()`
- `write.csv()`
- `lines()`
- `legend()`
- `matplot()`
- `par()`

2.14 Optional exercises

3. Make a plot with all adjusted prices, and normalise them to start at 1 so we can easily compare the stocks.
4. Repeat the last plot, but use log y-axis.