Lines are models

Léo Belzile

version of 2018-09-18

Contents

Chapter 1

Introduction

We shall use the **R** programming language througout the course. (as it is free and it is used in other statistics courses at EPFL). Visit the R-project website[1] to download the program. The most popular graphical cross-platform front-end is RStudio Desktop[2].

R is an object-oriented interpreted language. It differs from usual programming languages in that it is designed for interactive analyses.

Since **R** is not a conventional programming language, my teaching approach will be learning-by-doing. The benefit of using *Rmarkdown* is that you see the output directly and you can also copy the code.

1.1 Basics of R

You can find several introductions to **R** online. Have a look at the **R** manuals[3] or better at contributed manuals[4]. A nice official reference is An introduction to **R**[5]. You may wish to look up the following chapters of the **R** language definition (Evaluation of expressions[6] and part of the *Objects* chapter[7]).

1.1.1 Help

Help can be accessed via `help` or simply `?`. If you do not know what to query, use `??` in front of a string, delimited by captions `" "` as in `??"Cholesky decomposition"`. Help is your best friend if you don't know what a function does, what are its arguments, etc.

1.1.2 Basic commands

Basic **R** commands are fairly intuitive, especially if you want to use your **R** as a calculator. Elementary functions such as `sum`, `min`, `max`, `sqrt`, `log`, `exp`, etc., are self-explanatory.

- Use `<-` to assign to a variable, and `=` for matching arguments inside functions
- Indexing in **R** starts at 1, **not** zero.

[1] https://cran.r-project.org/
[2] https://www.rstudio.com/products/rstudio/download/
[3] https://cran.r-project.org/manuals.html
[4] https://cran.r-project.org/other-docs.html
[5] http://colinfay.me/intro-to-r/index.html
[6] http://colinfay.me/r-language-definition/evaluation-of-expressions.html
[7] http://colinfay.me/r-language-definition/objects.html

- Most functions in **R** are vectorized, so avoid loops as much as possible.
- Integers are obtained by appending `L` to the number, so `2L` is an integer and `2` a double.

Besides integers and doubles, the common types are - logicals (`TRUE` and `FALSE`); - null pointers (`NULL`), which can be assigned to arguments; - missing values, namely `NA` or `NaN`. These can also be obtained a result of invalid mathematical operations such as `log(-2)`.

The above illustrates a caveat of **R**: invalid calls will often returns *something* rather than an error. It is therefore good practice to check that the output is sensical.

1.1.3 Linear algebra in R

R is an object oriented language, and the basic elements in **R** are (column) vector. Below is a glossary with some useful commands for performing basic manipulation of vectors and matrix operations:

- `c` creates a vector
- `cbind` (`rbind`) binds column (row) vectors
- `matrix` and `vector` are constructors
- `diag` creates a diagonal matrix (by default with ones)
- `t` is the function for transpose
- `solve` performs matrix inversion
- `%*%` is matrix multiplication, `*` is element-wise multiplication
- `crossprod` `crossprod(A, B)` calculates the cross-product, `t(A) %*% B` of two matrices `A` and `B`
- `eigen`/`chol`/`qr` perform respectively an eigendecomposition/Cholesky/QR decomposition of a matrix
- `rep` creates a vector of duplicates, `seq` a sequence. For integers i, j with $i < j$, `i:j` generates the sequence $i, i+1, \ldots, j-1, j$.

Subsetting is fairly intuitive and general; you can use vectors, logical statements. For example, if `x` is a vector, then

- `x[2]` returns the second element
- `x[-2]` returns all but the second element
- `x[1:5]` returns the first five elements
- `x[(length(x) - 5):length(x)]` returns the last five elements
- `x[c(1, 2, 4)]` returns the first, second and fourth element
- `x[x > 3]` return any element greater than 3. Possibly an empty vector of length zero!
- `x[ x < -2 | x > 2]` multiple logical conditions.
- `which(x == max(x))` index of elements satisfying a logical condition.

For a matrix `x`, subsetting now involves dimensions: `[1,2]` returns the element in the first row, second column. `x[,2]` will return all of the rows, but only the second column. For lists, you can use `[[` for subsetting by index or the `$` sign by names.

1.1.4 Packages

The great strength of **R** comes from its contributed libraries (called packages), which contain functions and datasets provided by third parties. Some of these (`base`, `stats`, `graphics`, etc.) are loaded by default whenever you open a session.

To install a package from CRAN, use `install.packages("package")`, replacing `package` by the package name. Once installed, packages can be loaded using `library(package)`; all the functions in `package` will be available in the environment.

The drawback of loading packages is that, if a function with the same name from another package is already present in your environment, it will be hidden. To palliate to this, you can use the `::` syntax to access a single function from an installed package, following the model `package::function`.

1.2 Tutorial 1

1.2.1 Data sets

We start by loading a dataset of the Old Faithful Geyser of Yellowstone National park. - Data sets are typically stored inside a `data.frame`, a matrix-like object whose columns contain the variables and the rows the observation vectors. - The columns can be of different types (`integer`, `double`, `logical`, `character`), but all the column vectors must be of the same length. - Variable names can be displayed by using `names(faithful)`. - Individual columns can be accessed using the column name using the `$` operator. For example, `faithful$eruptions` will return the first column of the `faithful` dataset. - To load a package already present in an **R** package, use the command `data` with the name of the `package` as an argument (must be a string).

The following functions can be useful to get a quick glimpse of the data:

- `summary` provides descriptive statistics for the variable.
- `str` provides the first few elements with each variable, along with the dimension
- `head` (`tail`) prints the first (last) n lines of the object to the console (default is $n = 6$).

```
# Load Old faithful dataset
data(faithful, package = "datasets")
# Query the database for documentation
?faithful
# look at first entries
head(faithful)
```

```
##   eruptions waiting
## 1     3.600      79
## 2     1.800      54
## 3     3.333      74
## 4     2.283      62
## 5     4.533      85
## 6     2.883      55
```

```
str(faithful)
```

```
## 'data.frame':    272 obs. of  2 variables:
##  $ eruptions: num  3.6 1.8 3.33 2.28 4.53 ...
##  $ waiting  : num  79 54 74 62 85 55 88 85 51 85 ...
```

```
# What kind of object is faithful?
class(faithful)
```

```
## [1] "data.frame"
```

Other common classes are `matrix` and `list`. The former has attributes `dim`, `ncol` and `nrow` in addition to `length`, which gives the total number of elements. A `list` is an unstructured class whose elements are

accessed using double indexing `[[ ]]` and elements are typically accessed using `$` symbol with names. To delete an element from a list, assign `NULL` to it. `data.frame` is a special type of list where all the elements are vectors of potentially different type, but of the same length.

1.2.2 Graphics

The `faithful` dataset consists of two variables: the regressand `waiting` and the regressor `eruptions`. One could postulate that the waiting time between eruptions will be smaller if the eruption time is small, since pressure needs to build up for the eruption to happen. We can look at the data to see if there is a linear relationship between the variables.

An image is worth a thousand words and in statistics, visualization is crucial. Scatterplots are produced using the function `plot`. You can control the graphic console options using `par` — see `?plot` and `?par` for a description of the basic and advanced options available.

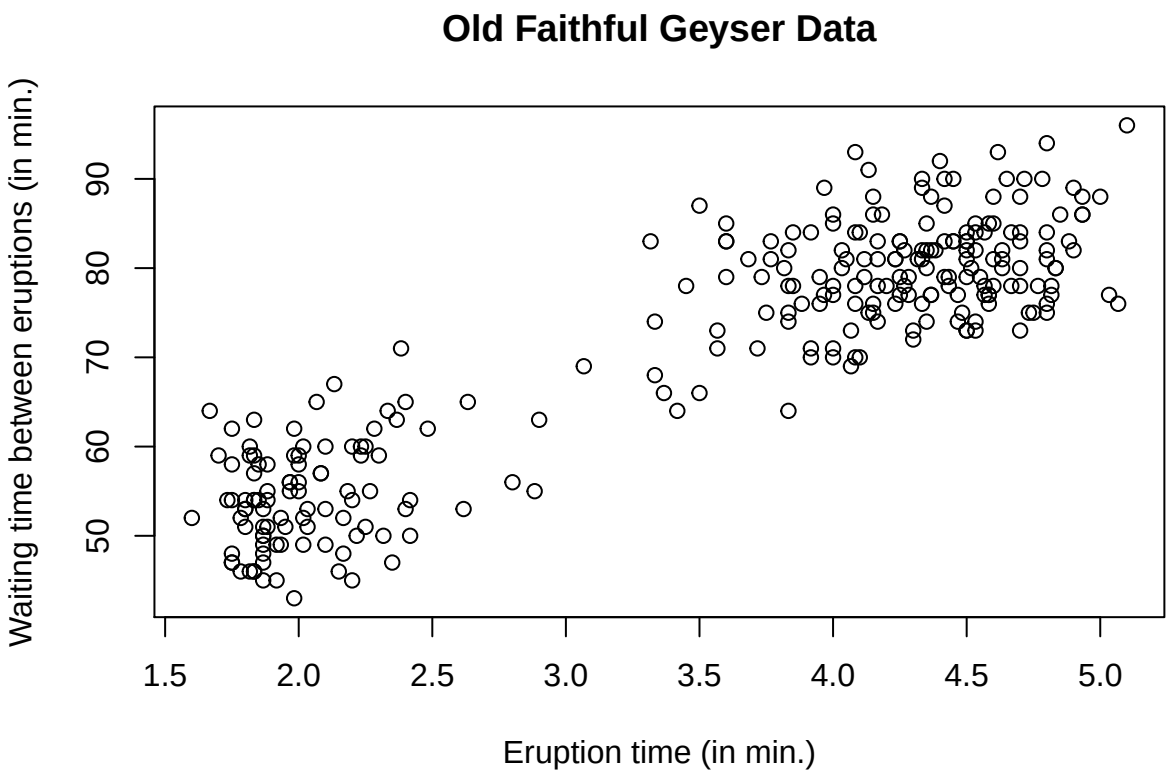
Once `plot` has been called, you can add additional observations as points (lines) to the graph using `point` (`lines`) in place of `plot`. If you want to add a line (horizontal, vertical, or with known intercept and slope), use the function `abline`.

Other functions worth mentioning at this stage:

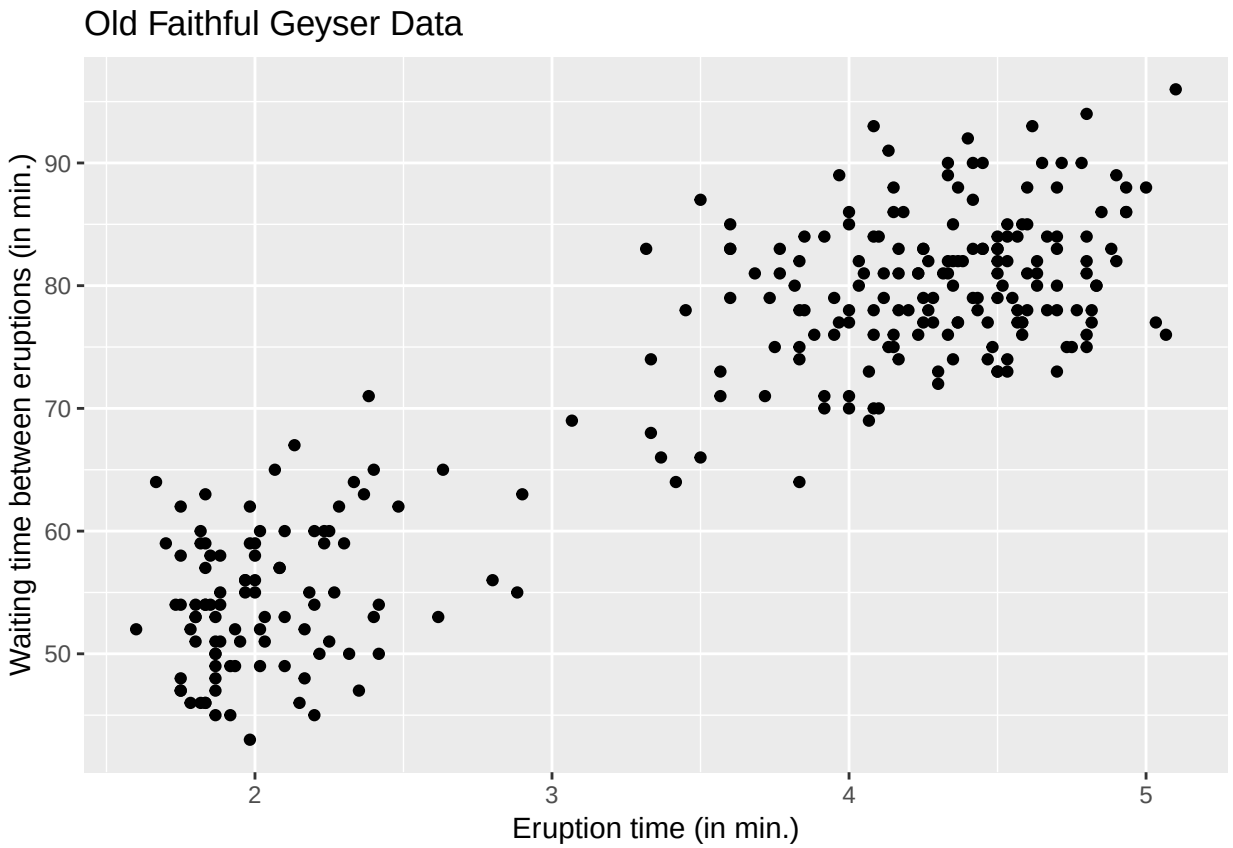
- `boxplot` creates a box-and-whiskers plot
- `hist` creates an histogram, either on frequency or probability scale (option `freq = FALSE`). `breaks` control the number of bins. `rug` adds lines below the graph indicating the value of the observations.
- `pairs` creates a matrix of scatterplots, akin to `plot` for data frame objects.

There are two options for basic graphics: the base graphics package and the package `ggplot2`. The latter is a more recent proposal that builds on a modular approach and is more easily customizable — I suggest you stick to either and `ggplot2` is a good option if you don't know **R** already, as the learning curve will be about the same. Even if the display from `ggplot2` is nicer, this is no excuse for not making proper graphics. Always label the axis and include measurement units!

```
# Scatterplots
# Using default R commands
plot(waiting ~ eruptions, data = faithful,
     xlab = "Eruption time (in min.)",
     ylab = "Waiting time between eruptions (in min.)",
     main = "Old Faithful Geyser Data")
```




```
#using the grammar of graphics (more modular)
#install.packages("ggplot2") #do this once only
library(ggplot2)
ggplot2::ggplot(data = faithful, aes(x = eruptions, y = waiting)) +
  geom_point() +
  labs(title = "Old Faithful Geyser Data",
       x = "Eruption time (in min.)",
       y = "Waiting time between eruptions (in min.)")
```



A simple linear model of the form

$$y_i = \beta_0 + \beta_1 \mathrm{x}_i + \varepsilon_i,$$

where ε_i is a noise variable with expectation zero and $\mathbf{x} =$ eruptions and $\boldsymbol{y} =$ waiting. We first create a matrix with a column of $\mathbf{1}_n$ for the intercept. We bind vectors by column (`cbind`) into a matrix, recycling arguments if necessary. Use `$` to obtain a column of the data frame based on the name of the variable (partial matching is allowed, e.g., `faithful$er` is equivalent to `faithful$eruptions` in this case).

```
## Manipulating matrices
n <- nrow(faithful)
p <- ncol(faithful)
y <- faithful$waiting
X <- cbind(1, faithful$eruptions)
```

1.2.3 Projection matrices

Recall that $\mathbf{H}_\mathbf{X} \equiv \mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$ is the orthogonal projection matrix onto span$(\mathbf{X})$. The latter has $p = 2$ eigenvalues equal to 1, is an $n \times n$ matrix of rank p, is symmetric and idempotent. We can verify the properties of $\mathbf{H}_\mathbf{X}$ numerically.

Whereas we will frequently use `==` to check for equality of booleans, the latter should be avoided for comparisons because computer arithmetic is exact only in base 2. For example, `1/10 + 2/10 - 3/10 == 0` will return `FALSE`, whereas `all.equal(1/10 + 2/10 - 3/10, 0)` will return `TRUE`. Us `all.equal` to check for equalities.

```r
Hx <- X %*% solve(crossprod(X)) %*% t(X)
# Create projection matrix onto complement
# `diag(n)` is the n by n identity matrix
Mx <- diag(n) - Hx
#Check that projection leaves X invariant
isTRUE(all.equal(X, Hx %*% X))
```

```
## [1] TRUE
```

```r
#Check that orthogonal projection maps X to zero matrix of dimension (n, p)
isTRUE(all.equal(matrix(0, nrow = n, ncol = p), Mx %*% X))
```

```
## [1] TRUE
```

```r
#Check that the matrix Hx is idempotent
isTRUE(all.equal(Hx %*% Hx, Hx))
```

```
## [1] TRUE
```

```r
#Check that the matrix Hx is symmetric
isTRUE(all.equal(t(Hx), Hx))
```

```
## [1] TRUE
```

```r
#Check that only a two eigenvalue are 1 and the rest are zero
isTRUE(all.equal(eigen(Hx, only.values = TRUE)$values, c(rep(1, p), rep(0, n - p))))
```

```
## [1] TRUE
```

```r
#Check that the matrix has rank p
isTRUE(all.equal(Matrix::rankMatrix(Hx), p, check.attributes = FALSE))
```

```
## [1] TRUE
```

1.2.4 Your turn

- Install the **R** package `ISLR` and load the dataset `Auto`. Be careful, as **R** is case-sensitive.
- Query the help file for information about the data set.
- Look at the first lines of `Auto`
- Create an explanatory variable `x` with horsepower and mileage per gallon as response `y`.
- Create a scatterplot of `y` against `x`. Is there a linear relationship between the two variables?
- Append a vector of ones to `x` and create a projection matrix.
- Check that the resulting projection matrix is symmetric and idempotent.

1.3 Exercises

(1.4) Oblique projections

Suppose that $\dim(\mathcal{M}(\mathbf{X})) \neq \dim(\mathcal{M}(\mathbf{W}))$. An oblique projection matrix is of the form $\mathbf{P} \equiv \mathbf{X}(\mathbf{W}^\top\mathbf{X})^{-1}\mathbf{W}^\top$ and appears in instrumental variable regression. The oblique projection is such that $\text{im}(\mathbf{P}) = \mathcal{M}(\mathbf{X})$, but $\text{im}(\mathbf{I}-\mathbf{P}) = \mathcal{M}(\mathbf{W}^\perp)$. This fact is illustrated below.

```
#Create two vectors (non-parallel)
x <- c(1, 2)
w <- c(-1, 0.1)
#Create usual hat projection matrix and form P
H <- x %*% solve(t(x) %*% x) %*% t(x)
#solve: matrix inverse
P <- x %*% solve(t(w) %*% x) %*% t(w)

isTRUE(all.equal((P %*% P), P)) #P is idempotent
```

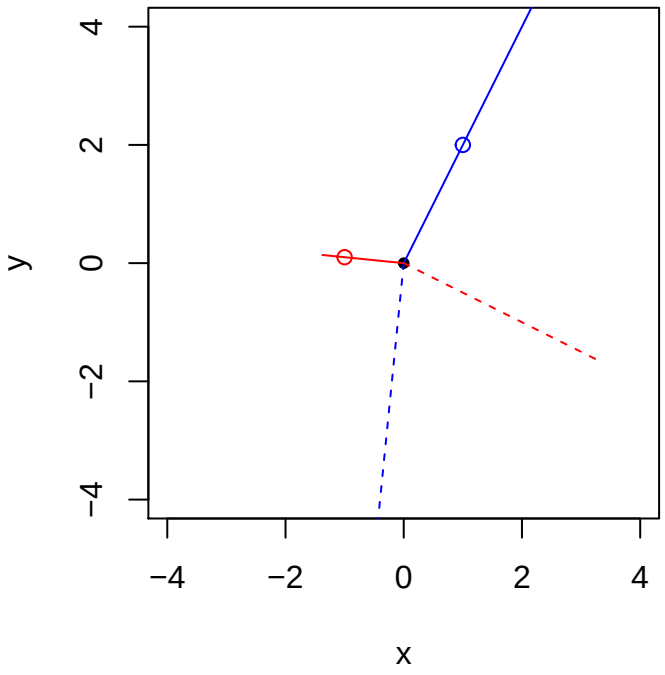
```
## [1] TRUE
```

```
P - t(P) #but not symmetric
```

```
##       [,1]   [,2]
## [1,] 0.000 -2.625
## [2,] 2.625  0.000
```

```
#Project a vector `vec' onto P, P transpose, I-P, I-(P transpose)
vec <- c(1.9, -1.5)
vec_P <- P %*% vec
vec_Pt <- t(P) %*% vec
vec_Id_minus_P <- (diag(2)-P) %*% vec
#diag: diagonal matrix, default to identity
vec_Id_minus_Pt <- (diag(2)-t(P)) %*% vec

#Plot the resulting vector along with the two vectors x and w (points)
par(pty = "s") #graphical console parameters (square region)
plot(NULL, xlab = "x", ylab = "y", xlim = c(-4, 4), ylim = c(-4, 4))
#create empty plot with labels x and y, on square region [-4,4]^2
points(0, 0, pch = 20) #points: add points to existing plot
points(x[1], x[2], col = "blue")
points(w[1], w[2], col = "red")
# blue line for P, dashed blue for I-P, red for Pt, red dashed for I-Pt
segments(x0 = 0, y0 = 0, x1 = vec_P[1], y1 = vec_P[2], col = "blue")
segments(x0 = 0, y0 = 0, x1 = vec_Pt[1], y1 = vec_Pt[2], col = "red")
segments(x0 = 0, y0 = 0, x1 = vec_Id_minus_P[1], y1 = vec_Id_minus_P[2], col = "blue", lty = 2)
segments(x0 = 0, y0 = 0, x1 = vec_Id_minus_Pt[1], y1 = vec_Id_minus_Pt[2], col = "red", lty = 2)
```


y
4
2
0
−2
−4
−4
−2
0
2
4
x