lineaRmodels

Léo Belzile

version of 2018-09-23

Contents

Preliminary remarks

This is a web complement to MATH 341 (Linear Models), a first regression course for EPFL mathematicians.

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] https://cran.r-project.org/
[2] https://www.rstudio.com/products/rstudio/download/

Chapter 1

Introduction

You can find several introductions to **R** online. Have a look at the **R** manuals[1] or better at contributed manuals[2]. A nice official reference is An introduction to **R**[3]. You may wish to look up the following chapters of the **R** language definition (Evaluation of expressions[4] and part of the *Objects* chapter[5]).

1.1 Basics of R

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 **R** as a calculator. Elementary functions such as `sum`, `min`, `max`, `sqrt`, `log`, `exp`, etc., are self-explanatory.

Some unconventional features of the language:

- Use `<-` to assign to a variable, and `=` for matching arguments inside functions
- Indexing in **R** starts at 1, **not** zero.
- 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] https://cran.r-project.org/manuals.html
[2] https://cran.r-project.org/other-docs.html
[3] http://colinfay.me/intro-to-r/index.html
[4] http://colinfay.me/r-language-definition/evaluation-of-expressions.html
[5] http://colinfay.me/r-language-definition/objects.html

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(A, B)` calculates the cross-product $\mathbf{A}^\top\mathbf{B}$, `t(A) %*% B`, of two matrices `A` and `B`.
- `eigen`/`chol`/`qr`/`svd` perform respectively an eigendecomposition/Cholesky/QR/singular value 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.

There are drawbacks to loading packages: if an object with the same name from another package is already present in your environment, it will be hidden. Use the double-colon operator `::` to access a single object from an installed package (`package::object`).

1.2 Tutorial 1

1.2.1 Data sets

- 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 data set from an (installed) **R** package, use the command `data` with the name of the `package` as an argument (must be a string). The package `datasets` is loaded by default whenever you open **R**, so these are always in the search path.

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 $n = 6$).

We start by loading a dataset of the Old Faithful Geyser of Yellowstone National park and looking at its entries.

```
# 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 of objects: - `matrix`: an object with attributes `dim`, `ncol` and `nrow` in addition to `length`, which gives the total number of elements. - `array`: a higher dimensional extension of `matrix` with arguments `dim` and `dimnames`. - `list`: 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.

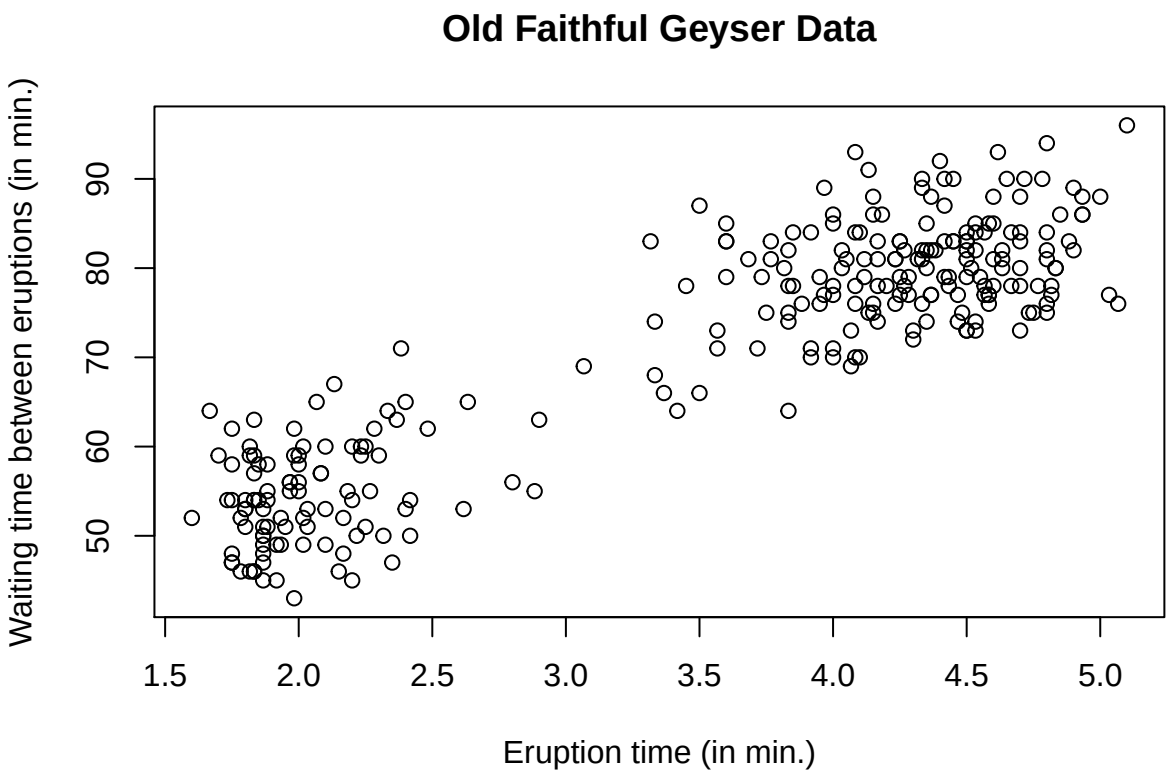
Once `plot` has been called, you can add additional observations as points (lines) to the graph using `point` (`lines`) in place of `plot`. If you want to add a line (horizontal, vertical, or with known intercept and slope), use the function `abline`.

Other functions worth mentioning at this stage:

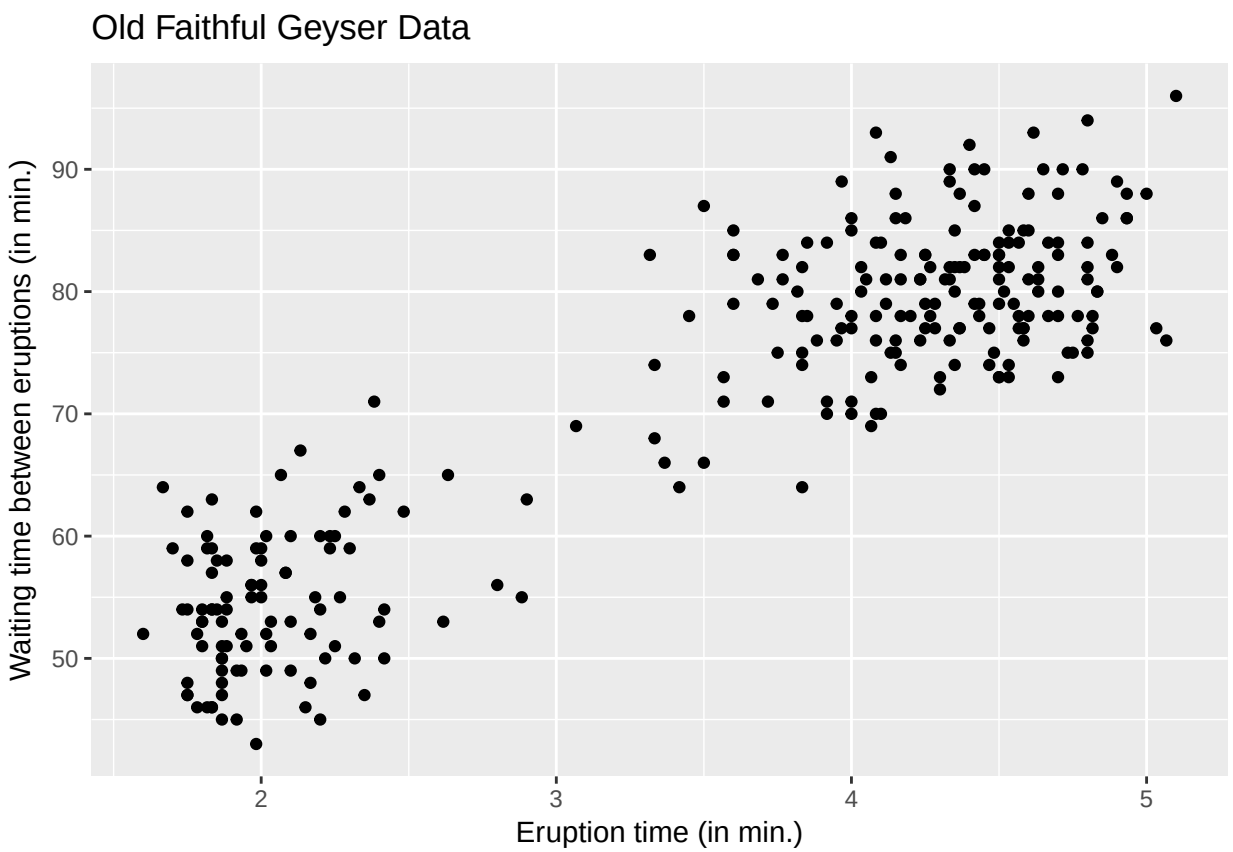
- `boxplot` creates a box-and-whiskers plot
- `hist` creates an histogram, either on frequency or probability scale (option `freq = FALSE`). `breaks` control the number of bins. `rug` adds lines below the graph indicating the value of the observations.
- `pairs` creates a matrix of scatterplots, akin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

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i,$$

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`. Use `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 Week 1 exercises

(1.4) Oblique projections

Suppose that $\mathsf{span}(\mathbf{X}) \neq \mathsf{span}(\mathbf{W})$, that both $\mathbf{X}$ and $\mathbf{W}$ are full-rank $n \times p$ matrices such that $\mathbf{X}^\top\mathbf{W}$ and $\mathbf{W}^\top\mathbf{X}$ are invertible. 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 $\mathrm{im}(\mathbf{P}) = \mathsf{span}(\mathbf{X})$, but $\mathrm{im}(\mathbf{I} - \mathbf{P}) = \mathsf{span}(\mathbf{W}^\perp)$. This fact is illustrated below.

We consider two non-parallel vectors in $\mathbb{R}^2$, $\mathbf{X}$ and $\mathbf{W}$. The figure shows the projection of a third vector (non-parallel to $\mathbf{X}$ or $\mathbf{W}$) onto the span of $\mathbf{P}$ (blue), $\mathbf{P}^\top$ (red), $\mathbf{I}_2 - \mathbf{P}$ (dashed blue) and $\mathbf{I}_2 - \mathbf{P}^\top$ (dashed red). The circles indicate the points $\mathbf{W}$ (red) and $\mathbf{X}$ (blue) on the plane. Notice that $\mathbf{I}_2 - \mathbf{P}^\top \perp \mathbf{P}$, whereas $\mathbf{I}_2 - \mathbf{P} \perp \mathbf{P}^\top$.

```r
#Create two vectors (non-parallel)
x <- c(1, 2)
w <- c(-1, 0.1)
#Create oblique projection matrix
P <- x %*% solve(t(w) %*% x) %*% t(w)

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

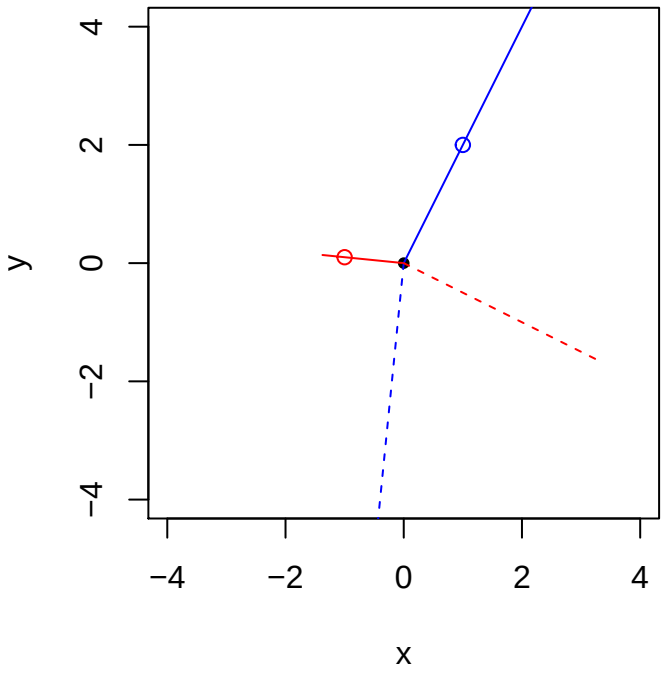
```
## [1] TRUE
```

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

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

```r
#Project a third 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

Chapter 2

Computational considerations

In this tutorial, we will explore some basic **R** commands and illustrate their use on the Motor Trend Car Road Tests dataset (`mtcars`).

2.1 Calculation of least square estimates

Consider as usual $\boldsymbol{y}$ and n-vector of response variables and a full-rank $n \times p$ design matrix $\mathbf{X}$. We are interested in finding the ordinary least square coefficient $\hat{\boldsymbol{\beta}}$, the fitted values $\hat{\boldsymbol{y}} = \mathbf{X}\hat{\boldsymbol{\beta}}$ and the residuals $\boldsymbol{e} = \boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}$.

Whereas orthogonal projection matrices are useful for theoretical derivations, they are not used for computations. Building $\mathbf{H_X}$ involves a matrix inversion and the storage of an $n \times n$ matrix. In Exercise series 2, we looked at two matrix decompositions: a singular value decomposition (SVD) and a QR decomposition. These are more numerically stable than using the normal equations $(\mathbf{X}^\top\mathbf{X})\boldsymbol{\beta} = \mathbf{X}^\top\boldsymbol{y}$ (the condition number of the matrix $\mathbf{X}^\top\mathbf{X}$ is the square of that of $\mathbf{X}$ — more on this later).

For more details about the complexity and algorithms underlying the different methods, the reader is referred to these notes of www.math.uchicago.edu/~may/REU2012/REUPapers/Lee.pdf[1]. This material is *optional*.

2.1.1 Normal equations

The following simply illustrates what has been derived in Exercise series 2. You will probably never use these commands, as **R** has devoted functions that are coded more efficiently. We can compute first the ordinary least square estimates using the formula $\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\boldsymbol{y}$.

```
data(mtcars)
y <- mtcars$mpg
X <- cbind(1, as.matrix(mtcars[,2:ncol(mtcars)]))
n <- nrow(X)
p <- ncol(X)
# Estimation of betahat:
XtX<- crossprod(X)
Xty <- crossprod(X, y)
# Solve normal equations
```

[1] Lee

```
betahat <- as.vector(solve(XtX, Xty))
#same as betahat <- solve(t(X) %*% X) %*% t(X) %*% y
```

2.1.2 Singular value decomposition

The SVD decomposition in **R** returns a list with elements `u`, `d` and `v`. `u` is the orthonormal $n \times p$ matrix, `d` is a vector containing the diagonal elements of $\mathbf{D}$ and `v` is the $p \times p$ orthogonal matrix. Recall that the decomposition is

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^\top$$

and that $\mathbf{V}\mathbf{V}^\top = \mathbf{V}^\top\mathbf{V} = \mathbf{U}^\top\mathbf{U} = \mathbf{I}_p$. The matrix $\mathbf{D}$ contains the singular values of $\mathbf{X}$, and the diagonal elements d_{ii}^2 corresponds to the (ordered) eigenvalues of $\mathbf{X}^\top\mathbf{X}$.

```
svdX <- svd(X)
# Projection matrix
Hx <- tcrossprod(svdX$u)
# t(U) %*% U gives p by p identity matrix
all.equal(crossprod(svdX$u), diag(p))
```

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

```
# V is an orthogonal matrix
all.equal(tcrossprod(svdX$v), diag(p))
```

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

```
all.equal(crossprod(svdX$v), diag(p))
```

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

```
# D contains singular values
all.equal(svdX$d^2, eigen(XtX, only.values = TRUE)$values)
```

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

```
# OLS coefficient from SVD
betahat_svd <- c(svdX$v %*%  diag(1/svdX$d) %*% t(svdX$u) %*% y)
all.equal(betahat_svd, betahat)
```

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

2.1.3 QR decomposition

R uses a QR-decomposition to calculate the OLS. There are specific functions to return coefficients, fitted values and residuals. One can also obtain the $n \times p$ matrix $\mathbf{Q}_1$ and the upper triangular $p \times p$ matrix $\mathbf{R}$ from the thinned QR decomposition,

$$\mathbf{X} = \mathbf{Q}_1\mathbf{R}.$$

```
qrX <- qr(X)
Q1 <- qr.Q(qrX)
R <- qr.R(qrX)
# Compute betahat from QR
betahat_qr1 <- qr.coef(qrX, y) #using built-in function
betahat_qr2 <- c(backsolve(R, t(Q1) %*% y)) #manually
all.equal(betahat, betahat_qr1, check.attributes = FALSE)
```

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

```
all.equal(betahat, betahat_qr2, check.attributes = FALSE)
```

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

```
# Compute residuals
qre <- qr.resid(qrX, y)
all.equal(qre, c(y - X %*% betahat), check.attributes = FALSE)
```

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

```
# Compute fitted values
qryhat <- qr.fitted(qrX, y)
all.equal(qryhat, c(X %*% betahat), check.attributes = FALSE)
```

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

```
# Compute orthogonal projection matrix
qrHx <- tcrossprod(Q1)
all.equal(qrHx, Hx)
```

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

2.2 Parameter estimation

We are now ready to fit a linear model with an intercept and a linear effect for the weight. The model will be of the form

$$\texttt{mpg}_i = \beta_0 + \texttt{wt}_i\beta_1 + \varepsilon_i.$$

We form the design matrix $(\mathbf{1}_n^\top, \texttt{wt}^\top)^\top$ and the vector of regressand `mpg`, then proceed with calculating the OLS coefficients $\hat{\boldsymbol{\beta}}$, the hat matrix $\mathbf{H}_\mathbf{X}$, the fitted values $\hat{\boldsymbol{y}}$ and the residuals $\boldsymbol{e}$.

```
#Design matrix
wt <- mtcars$wt
X <- cbind(1, wt)
mpg <- mtcars$mpg
#OLS estimates
XtXinv <- solve(crossprod(X))
beta_hat <- c(XtXinv %*% t(X) %*% mpg)
#Form orthogonal projection matrix
```

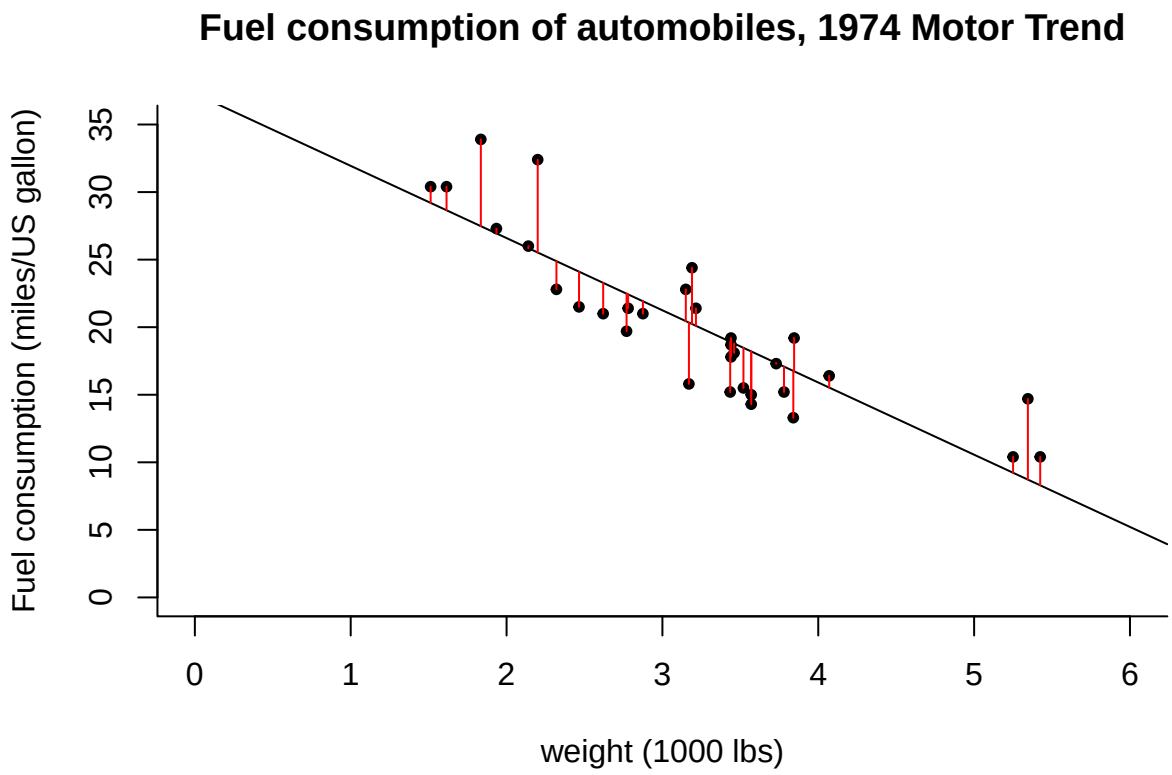
```
Hmat <- X %*% XtXinv %*% t(X)
#Create residuals and fitted values
fitted <- Hmat %*% mpg
res <- mpg - fitted
fitted <- Hmat %*% mpg
#Variance estimate and standard errors
s2 <- sum(res^2) / (length(res) - ncol(X))
std_err <- sqrt(diag(s2 * XtXinv))
```

The residuals $\boldsymbol{e} = \boldsymbol{y} - \hat{\boldsymbol{y}}$ can be interpreted as the *vertical* distance between the regression slope and the observation. This is illustrated in the following graph. For each observation y_i, a vertical line at distance e_i is drawn from the prediction $\hat{y}_i$.

Side remark: graphs and table should *always* be properly labelled (including units). The last line of the call to `plot` contains cosmetic options that alter the display of the scatterplot — you can check for yourself the effects of removing any (all) of these additional commands.

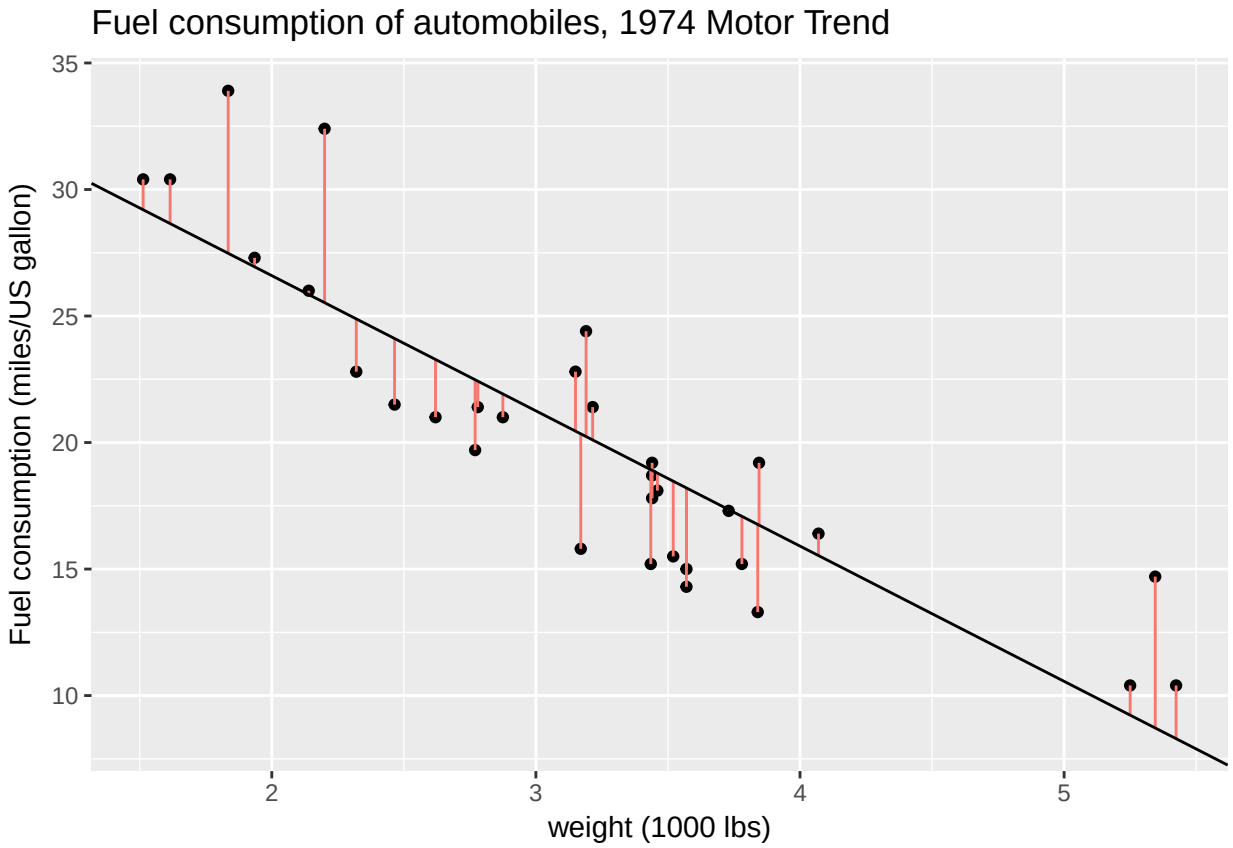
```
plot(mpg ~ wt, xlab = "weight (1000 lbs)", ylab = "Fuel consumption (miles/US gallon)",
     main = "Fuel consumption of automobiles, 1974 Motor Trend", data = mtcars,
     bty = "l", pch = 20, ylim = c(0, 35), xlim = c(0, 6)) #options to tweak the display
#Line of best linear fit
abline(a = beta_hat[1], b = beta_hat[2])

#Residuals are vertical distance from line to
for(i in 1:nrow(X)){
  segments(x0 = wt[i], y0 = fitted[i], y1 = fitted[i] + res[i], col = 2)
}
```



The same scatterplot, this time using `ggplot2`.

```
library(ggplot2, warn.conflicts = FALSE, quietly = TRUE)
#Create data frame with segments
vlines <- data.frame(x1 = mtcars$wt, y1 = fitted, y2 = fitted + res)
ggg <- ggplot(mtcars, aes(wt, mpg)) +
        geom_point() +
        labs(x = "weight (1000 lbs)",
            y = "Fuel consumption (miles/US gallon)",
            title = "Fuel consumption of automobiles, 1974 Motor Trend") +
      geom_segment(aes(x = x1, y = y1, xend = x1, yend = y2, color = "red"),
                   data = vlines, show.legend = FALSE) +
      geom_abline(slope = beta_hat[2], intercept = beta_hat[1])
print(ggg)
```



2.3 Interpretation

If the regression model is

$$y_i = \beta_0 + \mathrm{x}_{i1}\beta_1 + \mathrm{x}_{i2}\beta_2 + \varepsilon_i,$$

the interpretation of β_1 in the linear model is as follows: a unit increase in x leads to β_1 units increase in y, everything else (i.e., x_{i2}) being held constant.

For the `mtcars` regression above, an increase of the weight of the car of 1000 pounds leads to an average decrease of 5.34 miles per US gallon in distance covered by the car. We could easily get an equivalent statement in terms of increase of the car fuel consumption for a given distance.

2.4 The `lm` function

The function `lm` is the workshorse for fitting linear models. It takes as input a formula: suppose you have a data frame containing columns `x` (a regressor) and `y` (the regressand); you can then call `lm(y ~ x)` to fit the linear model $y = \beta_0 + \beta_1 x + \varepsilon$. The explanatory variable `y` is on the left hand side, while the right hand side should contain the predictors, separated by a `+` sign if there are more than one. If you provide the data frame name using `data`, then the shorthand `y ~ .` fits all the columns of the data frame (but `y`) as regressors.

To fit higher order polynomials or transformations, use the `I` function to tell **R** to interpret the input "as is". Thus, `lm(y~x+I(x^2))`, would fit a linear model with design matrix $(\mathbf{1}_n, \mathbf{x}^\top, \mathbf{x}^2)^\top$. A constant is automatically included in the regression, but can be removed by writing `-1` or `+0` on the right hand side of the formula.

```
# The function lm and its output
fit <- lm(mpg ~ wt, data = mtcars)
fit_summary <- summary(fit)
```

The `lm` output will display OLS estimates along with standard errors, t values for the Wald test of the hypothesis $\mathrm{H}_0 : \beta_i = 0$ and the associated P-values. Other statistics and information about the sample size, the degrees of freedom, etc., are given at the bottom of the table.

Many methods allow you to extract specific objects. For example, the functions `coef`, `resid`, `fitted`, `model.matrix` will return $\hat{\boldsymbol{\beta}}$, $\boldsymbol{e}$, $\hat{\boldsymbol{y}}$ and $\mathbf{X}$, respectively.

```
names(fit)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

```
names(fit_summary)
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```