Matrix methods – Least squares problems

MATH 2740 – Mathematics of Data Science – Lecture 05

Julien Arino
julien.arino@umanitoba.ca

Department of Mathematics @ University of Manitoba

Fall 202X

The University of Manitoba campuses are located on original lands of Anishinaabeg, Ininew, Anisininew, Dakota and Dene peoples, and on the National Homeland of the Red River Métis. We respect the Treaties that were made on these territories, we acknowledge the harms and mistakes of the past, and we dedicate ourselves to move forward in partnership with Indigenous communities in a spirit of Reconciliation and collaboration.

Outline

Getting the Canadian census data

Least squares problem – Initial considerations

Getting the Canadian census data

Least squares problem – Initial considerations

Remark

This is the slide version of `Canada-census-get-plot-data.Rmd` in the folder `CODE` of the course repository

The `html` file generated there (link) is much easier to read than these slides

Canadian census data

We want to consider the evolution of the population of Canada through time

For this, we grab the Canadian census data

Search for (Google) “Canada historical census data csv", since csv (comma separated values) is a very easy format to use with R

Here, we find a csv for 1851 to 1976

We follow the link to Table A2-14, where we find another link, this time to a csv file. This is what we use in R

Grabing the Canadian census data

The function `read.csv` reads in a file (potentially directly from the web)

```
prefix = "https://www150.statcan.gc.ca/n1/en/pub/11-516-x/sectiona/"
page = "A2_14-eng.csv?st=L7vSnqio"
url = paste0(prefix, page)
data_old = read.csv(url)
```

This directly assigns the result to the variable `data`

We then use the function `head` to show the first few lines in the result

```
head(data_old)
```

```
##    X Series.A2.14.
## 1 NA
## 2 NA          Year
## 3 NA
## 4 NA
## 5 NA
## 6 NA
##   Population.of.Canada..by.province..census.dates..1851.to.1976 X.1
## 1                                                                NA
## 2                                                        Canada  NA Newfo
## 3                                                                NA
## 4                                                                NA
## 5                                                             2  NA
## 6                                                                NA
##      X.3 X.4    X.5       X.6    X.7     X.8       X.9 X.10    X.11 X.12
## 1             NA                                          NA           NA
```

Obviously, this does not make a lot of sense. This is normal: take a look at the first few lines in the file. They take the form

```
head(data_old, n = 1)

##    X Series.A2.14.
## 1 NA
##   Population.of.Canada..by.province..census.dates..1851.to.1976 X.1 X.2 X
## 1                                                                 NA
##   X.5 X.6 X.7 X.8 X.9 X.10 X.11 X.12 X.13 X.14 X.15 X.16 X.17 X.18 X.19 X
## 1                      NA        NA        NA        NA             NA
```

This happens often: the first few lines are here to describe the information contained in the data set, i.e., the so-called **metadata**

It is easy to deal with this: the function `read.csv` takes the optional argument `skip`, which indicates how many lines to skip at the beginning of the file

The second line is also empty, so let us skip it too

```
data_old = read.csv(url, skip = 2)
head(data_old, n = 2)

##    X Year Canada X.1 Newfound. Prince X.2   Nova       New Quebec Ontario
## 1 NA              NA      land Edward  NA Scotia Brunswick
## 2 NA              NA           Island  NA
##   Manitoba X.3 Saskat. X.4 Alberta X.5  British X.6     Yukon   Northwest
## 1           NA  chewan  NA          NA Columbia  NA Territory Territories
## 2           NA          NA          NA           NA
##   X.8
## 1  NA
## 2  NA
```

To make things legible, the table authors used 3 rows to write long names (e.g., Prince Edward Island is written over 3 rows)

Note, however, that `read.csv` has rightly picked up on the first row being the column names

Because we are only interested in the total population of the country and the year, let us get rid of the first 4 rows and of all columns except the second (Year) and third (Canada)

```
data_old = data_old[5:dim(data_old)[1], 2:3]
head(data_old, n=4)
```

```
##   Year     Canada
## 5 1976 22,992,604
## 6
## 7 1971 21,568,311
## 8 1966 20,014,880
```

Still not perfect...

```
head(data_old, n=3)

##   Year     Canada
## 5 1976 22,992,604
## 6
## 7 1971 21,568,311

tail(data_old, n=2)

##                            Year Canada
## 28 see notes to series A15-53.
## 29                1848 figure.
```

1. there are some empty rows
2. the last few rows need to be removed too, they contain remarks about the data
3. the population counts contain commas
4. it would be better if years were increasing

Fixing these issues

1 and 2 are easy: remark that the Canada column is empty for both issues. Entries in the column are strings. Looking for empty content therefore means looking for empty strings

So to fix 1 and 2, we keep the rows where Canada does not equal the empty string

To get rid of commas, we just need to substitute an empty string for ","

To sort, we find the order for the years and apply it to the entire table

Finally, as remarked above, for now, both the year and the population are considered as strings. To plot, we will have to indicate that these are numbers, not characters

```r
data_old = data_old[which(data_old$Canada != ""),]
data_old$Canada = gsub(",", "", data_old$Canada)
order_data = order(data_old$Year)
data_old = data_old[order_data,]
data_old$Year = as.numeric(data_old$Year)
data_old$Canada = as.numeric(data_old$Canada)
head(data_old)
```

```
##    Year  Canada
## 23 1851 2436297
## 22 1861 3229633
## 21 1871 3689257
## 20 1881 4324810
## 19 1891 4833239
## 17 1901 5371315
```

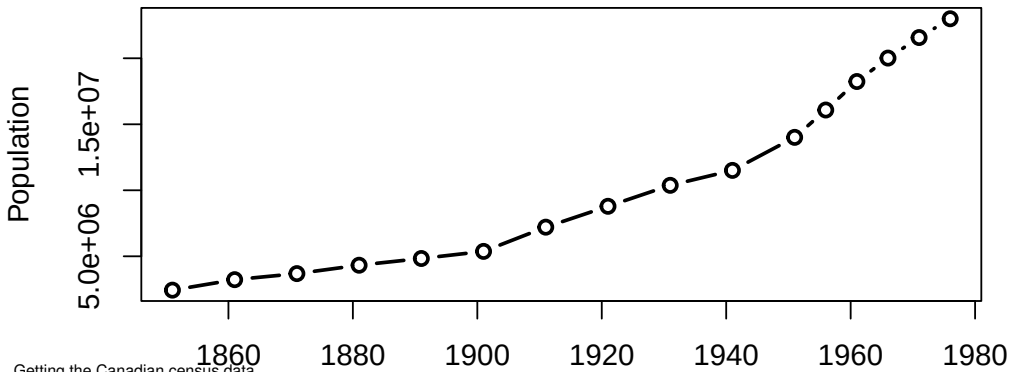
Row numbers are a little weird

```
row.names(data_old) = 1:dim(data_old)[1]
head(data_old)
```

```
##   Year  Canada
## 1 1851 2436297
## 2 1861 3229633
## 3 1871 3689257
## 4 1881 4324810
## 5 1891 4833239
## 6 1901 5371315
```

Well, that looks about right! Let's see what this looks like in a graph

```r
plot(data_old$Year, data_old$Canada,
    type = "b", lwd = 2,
    xlab = "Year", ylab = "Population")
```



But wait, this is only to 1976..!

Looking around, we find another table here

There's a download csv link in there, let us see where this leads us

```
data_new = read.csv("https://www12.statcan.gc.ca/census-recensement/2011/dp-
```

The table is 720KB, so surely there must be more to this than just the population

To get a sense of that, we dump a few random rows

```
data_new[sort(unique(c(1,sample(nrow(data_new), 4)))), ]
```

```
##         GEOGRAPHY.NAME                                CHARACTERISTIC YEAR.S
## 1               Canada                  Population (in thousands)    195
## 1656     New Brunswick Population by single years of age: 96 years    201
## 3022 Trois-Rivi\xe8res Population by single years of age: 19 years    201
## 3601           Ontario Non-official mother tongue: Chinese, n.o.s.    201
## 7758           Calgary  Population by single years of age: 7 years    201
##       TOTAL FLAG_TOTAL
## 1     16081
## 1656    290
## 3022   2065
## 3601 202225
## 7758  14335
```

Flat tables

This is a **flat table**: each row contains a piece of information, the columns describe what this piece of information is about

This has *way more* information than we need: we just want the population of Canada and here we get 9960 rows over 213 characteristics

Also, population is expressed in thousands, so once we selected what we want, we need to multiply by 1,000

Selecting rows

We want the rows where the geography is “Canada” and the characteristic is “Population (in thousands)”

Find indices of rows that satisfy the first criterion and those that satisfy the second; intersecting these two sets of indices, we have selected the rows we want

```
idx_CAN = which(data_new$GEOGRAPHY.NAME == "Canada")
idx_char = which(data_new$CHARACTERISTIC == "Population (in thousands)")
idx_keep = intersect(idx_CAN, idx_char)
```

Let us keep only these rows

```
data_new = data_new[idx_keep,]
head(data_new, n = 8)
```

```
##   GEOGRAPHY.NAME            CHARACTERISTIC YEAR.S. TOTAL FLAG_TOTAL
## 1         Canada Population (in thousands)    1956 16081
## 2         Canada Population (in thousands)    1961 18238
## 3         Canada Population (in thousands)    1966 20015
## 4         Canada Population (in thousands)    1971 21568
## 5         Canada Population (in thousands)    1976 22993
## 6         Canada Population (in thousands)    1981 24343
## 7         Canada Population (in thousands)    1986 25309
## 8         Canada Population (in thousands)    1991 27297
```

We want to concatenate this data frame with the one from earlier

To do this, we need the two data frames to have the same number of columns, same column names and same entry types (notice that `YEAR.S.` in `data_new` is a column of characters)

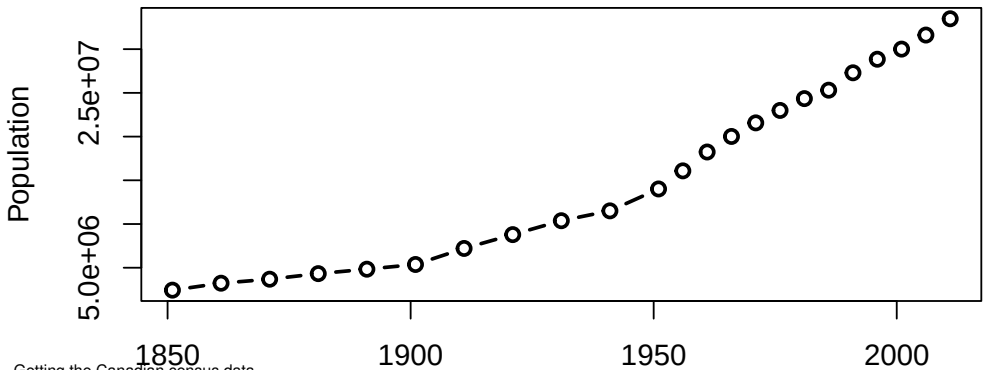
What remains to be done

- Rename the columns in the pruned old data (data_pruned) to `year` and `population`. Personally, I prefer lowercase column names.. and `population` is more informative than `Canada`
- Keep only the relevant columns in `data_new`, rename them accordingly and multiply population by 1,000 there
- Transform year in `data_new` to numbers
- We already have data up to and including 1976 in `data_old`, so get rid of that in `data_new`
- Append the rows of `data_new` to those of `data_pruned`

```r
colnames(data_old) = c("year", "population")
data_new = data_new[,c("YEAR.S.","TOTAL")]
colnames(data_new) = c("year", "population")
data_new$year = as.numeric(data_new$year)
data_new = data_new[which(data_new$year>1976),]
data_new$population = data_new$population*1000

data = rbind(data_old,data_new)
```

```r
plot(data$year, data$population,
    type = "b", lwd = 2,
    xlab = "Year", ylab = "Population")
```



Save the processed data

In case we need the data elsewhere, we save the data to a `csv` file

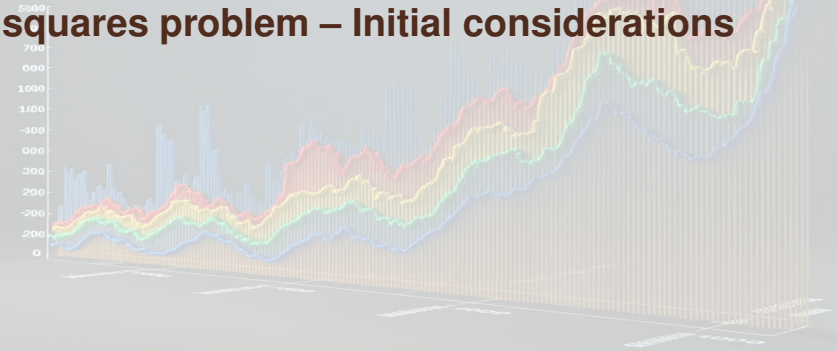
```
write.csv(data, file = "../CODE/Canada_census.csv")
```

Using `readr` saves the data without row numbers (by default), so we can do this instead

```
readr::write_csv(data, file = "../CODE/Canada_census.csv")
```

Getting the Canadian census data

Least squares problem – Initial considerations



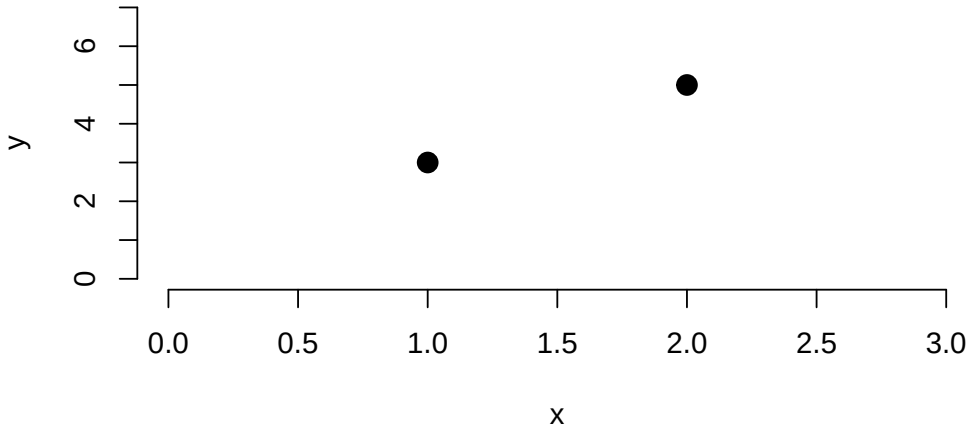
We just collected the census data for Canada

Suppose we want to predict the population of Canada in 20 years given the historical population growth seen in the previous plot. What can we do?

If there were just two points, we could easily "drive" a line through these two points. However, we have much more than two points, so we will use *fitting*, *i.e.*, try to make a curve come as close to possible to the points

We start with a line, giving rise to **linear least squares**

Least squares approximation – A trivial case



We want to find the equation of a line $y = a + bx$ that goes through these two points, i.e., we seek a and b such that

$$3 = a + b$$
$$5 = a + 2b$$

i.e., they satisfy $y = a + bx$ for $(x, y) = (1, 3)$ and $(x, y) = (2, 5)$

This is a linear system with 2 equations and 2 unknowns a and b

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \end{pmatrix}$$

We know from Theorem **??** (Linear algebra in a nutshell) that this system has a unique solution if the matrix

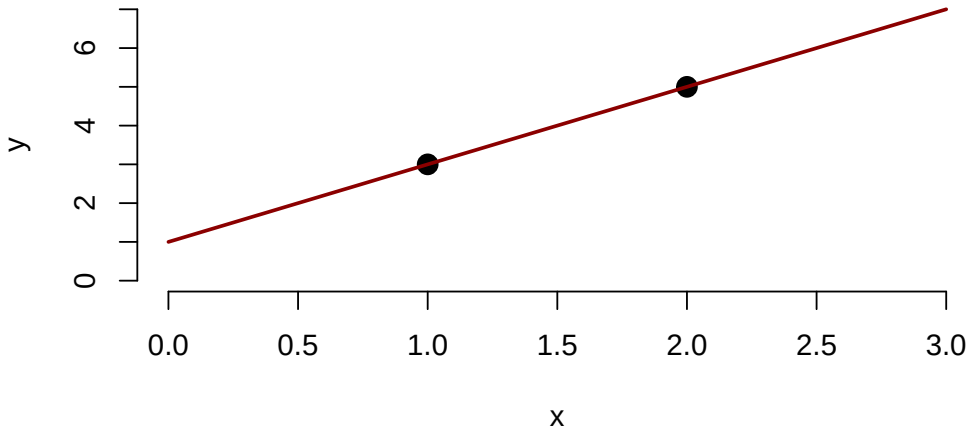
$$M = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$$

is invertible

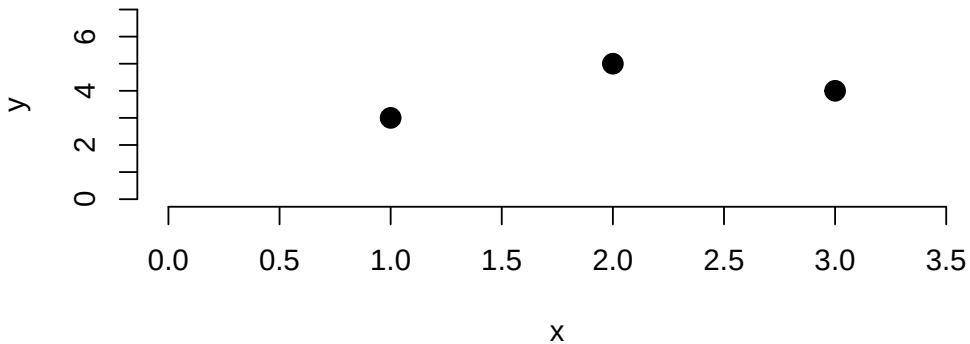
$\det(M) = 1$, so we are good, we'll find a and b easily..

```
A = matrix(c(1,1,1,2), nr = 2, nc = 2, byrow = TRUE)
rhs = matrix(c(3,5), nr = 2, nc =1)
coefs = solve(A,rhs)
coefs
```

```
##      [,1]
## [1,]    1
## [2,]    2
```


y
0
2
4
6
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x

Now let's add another point



These points are clearly not colinear, so there is not one line going through the 3

We end up with an *overdetermined* system

$$
\begin{aligned}
3 &= a + b \\
5 &= a + 2b \\
4 &= a + 3b
\end{aligned}
$$

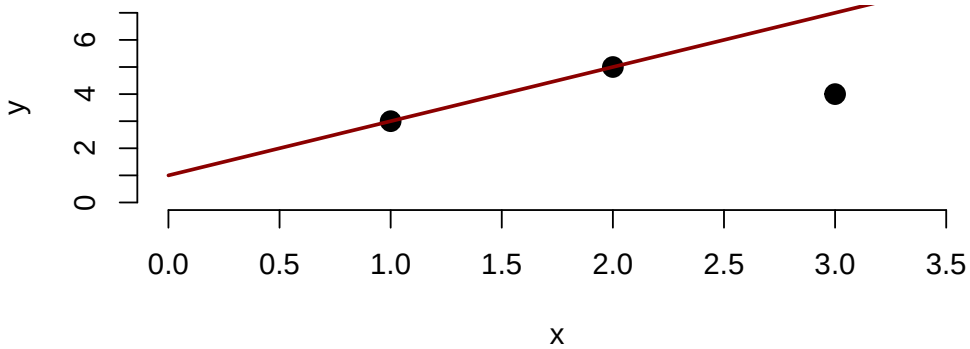
i.e.,

$$
\begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \\ 4 \end{pmatrix}
$$

We have verified visually that the points are not colinear, so this system has no solution

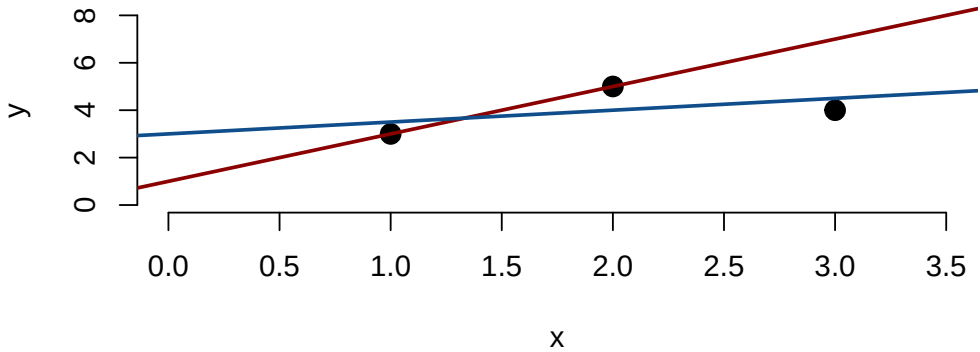
(If you had to do it for good, you consider two vectors stemming from these 3 points and compute the angle between them or check that one is a multiple of the other)

So let us instead try to find the line that comes "closest" to the 3 points



Obviously, not sensational..

Let's eyeball it – Less steep and higher y-intercept



The blue line looks better, but how do we quantify this?

How do we find "how far away" we are?

- We could use projections onto the line (which we know minimises the distance)

- However, this will be a problem if we later decide that rather than a straight line, we want to use something more "funky" like a quadratic or an exponential

Compare, for a given value x, the distance between the true value y and the value of y obtained using the curve (line, here) that we use to fit the data

Let (x_i, y_i) be the data points, i.e., here, $(x_1, y_1) = (1, 3)$, $(x_2, y_2) = (2, 5)$ and $(x_3, y_3) = (3, 4)$

Suppose we use a line with equation $y = a + bx$ and that we pick a value for a and b. Then at x_1,

$$\tilde{y}_1 = a + bx_1 = a + b$$

at x_2

$$\tilde{y}_2 = a + bx_2 = a + 2b$$

and at x_3,

$$\tilde{y}_3 = a + bx_3 = a + 3b$$

Consider x_1, for instance. The error we made by using the line with coefficients (a, b) is $\overrightarrow{(x_1, y_1)(x_1, \tilde{y}_1)}$

For future use, let us create a function for $y = a_0 + a_1 x$

```
my_line = function(x, a_0, a_1){
    return(a_0 + a_1*x)
}
```

Functions are super useful when programming

```
my_line(1,2,3)
```

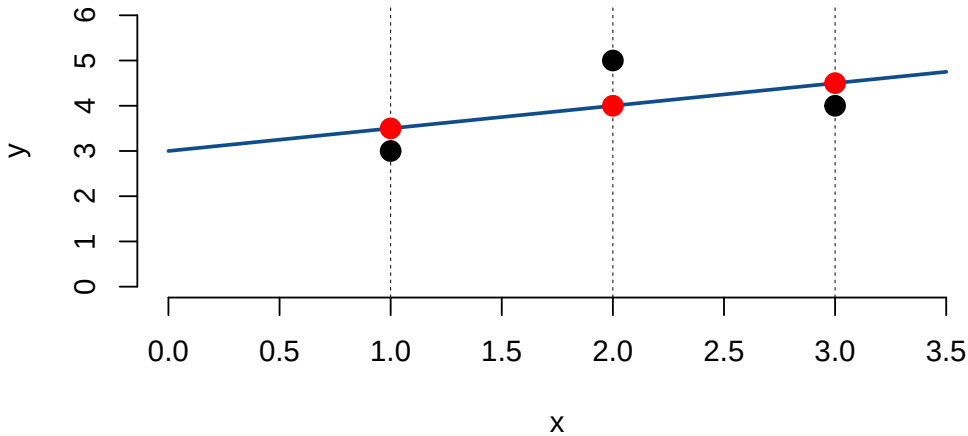
```
## [1] 5
```

```
my_line(a_0 = 2, a_1 = 3, x = 1)
```

```
## [1] 5
```

```
my_line(x = c(1,2,3), a_0 = 2, a_1 = 3)
```

```
## [1]  5  8 11
```


y
0 1 2 3 4 5 6
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5
x

Let us return to the error

$$\overrightarrow{(x_1, y_1)(x_1, \tilde{y}_1)}$$

We have

$$\overrightarrow{(x_1, y_1)(x_1, \tilde{y}_1)} = (x_1 - x_1, y_1 - \tilde{y}_1) = (0, y_1 - \tilde{y}_1)$$

Let us call

$$\varepsilon_1 = y_1 - \tilde{y}_1$$

We can compute ε_2 and ε_3 too. And we can then form the **error vector**

$$\mathbf{e} = (\varepsilon_1, \varepsilon_2, \varepsilon_3)^T$$

The norm of $\mathbf{e}$, $\|\mathbf{e}\|$, then tells us how much error we are making for the choice of (a, b) we are using

The norm of $\|\mathbf{e}\|$ tells us how much error we are making for the choice of (a, b) we are using

So our objective is to find (a, b) such that $\|\mathbf{e}\|$ is minimal

We could use various norms, but the Euclidean norm has some very interesting properties, so we use

$$\|\mathbf{e}\| = \sqrt{\varepsilon_1^2 + \varepsilon_2^2 + \varepsilon_3^2}$$

The linear least squares problem

Given a collection of data points $(x_1, y_1), \ldots, (x_n, y_n)$, find the coefficients a, b of the line $y = a + bx$ such that

$$\|\mathbf{e}\| = \sqrt{\varepsilon_1^2 + \cdots + \varepsilon_n^2} = \sqrt{(y_1 - \tilde{y}_1)^2 + \cdots + (y_n - \tilde{y}_n)^2}$$

is minimal, where $\tilde{y}_i = a + bx_i$, for $i = 1, \ldots, n$

Let us first hack a brute force solution! (For the example we have been using this far)

We have our three points in the list `points`, the function `my_line` that computes the value $\tilde{y}$ given x and a, b, so let us make a new function that, given a, b, computes **e**

We'll also pass the points `points`

A function to compute the error

```
error = function(a_0, a_1, points) {
    y_tilde = my_line(points$x, a_0 = a_0, a_1 = a_1)
    e = points$y - y_tilde
    return(sqrt(sum(e^2)))
}
error(a_0 = 2, a_1 = 3, points)

## [1] 7.874008

error(a_0 = 3, a_1 = 0.5, points)

## [1] 1.224745

error(a_0 = 3.1, a_1 = 0.48, points)

## [1] 1.229471
```

We can't be doing this by hand..!

Let's minimise using something cool – *genetic algorithms*

- Genetic algorithms are a stochastic *optimisation* method. There are other types, e.g., gradient descent (deterministic)
- The idea is to use a mechanism mimicking evolution's drive towards higher fitness
- The function value is its fitness
- We try different genes (here, different values of a, b) and evaluate their fitness.. keep the good ones
- We mutate or crossover genes, throw in new ones, etc.
- We keep doing this until we reach a stopping criterion
- We then return the best gene we found

```r
if (!require("GA", quietly = TRUE)) {
  install.packages("GA")
  library(GA)
}
GA = ga(type = "real-valued",
        fitness = function(x) -error(a_0 = x[1], a_1 = x[2], points),
        suggestions = c(a_0 = 2, a_1 = 3),
        lower = c(-10, -10), upper = c(10, 10),
        popSize = 200, maxiter = 150)
```

GA returns an S4 object of class `ga`

```
GA

## An object of class "ga"
##
## Call:
## ga(type = "real-valued", fitness = function(x) -error(a_0 = x[1],     a_1
##
## Available slots:
##  [1] "call"         "type"         "lower"        "upper"        "nBits"
##  [6] "names"        "popSize"      "iter"         "run"          "maxiter
## [11] "suggestions"  "population"   "elitism"      "pcrossover"   "pmutati
## [16] "optim"        "fitness"      "summary"      "bestSol"      "fitness
## [21] "solution"
```

Access slots using @. For instance, the GA finds the coefficients

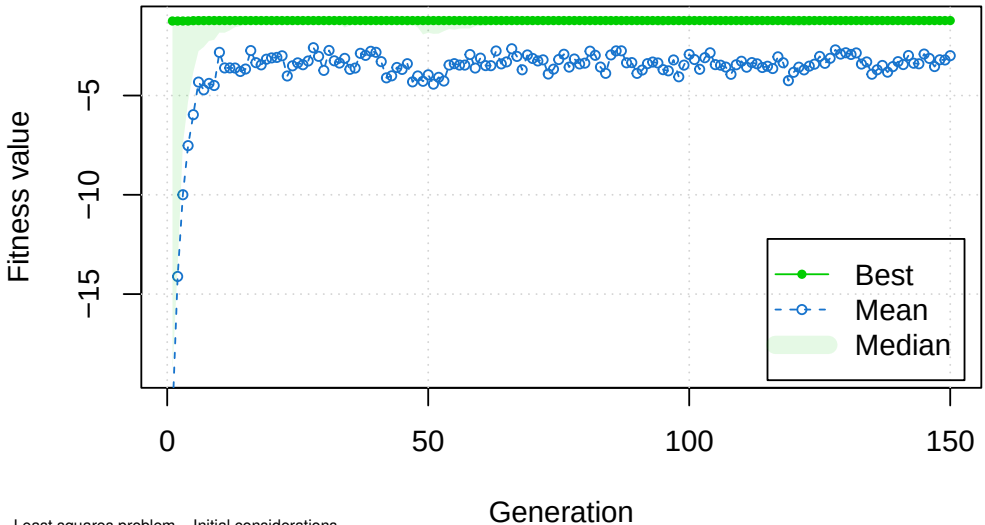
```
GA@solution
```

```
##             x1        x2
## [1,] 2.999889 0.5000423
```

with fitness value (i.e., $\|\boldsymbol{e}\|$)

```
-GA@fitnessValue
```

```
## [1] 1.224745
```


Fitness value
−5
−10
−15
0
50
100
150
Generation
Best
Mean
Median

- Here, however, we do not have to go brute force: we can reason using mathematics

- We now take a little detour on the math side of things, we will come back to code in a while..