Statistical computing MATH10093
Computer lab 5
Solutions

Finn Lindgren

27/2/2019

Summary

In this lab session you will investigate matrix factorisation methods and condition numbers for least squares estimation of statistical linear models. You will not hand in anything, but you should keep your code script file for later use.

1. Initialise: Open `RStudio` and either
   (a) Open the relevant existing project for the lab, if you have one, or
   (b) Create a new project for the folder where you want to store the lab files.

2. **Theory question:**
   In Lecture 5, an approximate bound was derived for the numerical approximation error between a derivative and assymmetric first order differences. Use the same technique to derive the result stated in the lecture for a bound on the approximation error for a symmetric first order difference, $[f(\theta+h)-f(\theta-h)]/(2h)$. Also show that the bound is minimised for the h given in the lecture.

   Solution:
   Triangle inequality:

$$\begin{aligned}
&\left|\frac{\text{comp}\{f(\text{comp}[\theta+h])\}-\text{comp}\{f(\text{comp}[\theta-h])\}}{2h}-f'(\theta)\right| \\
&\quad\le\left|\frac{\text{comp}\{f(\text{comp}[\theta+h])\}-\text{comp}\{f(\text{comp}[\theta-h])\}}{2h}-\frac{f(\theta+h)-f(\theta-h)}{2h}\right| \\
&\qquad+\left|\frac{f(\theta+h)-f(\theta-h)}{2h}-f'(\theta)\right|
\end{aligned}$$

   Using the comp model from the lecture,

$$\begin{aligned}
&\text{comp}\{f(\text{comp}[\theta+h])\}-\text{comp}\{f(\text{comp}[\theta-h])\} \\
&\quad\approx f(\theta+h)+\epsilon_f f(\theta+h)+\epsilon_\theta\theta f'(\theta+t_1 h)-f(\theta-h)-\epsilon'_f f(\theta-h)-\epsilon'_\theta\theta f'(\theta-t'_1 h)
\end{aligned}$$

for some small ϵ_f, ϵ'_f, ϵ_θ, and ϵ'_θ, and some $t_1, t'_1 \in [0,1]$. The first error contribution is then approximately

$$\begin{aligned}
&\left|\frac{\text{comp}\{f(\text{comp}[\theta+h])\} - \text{comp}\{f(\text{comp}[\theta-h])\}}{2h} - \frac{f(\theta+h)-f(\theta-h)}{2h}\right| \\
&\quad\approx \frac{1}{2h}\left|\epsilon_f f(\theta+h) - \epsilon'_f f(\theta-h) + \epsilon_\theta \theta f'(\theta+t_1 h) - \epsilon'_\theta \theta f'(\theta - t'_1 h)\right| \\
&\quad\lesssim \frac{1}{2h}\left(|\epsilon_f||f(\theta+h)| + |\epsilon'_f||f(\theta-h)| + |\epsilon_\theta||\theta||f'(\theta+t_1 h)| + |\epsilon'_\theta||\theta||f'(\theta-t'_1 h)|\right) \\
&\quad\lesssim \frac{1}{2h}\left(2\epsilon_0 L_0 + 2\epsilon_0|\theta|L_1\right),
\end{aligned}$$

which matches the first term of the bound from the lecture, after cancelling the 2s. For the second term, third order Taylor expansion around $h = 0$ gives

$$\begin{aligned}
&\left|\frac{f(\theta+h)-f(\theta-h)}{2h} - f'(\theta)\right| \\
&\quad\approx \left|\frac{f(\theta) + hf'(\theta) + h^2 f''(\theta)/2 + h^3 f'''(\theta)/6 - f(\theta) + hf'(\theta) - h^2 f''(\theta)/2 + h^3 f'''(\theta)/6}{2h} - f'(\theta)\right| \\
&\quad= \left|\frac{2h^3 f'''(\theta)/6}{2h}\right| = \frac{h^2|f'''(\theta)|}{6} \lesssim \frac{h^2 L_3}{6},
\end{aligned}$$

which matches the second term of the error bound.

To find the optimum, take the derivative of the bound with respect to h, set to zero, and solve for h.

Note: In LaTeX (and RMarkdown formulas), $\lesssim$ can be produced by `\lesssim`.

3. Define the following function for computing a vector norm:

```
vec_norm <- function(x) {
  sum(x^2)^0.5
}
```

Try it on a simple vector where you can verify that the result is correct, e.g.

```
c(vec_norm(c(1, 2, 3)), sqrt(1 + 4 + 9))

## [1] 3.741657 3.741657
```

4. Check if the `microbenchmark` package is available with `library("microbenchmark")`. If it's not installed, use `install.packages("microbenchmark")` to install it. Do the same check for the package `xtable`.

   Take a look at the computational speed of matrix operations, by running the following code:

```
n <- 100000
m <- 100
b <- rnorm(n)
A <- matrix(rnorm(n * m), n, m)

## Two R expressions for the same theoretical expression:
microbenchmark::microbenchmark(t(A) %*% b,
                               t(b %*% A))

# Check that the results are the same:
At_b <- t(A) %*% b
bA_t <- t(b %*% A)
vec_norm(At_b - bA_t) ## Should produce the same result, with difference norm zero.
```

Which method for computing $\boldsymbol{A}^\top \boldsymbol{y}$ is faster? Can you guess why?

In a RMarkdown or knitr document, the following, with code chunk option `results="asis"` can be used to generate a nice tabular layout of the timing benchmarks:

```
bm <- microbenchmark::microbenchmark(t(A) %*% b,
                                     t(b %*% A))
xtable::xtable(summary(bm))
```

	expr	min	lq	mean	median	uq	max	neval
1	t(A) %*% b	61.56	67.83	72.75	71.15	74.58	125.86	100.00
2	t(b %*% A)	18.43	19.47	21.23	20.16	22.64	33.59	100.00

For $\boldsymbol{ABb}$, the order in which the multiplication is done is important:

```
m <- 2000
n <- 2000
p <- 2000
A <- matrix(rnorm(m * n), m, n)
B <- matrix(rnorm(n * p), n, p)
b <- rnorm(p)
microbenchmark::microbenchmark(ABb1 <- A %*% B %*% b,
                               ABb2 <- (A %*% B) %*% b,
                               ABb3 <- A %*% (B %*% b),
                               times = 1)


## Unit: milliseconds
##                     expr        min         lq       mean     median
##    ABb1 <- A %*% B %*% b 5589.68483 5589.68483 5589.68483 5589.68483
##  ABb2 <- (A %*% B) %*% b 5618.84965 5618.84965 5618.84965 5618.84965
##  ABb3 <- A %*% (B %*% b)   10.44177   10.44177   10.44177   10.44177
##          uq        max neval
```

```
##  5589.68483 5589.68483     1
##  5618.84965 5618.84965     1
##    10.44177   10.44177     1
```

```
## Should produce the same result,
## with difference norms close to zero.
c(vec_norm(ABb1 - ABb2),
  vec_norm(ABb1 - ABb3),
  vec_norm(ABb2 - ABb3))
```

```
## [1] 0.000000e+00 2.691768e-10 2.691768e-10
```

```
c(vec_norm(ABb1 - ABb2),
  vec_norm(ABb1 - ABb3),
  vec_norm(ABb2 - ABb3)) / .Machine$double.eps
```

```
## [1]       0 1212264 1212264
```

Which method is fastest? Why? Are the computed results the same?
Answers:
Transposing a vector doesn't require any copying. Transposing a matrix requires copying the values to new locations.
One matrix-matrix multiplication is very expensive compared with matrix-vector multiplication. The results are numerically very close, and one can likely express bounds for them in terms of `.Machine$double.eps` and m, n, p.

5. You'll now investigate some numerical issues when doing numerical least squares estimation.

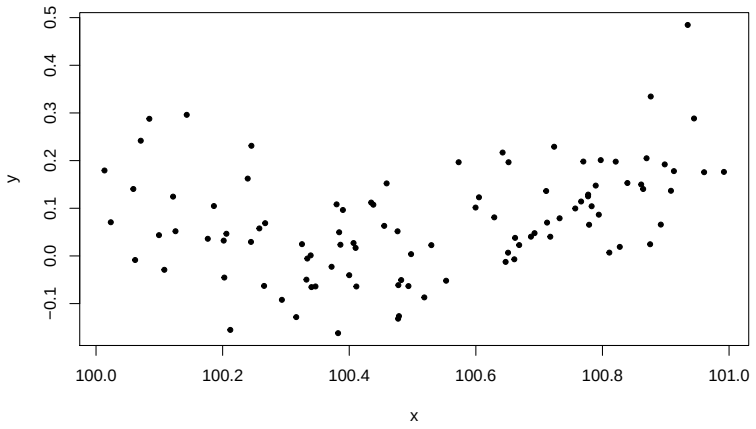
   (a) Run the following code that generates synthetic observations from a linear model

   $$y_i = \frac{x_i - 100.5}{5} + (x_i - 100.5)^2 + e_i,$$

   where $e_i \sim \mathrm{N}(0, \sigma_e = 0.1)$, independent, $i = 1, \ldots, n = 100$.

```
## Set the "seed" for the random number sequences, so we can
## reproduce exactly the same random numbers every time we run the code
set.seed(1)
## Simulate the data
# x: 100 random numbers between 100 and 101
data <- data.frame(x = 100 + sort(runif(100)))
# y: random variation around a quadrati function:
data <- within(data,
               y <- (x - 100.5) / 5 + (x - 100.5)^2 +
                 rnorm(nrow(data), sd = 0.1))
```

```r
## Let's try a nice way of plotting data we have stored in a data frame:
plot(y ~ x, data, pch = 20)
## See ?plot.formula for more information; this is an example
## of R calling a special function for special inputs, in this
## the formula "y ~ x".
```



Note: the `data$y` assignment was broken over two lines, with the `+` at the end of the first line. If we had moved it to the beginning of the second line, R would have terminated the expression at the end of the first line, since that code forms a complete R expression! To avoid surprises in such situations, it's safer to wrap the expression in brackets, like `x <- (1 + 2)`, since such an expression isn't a complete and valid expression until the final bracket is reached, even if there are line breaks inbetween.

(b) What are the true values of β_0, β_1, and β_2 in the standardised model formulation

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + e_i?$$

Hint: Identify the values by matching the terms when expanding the expression in 5(a).

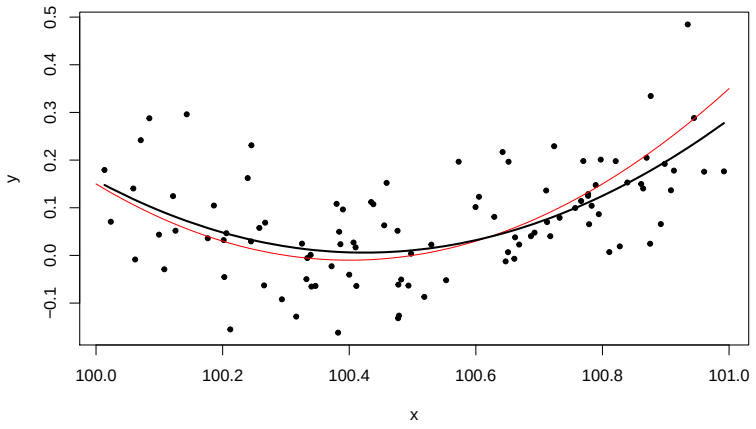
Solution:

$$\beta_0 = -\frac{100.5}{5} + 100.5^2 = 10080.15$$
$$\beta_1 = \frac{1}{5} - 201 = -200.8$$
$$\beta_2 = 1$$

```
beta_true <- c(10080.15, -200.8, 1)
```

Use these β-values to plot the quadratic true model predictor function as a function of $x \in [100, 101]$, together with the observations, and the estimate provided by `lm()`:

```
## beta_true <- c(...)
plot(y ~ x, data, pch = 20)
## See ?curve
curve(beta_true[1] + beta_true[2] * x + beta_true[3] * x^2,
      100, 101, add = TRUE, col = 2)

## lm manages to estimate the regression:
mod1 <- lm(y ~ x + I(x^2), data = data)
## Add the fitted curve:
lines(data$x, fitted(mod1), lwd = 2)
```



(c) Use the `model.matrix(mod1)` function to extract the $\boldsymbol{X}$ matrix for the vector formulation of the model, $\boldsymbol{y} = \boldsymbol{X} + \boldsymbol{e}$. (See `?model.matrix` for more information). Then use the direct *normal equations* solve shown in Lecture 5 to estimate the β parameters. Does it work?

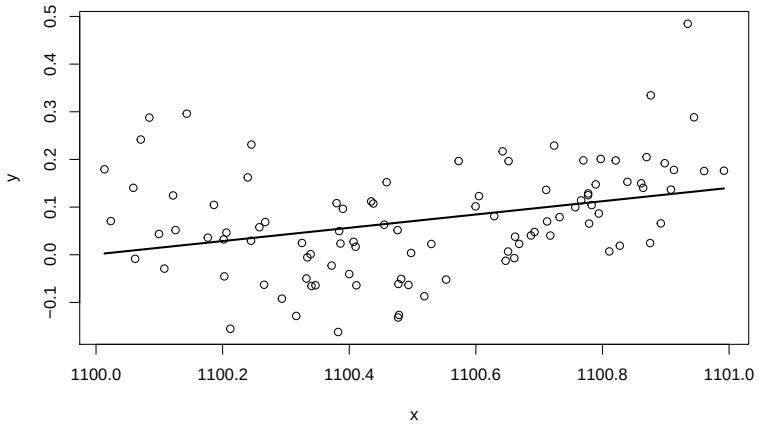
```
## direct solution of the normal equations doesn't work
X <- model.matrix(mod1)
beta.hat <- solve(t(X) %*% X, t(X) %*% data$y)

## Error in solve.default(t(X) %*% X, t(X) %*% data$y):  system is
computationally singular:  reciprocal condition number = 3.9864e-19
```

(d) What if the whole model was shifted to the right, so that we were given $x+1000$ instead of the original x-values? The model expression would still be able to represent the same quadratic expression but with different β values. Create a new data set:

```
data2 <- data
data2$x <- data$x + 1000
```

Estimate the model using `lm()` for this new data set. Does it work?

```
mod2 <- lm(y ~ x + I(x^2), data = data2)
plot(y ~ x, data2)
lines(data2$x, fitted(mod2), lwd = 2)
## lm fails to fit the correct curve, without warning!
```



(e) Define a function `cond_nr()` taking a matrix `X` as input and returning the condition number, computed with the help of `svd` (see Lecture 5 for a code example for the condition number).

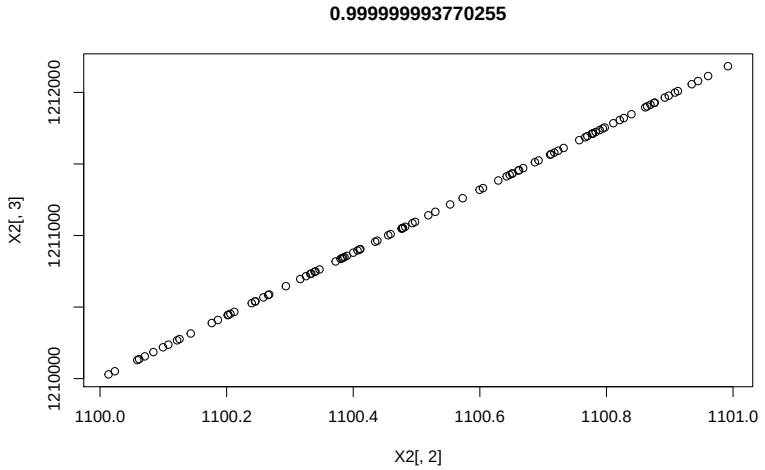
```
cond_nr <- function(X) {
  d <- svd(X)$d
  max(d) / min(d)
}
```

(f) With the help of `model.matrix`, examine the condition numbers of the model matrices in the two cases from the previous tasks. `lm()` computes the fit using the QR decomposition approach, not by direct solution of the normal equations. Why was the second `lm()` fit so bad?

```
cond_nr(X) # manageable condition number

## [1] 1560769713

X2 <- model.matrix(mod2)
cond_nr(X2) # very large condition number,

## [1] 2.242481e+13

            # but not large enough to trigger a warning!
```

(g) Plot the second and third columns of the model matrix against each other, and use `cor` to examine their correlation (see `?cor`). How does this explain the large condition number?

```
plot(X2[,2], X2[,3], main=cor(X2[,2], X2[,3])) ## Very close to co-linear
```



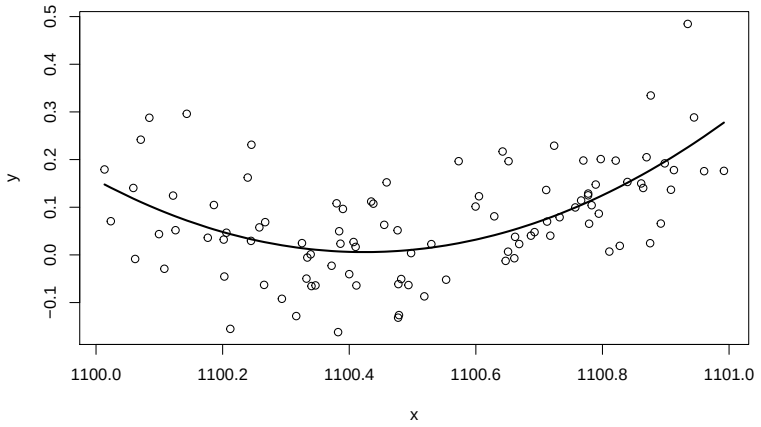
Answer:
In both cases the columns are virtually co-linear, which leads to a very small d_3 value, and therefore a large condition number.

(h) Since the linear model says simply that the expected value vector $\mathrm{E}(\boldsymbol{y})$ lies in the space spanned by the columns of $\boldsymbol{X}$, one possibility is to attempt to arrive at a better conditioned $\boldsymbol{X}$ by linear rescaling and/or recombination of its columns. This is always equivalent to a linear re-parameterization.

Try this on the model matrix of the model that causes `lm` to fail. In particular, for each column (except the intercept column) subtract the column mean (e.g., try the `sweep` and `colMeans` function, see example code below). Then divide each column (except the intercept column) by its standard deviation. Find the condition number of the new model matrix. Fit the model with this model matrix using something like `mods <- lm(y ~ Xs - 1, data = data)` where `Xs` is the re-scaled model matrix. (This will only use the `y` part of `data`.) Produce a plot that confirms that the resulting fit is sensible now.

```
## partial example code:
Xs <- X2
mean_vec <- colMeans(Xs[, 2:3])
sd_vec <- c(sd(Xs[, 2]), sd(Xs[, 3]))
Xs[, 2:3] <- sweep(Xs[, 2:3], 2, mean_vec, FUN="-")
Xs[, 2:3] <- ...
```

```
## Solution code:
Xs <- X2
mean_vec <- colMeans(Xs[, 2:3])
sd_vec <- c(sd(Xs[, 2]), sd(Xs[, 3]))
Xs[, 2:3] <- sweep(Xs[, 2:3], 2, mean_vec, FUN="-")
Xs[, 2:3] <- sweep(Xs[, 2:3], 2, sd_vec, FUN="/")

mods <- lm(y ~ Xs - 1, data = data)
plot(y ~ x, data2)
lines(data2$x, fitted(mods), lwd=2)
```



Note: If the data is split into estimation and test sets, the same transformation must be applied to the test data, using the scaling parameters derived from the observation data. Therefore the `scale()` function isn't very usful for this.

(i) Compute the condition numbers for `X2` and `Xs`. Also compute the correlation between column 2 and 3 of `Xs`.

```
cond_nr(X2)

## [1] 2.242481e+13

cond_nr(Xs)

## [1] 17917.6

cor(Xs[, 2], Xs[, 3])

## [1] 1
```

Did subtracting and rescaling help?

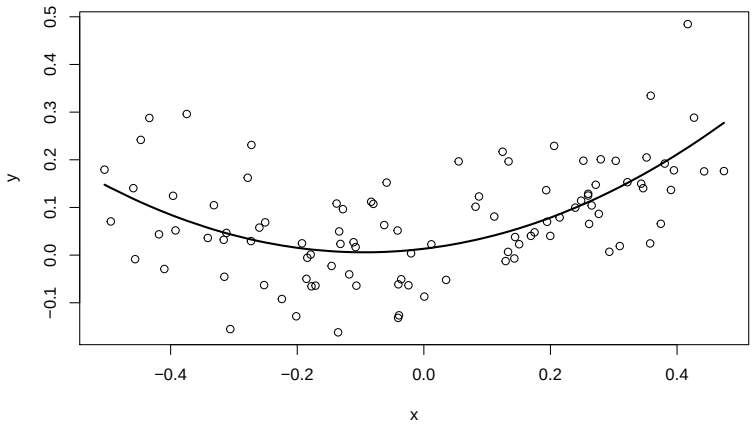
Answer: Yes, the condition number was significantly reduced. The

correlation between the columns is however still very close to 1, meaning that the matrix is still close to singular.

(j) An alternative fix is to subtract the mean `x` value from the original `x` vector before defining the quadratic model. Try this and see what happens to the condition number and column correlations of the model matrix.

```
## Data setup:
data3 <- data2
data3$x <- data3$x - mean(data3$x)
```

```
## Solution code:
mod3 <- lm(y ~ x + I(x^2), data = data3)
plot(y ~ x, data3)
lines(data3$x, fitted(mod3), lwd=2)

X3 <- model.matrix(mod3)
cond_nr(X3)

## [1] 15.36915

cor(X3[, 2], X3[, 3])

## [1] -0.09094755
```



Different methods for modifying the inputs and model definitions require different calculations for compensating in the model parameters, and has to be figured out for each case separately. The most common effect is to change the interpretation of the β_0 parameters.