Statistical computing MATH10093
Computer lab 5

Finn Lindgren

26/2/2020

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.

   In both cases, remember to access the project and files via the `M:` drive, and *not*, e.g., `Desktop` or `Documents`.

2. Start a new Rmarkdown document for the lab, and add a setup code chunk:

```
suppressPackageStartupMessages(library(tidyverse))
library(xtable)
library(microbenchmark)
```

   If any of those libraries aren't installed, use `install.packages()` to install them.

3. **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.

4. 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
```

5. We will take a look at the computational speed of matrix operations. First construct A vector $\boldsymbol{b}$ and matrix $\boldsymbol{A}$:

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

Let's check the speed difference between $\boldsymbol{A}^\top \boldsymbol{b}$ and $(\boldsymbol{b}^\top \boldsymbol{A})^\top$ (convince yourself that the two expressions would be equivalent in exact arithmetic, and possibly also in floating point arithmetic):

```
# 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.

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

Note that when `b` is a vector (as opposed to a single-column matrix) it is interpreted as either a column vector or a row vector, depending on the operation. Check what happens with `as.matrix(b)` and `t(b)`. 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(t(A) %*% b, t(b %*% A), t(t(b) %*% A))
print(xtable(summary(bm),
             caption = paste("Unit:",
                             attr(summary(bm), "unit"))))
```

You can define a simple wrapper function for shorter code, such as this one:

```
# Usage: microbenchmark_table(bm)
#   where bm is a microbenchmark output object.
microbenchmark_table <- function(bm) {
  print(xtable(summary(bm),
               caption = paste("Unit:",
                               attr(summary(bm), "unit"))))

}
```

6. For $\boldsymbol{ABb}$, the order in which matrix&vector 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)
```

```
bm <- microbenchmark(ABb1 <- A %*% B %*% b,
                     ABb2 <- (A %*% B) %*% b,
                     ABb3 <- A %*% (B %*% b),
                     times = 1)
microbenchmark_table(bm)
```

   The three versions should produce very similar numerical results:

```
print(xtable(
  cbind(Difference =
          c("|1-2|" = vec_norm(ABb1 - ABb2),
            "|1-3|" = vec_norm(ABb1 - ABb3),
            "|2-3|" = vec_norm(ABb2 - ABb3)),
        DifferenceRelEps =
          c("|1-2|" = vec_norm(ABb1 - ABb2),
            "|1-3|" = vec_norm(ABb1 - ABb3),
            "|2-3|" = vec_norm(ABb2 - ABb3)) / .Machine$double.eps),
  digits = c(0, -2, 1) # See ?xtable for explanation.
))
```

   Which method is fastest? Why? Are the computed results the same?

7. For a square matrix $\boldsymbol{A}$ of size 2000×2000, compare the cost of
   `solve(A) %*% b` (Get the matrix inverse $\boldsymbol{A}^{-1}$, then multiply with $\boldsymbol{b}$)
   against the cost of
   `solve(A, b)` (Solve the linear system $\boldsymbol{Au} = \boldsymbol{b}$).

8. 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 quadratic function:
data <- data %>%
  mutate(y = (x - 100.5) / 5 + (x - 100.5)^2 + rnorm(n(), sd = 0.1))
```

Let's plot the data we have stored in the `data.frame`:

```
ggplot(data) + geom_point(aes(x, y))
```

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

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

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

```
beta_true <- c(?, ?, ?)
```

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()`. Since the observed x values are relatively dense, we don't necessarily need a special plot sequence of x values, but can reuse the observed values.

```
data <- data %>%
  mutate(mu_true = beta_true[1] + beta_true[2] * x + beta_true[3] * x^2)
pl <- ggplot(data) +
  geom_point(aes(x, y)) +
  geom_line(aes(x, mu_true), col ="red")
pl
```

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

(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?

(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 %>%
  mutate(x = x + 1000)
```

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

(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).

(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?

(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?

(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 `mod3 <- lm(y ~ x1 + x2 + x3 - 1, data = data3)` (See partial code below.)

Produce a plot that confirms that the resulting fit is sensible now.

```
## partial example code:
mean_vec <- colMeans(X2[, 2:3])
sd_vec <- c(sd(X2[, 2]), sd(X2[, 3]))
X3 <- cbind(X2[, 1],
            sweep(X2[, 2:3], 2, mean_vec, FUN = "-"))
X3[, 2:3] <- sweep(???)
data3 <- data2 %>%
  mutate(x1 = X3[, 1], x2 = X3[, 2], x3 = X3[, 3])
```

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 `X3`. Also compute the correlation between column 2 and 3 of `X3` (subtract 1 to see how close to 1 it is).

Did subtracting and rescaling help?

(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:
data4 <- data2 %>%
  mutate(x_shifted = x - mean(x))
```

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.