Group 21 Lab 2 Report

Stefano Toffol (steto820), Mim Kemal Tekin (mimte666)

08 March, 2019

Question 1: Optimizing a Model Parameter

The file mortality_rate.csv contains information about mortality rates of the fruit flies during a certain period.

Task 1.1

- *Add one more variable LMR to the data which is the natural algorithm of Rate.*
- *Divide the data into training and test sets.*

```
################################### TASK 1.1 ###################################

library(ggplot2)

df_mortality = read.csv2("../datasets/mortality_rate.csv")

# calculate natural log of Rate variable
df_mortality$LMR = log(df_mortality$Rate)

# split data
n = dim(df_mortality)[1]
set.seed(123456)
id = sample(1:n, floor(n*0.5))
train = df_mortality[id, ]
test = df_mortality[-id, ]
```

Task 1.2

- *Create a function called myMSE() which takes parameters λ and list pars containing vector X, Y, Xtest, Ytest fits a LOESS model with response Y and predictor X using loess() function with penalty λ (parameter enp.target in loess()) and then predicts the model for Xtest. The function should compute the predictive MSE, print it and return as a result.*

```
################################### TASK 1.2 ###################################

# lambda <numeric> : penalization parameter
# pars <list> : contain X, Y, Xtest, Ytest
myMSE = function(lambda, pars){
  set.seed(123456)
  loess_model = loess(formula = Y ~ X, data = pars, enp.target = lambda)
  preds = predict(loess_model, pars$Xtest)
  mse = mean((pars$Ytest - preds)^2)
  counter <<- counter + 1
  message(paste("Lambda =", lambda))
  message(paste("MSE =", mse))
  message("----------")
  return(mse)
}
```

Task 1.3

- *Use a simple approach: use function myMSE(), training and test sets with response LMR and predictor Day and the following λ values to estimate the predictive MSE values:* $\lambda = 0.1, 0.2, ..., 40$

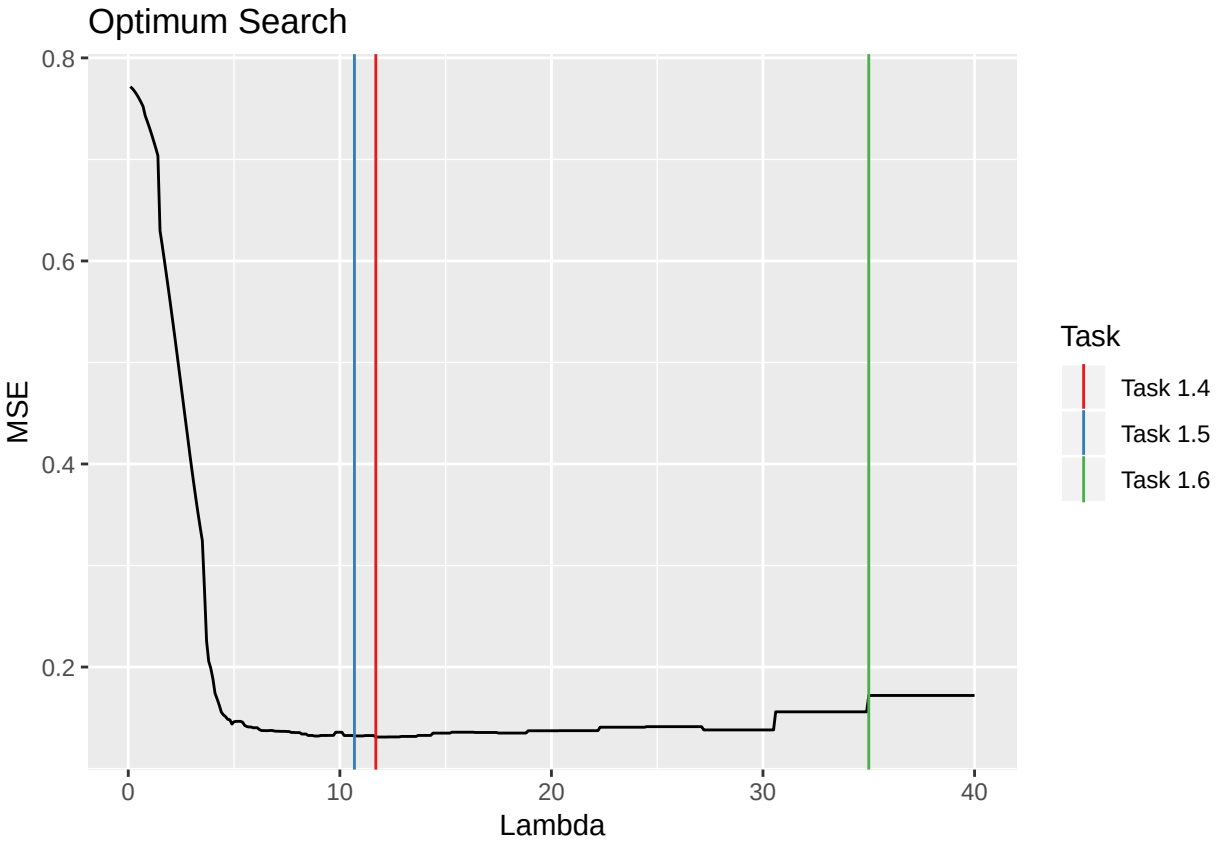
```
################################### TASK 1.3 ###################################

pars = list()
pars$X = train$Day
pars$Y = train$LMR
pars$Xtest = test$Day
pars$Ytest = test$LMR

lambda = seq(0.1, 40, 0.1)
counter = 0
mse = c()
for(i in lambda){
  mse = c(mse, myMSE(i, pars))
}
```

Task 1.4

- *Create a plot of the MSE values versus λ and comment on which λ value is optimal. How many evaluations of myMSE() were required (read ?optimize) to find this value?*




```
## [1] "Minimum Lambda = 11.7"
```

```
## [1] "Evaluation Count = 117"
```

In the plot, when lambda increases, mse error decreases exponentially until 5. After that point we can see a general increment and after around 20 of lambda, mse starts to still constant for some lambda values. But from plot we can say we have at least 3 deep value.
myMSE function found the optimum lambda, equal to 11.7, after 400 evaluation. The approach used is actually a brute-force over the discretized lambda vector, which is composed by 400 values: finding the minimum of that vector naturally requires checking every single observation.

Task 1.5

- *Use optimize() function for the same purpose, specify range for search [0.1,40] and the accuracy 0.01. Have the function managed to find the optimal MSE value? How many myMSE() function evaluations were required? Compare to step 4.*

```
## [1] "Minimum Lambda = 10.6936106512018"
```

```
## [1] "Evaluation Count = 18"
```

In this task we used optimize function to perform a Golden-Section Search algorithm. We specify whole interval that we used in 4th task. This algorithm found the optimal lambda as 10.6936 in the 18th evaluation (iteration). If we compare this value with output of task 1.4, it is not the optimal lambda. It is less than optimal lambda, but it is also close to optimal. This function stuck on a local minimum which is visible before the x-intercept of Task-1. In the plot of 4th task we mention about it is clear to see we have many deep points, so Golden-Section Search can be misleading in this way.

Taask 1.6

Use optim() function and BFGS method with starting point $\lambda = 35$ to find the optimal λ value. How many myMSE() function evaluations were required (read ?optim)? Compare the results you obtained with the results from step 5 and make conclusions.

```
## [1] "Minimum Lambda = 35"
```

```
## [1] "Evaluation Count = 3"
```

We can see we have only 3 evaluations were necessary to reach convergence and the optimal value is 35 according to this algorithm. In the function plot, we can see many deep points as we said before. If we compare this with the result of Task 1.5, we can see this value is much bigger. This algorithm worked in a really short time, but is not close to reality. This happens because the algorithm is based on Gradient-Search and in Gradient-Search algorithms, starting point is important in order to not capture a local optima. Additionally, we can say that it is not a suitable function to search with gradient search. We can see in the plot of task 1.4 that function has many flat areas (especially around starting point) and Gradient-Search algorithms are not good to run on this kind of functions, not moving much from the starting point.

Question 2

First of all we load the data with the following script:

The data loaded come from a normal distribution on unknown μ and σ^2 parameters. To find the ML estimates of them ($\hat{\mu}_{ML}$ and $\hat{\sigma}^2_{ML}$) we first need to define the log-likelihood function. For a univariate normal distribution the log-likelihood function is the following:

$$l(\mu, \sigma^2; \overrightarrow{x}) = -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln(\sigma^2) - \frac{1}{2\sigma^2}\sum_{j=1}^{n}(x_j - \mu)^2$$

In code can be translated as following:

```
# Log-likelihood
log_lik_norm <- function(x, par) {

  mu <- par[1]
  sigma <- par[2]
  n <- length(x)
  llik <- -(n/2)*log(2*pi) - (n/2)*log(sigma^2) - sum((x-mu)^2)/(2*sigma^2)
  return(llik)

}
```

A first analytical solution can be achieved setting the derivative to zero. It will result in the above mentioned *Maximum Likelihood* (*ML*) estimates of the parameters. Here are the equations needed to be solved (the intermediate steps are skipped):

$$\frac{\partial l(\mu, \sigma^2; \overrightarrow{x})}{\partial \mu} = \frac{1}{\sigma^2}\left(\sum_{j=1}^{n} x_j - n\mu\right) = 0$$
$$\frac{\partial l(\mu, \sigma^2; \overrightarrow{x})}{\partial \sigma^2} = \frac{1}{\sigma}\left[\frac{1}{\sigma^2}\sum_{j=1}^{n}(x_j - \mu)^2 - n\right] = 0$$

From then on is easy to isolate each single parameter and solve the equations. The *ML* are therefore equal to:

$$\hat{\mu}_{ML} = \frac{1}{n}\sum_{j=1}^{n} x_j$$
$$\hat{\sigma}^2_{ML} = \frac{1}{n}\sum_{j=1}^{n}(x_j - \hat{\mu}_{ML})^2$$

In other words, the analytical solution will lead to the sample mean and to the *unadjusted* sample variance for the estimates of, respectively, $\hat{\mu}_{ML}$ and $\hat{\sigma}^2_{ML}$. For our sample they are equal to 1.27553 and 4.02394.

A numeric solution can instead be achieved using the function `optim()`. Several methods to solve this kind of problems have been implemented. We are going to use the *conjugate gradients* (*CG*) method and the so-called *quasi-Newton* algorithm, called *BFGS* inside the function. We will use $(\mu, \sigma^2) = (0, 1)$ as starting point for the optimization. However the function `optim()` is designed to find the minimum of a function,

therefore we will need to give it the negative log-likelihood.
There is also the possibility to manually specify the gradient function of the (negative) Gaussian log-likelihood. We will also try this solution and compare the results with the default settings.

We will of course NOT maximize the ordinary log-likelihood: this quantity is in fact given by a multiplication of individual contributions to the likelihood, which are all values between 0 and 1 (probabilities), potentially resulting in extremely small numbers, really close to 0. No matter how powerful a computer is, any machine will struggle in representing such small numbers. It may even approximate it to zero itself due to rounding issues. Using the logarithm will instead transform all the multiplication of the individual density function ($\prod_{i=1}^{n} p(\theta; x_i)$) to an easier-to-handle sum ($\sum_{i=1}^{n} \log(p(\theta; x_i))$) of much greater quantities (in terms of absolute values). In fact any probability will be transformed in a negative number, where values originally close to zero will instead become much greater (on an absolute scale). The summation of such numbers corresponds to a value < -1, giving less troubles to its representation in a computer, improving the precision of the computations.

The following is the code that has been used to perform the task:

```
# Negative log-likelihood
neg_log_lik_norm <- function(x, par) -log_lik_norm(x, par)
# Gradient of negative log-likelihood
gradient_neg_norm <- function(x, par) {

  n <- length(x)
  mu <- par[1]
  sigma <- par[2]
  gradient <- c((sum(x)-n*mu)/(sigma^2),  # To respect to mu
                (sum((x-mu)^2)/sigma^2-n)/sigma)  # To respect to sigma
  return(-gradient)

}
# New numeric estimates of the parameters
nllik_cg <- optim(par = c(0, 1), neg_log_lik_norm, x = data, method = "CG")
nllik_bfgs <- optim(par = c(0, 1), neg_log_lik_norm, x = data, method = "BFGS")
grad_cg <- optim(par = c(0, 1), neg_log_lik_norm, gr = gradient_neg_norm,
                 x = data, method = "CG")
grad_bfgs <- optim(par = c(0, 1), neg_log_lik_norm, gr = gradient_neg_norm,
                   x = data, method = "BFGS")
```

In general all results are converging and brings almost identical results, extremely close to the *ML* estimates (Table 1). The biggest differences between the various algorithms and the specification of the gradient are the number of times the function itself and the gradients are evaluated. As we can see, the *CG* algorithm is more expensive in terms of computational resources if compared to the *BFGS* one. In particular, if the gradient is not specified, more than 200 evaluations of the function are needed before reaching convergence with the *CG* method, while only 53 if the vector of the first derivatives is given. The *BFGS* seems preferable in any case.

It's instead hard to state whether or not to specify the gradient: setting the `gr` argument leads to 3 more evaluations of the function before reaching convergence, but the final result for $\hat{\mu}$ is closer to the *ML* estimations when the gradient is specified. As a general rule, we would suggest to specify the gradient when this is possible.

Table 1: Comparison table of the different algorithms.

Algorithm	Gradient	$\hat{\mu}$	$\hat{\sigma}$	$-l(\mu, \sigma; x)$	Function evaluations	Gradient evaluations
CG	Non specified	1.275528	2.005977	211.5069	210	35
	Specified	1.275528	2.005977	211.5069	53	17
BFGS	Non specified	1.275527	2.005977	211.5069	37	15
	Specified	1.275528	2.005977	211.5069	40	15

```r
# Code for the in-line results (Q2):

# ML estimates
round(mean(data), 5) # mu
round(var(data)*((n-1)/n), 5) # var
```

Appendix

```r
knitr::opts_chunk$set(echo = TRUE, message=F, echo=F)

################################### TASK 1.1 ###################################

library(ggplot2)

df_mortality = read.csv2("../datasets/mortality_rate.csv")

# calculate natural log of Rate variable
df_mortality$LMR = log(df_mortality$Rate)

# split data
n = dim(df_mortality)[1]
set.seed(123456)
id = sample(1:n, floor(n*0.5))
train = df_mortality[id, ]
test = df_mortality[-id, ]

################################### TASK 1.2 ###################################

# lambda <numeric> : penalization parameter
# pars <list> : contain X, Y, Xtest, Ytest
myMSE = function(lambda, pars){
  set.seed(123456)
  loess_model = loess(formula = Y ~ X, data = pars, enp.target = lambda)
  preds = predict(loess_model, pars$Xtest)
  mse = mean((pars$Ytest - preds)^2)
  counter <<- counter + 1
  message(paste("Lambda =", lambda))
  message(paste("MSE =", mse))
  message("----------")
  return(mse)
}

################################### TASK 1.3 ###################################

pars = list()
pars$X = train$Day
pars$Y = train$LMR
pars$Xtest = test$Day
pars$Ytest = test$LMR

lambda = seq(0.1, 40, 0.1)
counter = 0
mse = c()
for(i in lambda){
  mse = c(mse, myMSE(i, pars))
}

################################### TASK 1.4 ###################################
```

```
counter = 0
optimum_mses = which(mse==min(mse))
optimum_lambdas = lambda[optimum_mses]

# plot lambda vs mse
xintercepts = data.frame(x = c(11.7, 10.69, 35),
                         task = c("Task 1.4", "Task 1.5", "Task 1.6"))
ggplot() +
  geom_line(aes(x = lambda, y = mse)) +
  geom_vline(aes(xintercept = x, color=task), data = xintercepts) +
  labs(title = "Optimum Search", x = "Lambda", y = "MSE", color = "Task") +
  scale_color_brewer(palette = "Set1")

print(paste("Minimum Lambda =", optimum_lambdas[1]))
print(paste("Evaluation Count =", optimum_mses[1]))

################################### TASK 1.5 ###################################

set.seed(123456)
# We used global counter to count how many iteration is done by optimizer
counter = 0
obj_optimize = optimize(myMSE, c(0.1, 40), tol = 0.01, pars = pars)

print(paste("Minimum Lambda =", obj_optimize$minimum))
print(paste("Evaluation Count =", counter))

################################### TASK 1.6 ###################################

counter = 0
obj_optim = optim(par = 35, fn=myMSE, method = "BFGS", pars = pars)

print(paste("Minimum Lambda =", obj_optim$par))
print(paste("Evaluation Count =", counter))


# ------------------------------------------------------------------------------
# A2
# ------------------------------------------------------------------------------


load("../datasets/data.RData")


# Log-likelihood
log_lik_norm <- function(x, par) {

  mu <- par[1]
  sigma <- par[2]
  n <- length(x)
  llik <- -(n/2)*log(2*pi) - (n/2)*log(sigma^2) - sum((x-mu)^2)/(2*sigma^2)
  return(llik)

}
```

```r
n <- length(data)


# Negative log-likelihood
neg_log_lik_norm <- function(x, par) -log_lik_norm(x, par)
# Gradient of negative log-likelihood
gradient_neg_norm <- function(x, par) {

  n <- length(x)
  mu <- par[1]
  sigma <- par[2]
  gradient <- c((sum(x)-n*mu)/(sigma^2),  # To respect to mu
                (sum((x-mu)^2)/sigma^2-n)/sigma)  # To respect to sigma
  return(-gradient)

}
# New numeric estimates of the parameters
nllik_cg <- optim(par = c(0, 1), neg_log_lik_norm, x = data, method = "CG")
nllik_bfgs <- optim(par = c(0, 1), neg_log_lik_norm, x = data, method = "BFGS")
grad_cg <- optim(par = c(0, 1), neg_log_lik_norm, gr = gradient_neg_norm,
                 x = data, method = "CG")
grad_bfgs <- optim(par = c(0, 1), neg_log_lik_norm, gr = gradient_neg_norm,
                   x = data, method = "BFGS")


library(kableExtra)
df_tab <- data.frame(Algorithm = c("CG", "CG", "BFGS", "BFGS"),
                     Gradient = rep(c("Non specified", "Specified"), 2),
                     Mu = c(nllik_cg$par[1], grad_cg$par[1],
                            nllik_bfgs$par[1], grad_bfgs$par[1]),
                     Sigma = c(nllik_cg$par[2], grad_cg$par[2],
                               nllik_bfgs$par[2], grad_bfgs$par[2]),
                     LogLik = c(nllik_cg$value, grad_cg$value,
                                nllik_bfgs$value, grad_bfgs$value),
                     Fun = c(nllik_cg$counts[1], grad_cg$counts[1],
                             nllik_bfgs$counts[1], grad_bfgs$counts[1]),
                     Grad = c(nllik_cg$counts[2], grad_cg$counts[2],
                              nllik_bfgs$counts[2], grad_bfgs$counts[2]))

kable(df_tab, "latex", booktabs = T, align = "c", escape = F,
      col.names = c("Algorithm", "Gradient", "$\\hat\\mu$", "$\\hat\\sigma$",
                    "$-l(\\mu, \\sigma; x)$",
                    linebreak("Function\nevaluations", "c"),
                    linebreak("Gradient\nevaluations", "c")),
      caption = "Comparison table of the different algorithms.") %>%
  collapse_rows(columns = 1, valign = "top") %>%
  kable_styling(latex_options = "hold_position", font_size = 8)


# Code for the in-line results (Q2):

# ML estimates
```

```r
round(mean(data), 5) # mu
round(var(data)*((n-1)/n), 5) # var
```