Computational statistics - Lab 2

Alexander Karlsson (aleka769) Lennart Schilling(lensc874)

2019-02-05

Contents

Question 1: Optimizing a model parameter

Import data

The data will be imported. The variable `LMR`, calculated as the natural logarithm of `Rate`, will be added to the data. Then, it will be split into train (50%) and test set (50%).

```
# importing data
data = read.csv("mortality_rate.csv", sep = ";", dec = ",")
# adding variable 'LMR'
data$LMR = log(data$Rate)
# splitting data in train/test set (50:50)
n = dim(data)[1]
set.seed(123456)
id = sample(1:n, floor(n*0.5))
train = data[id,]
test = data[-id,]
```

Define `myMSE` function

The `myMSE`-function will be created.

```
myMSE = function(lambda, pars, print = FALSE, iterationCounterID = NULL) {
  # checking input
  if(!is.numeric(lambda)) stop("'lambda' has to be numeric.")
  if(!is.list(pars)) stop("'pars' is not a list.")
  if(!all("X" %in% names(pars),
          "Y" %in% names(pars),
          "Xtest" %in% names(pars),
          "Ytest" %in% names(pars))) stop("'pars' does not include all correct elements.")
  # counting iterations
  if(is.numeric(iterationCounterID)) {
    if(!exists(paste0("nIterations", iterationCounterID))) {
      assign(x = paste0("nIterations", iterationCounterID),
             value = 1,
             envir = globalenv()
      )
    } else {
      currentNumber = get(paste0("nIterations", iterationCounterID))
      assign(x = paste0("nIterations", iterationCounterID),
             value = currentNumber + 1,
             envir = globalenv()
      )
    }
  }
  # fitting loess model
  loessModel = loess(unlist(pars$Y) ~ unlist(pars$X),
                     enp.target = lambda)
  # predicting for Xtest
  yPred = predict(loessModel, unlist(pars$Xtest))
  # calculating predictive MSE
  mse = mean((unlist(pars$Ytest) - yPred)^2)
  # printing and returning predictive mse
```

```
  if(print) print(mse)
  return(mse)
}
```

Function call

Since the myMSE-function requires a list pars as an input, this list will be created first.

```
# creating pars
pars = list(X = train$Day,
            Y = train$LMR,
            Xtest = test$Day,
            Ytest = test$LMR)
```

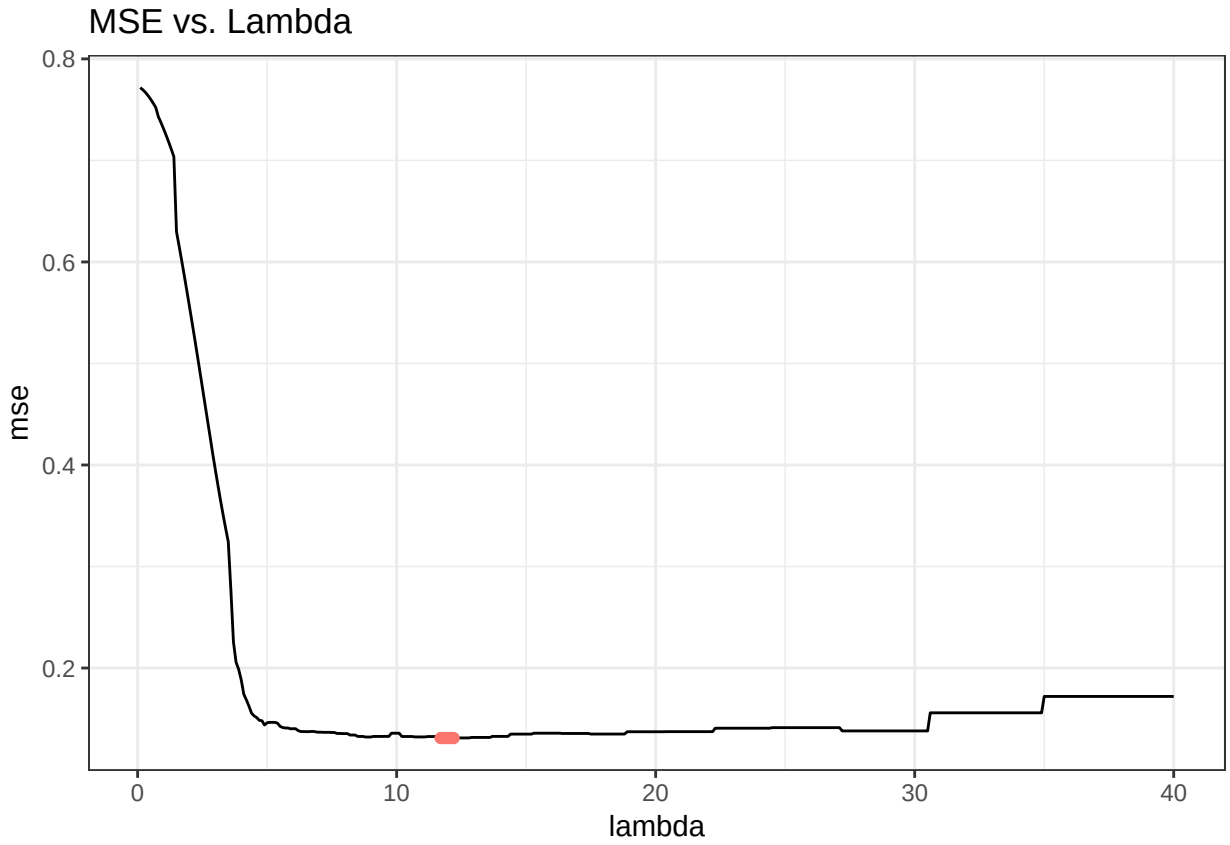
Looping over each lambda value from 0.1 to 40 in 0.1-steps, the mse generated by the myMSE-function using pars as an input parameter will be stored in the vector mseCollection.

```
# generating mse for the different lambdas
mseCollection = c()
for(lambda in seq(from = 0.1, to = 40, by = 0.1)) {
  mseCollection = c(mseCollection,
                    myMSE(lambda = lambda,
                          pars = pars))
}
```

Plot predictive MSE as function of lambda

The different mse are plotted.

```
# creating plotData
plotData = as.data.frame(cbind(lambda = seq(from = 0.1, to = 40, by = 0.1),
                               mse = mseCollection))
# identifying optimal lambda
optLambda = plotData[which(plotData$mse == min(plotData$mse)), ]
# plotting
library(ggplot2)
ggplot() +
  geom_line(data = plotData,
            aes(x = lambda,
                y = mse)) +
  geom_point(data = optLambda,
             aes(x = lambda,
                 y = mse,
                 color = "red"),
             show.legend = FALSE) +
  theme(legend.position = "none") +
  labs(title = "MSE vs. Lambda") +
  theme_bw()
```



The plot shows that there are multiple lambdas with the same minimum mse with a value close to 11 (marked by red dots).

```
knitr::kable(optLambda,
             row.names = FALSE,
             caption = "optimal lambdas and their mse")
```

Table 1: optimal lambdas and their mse

lambda	mse
11.7	0.131047
11.8	0.131047
11.9	0.131047
12.0	0.131047
12.1	0.131047
12.2	0.131047

By printing the lambdas with the minimum mse, the lambdas with values *11.7*, *11.8*, *11.9*, *12.0*, *12.1* and *12.2* are identified as the optimal lambdas.

To identify the optimal lambdas, the function `myMSE` had to be evaluated

```
length(seq(from = 0.1, to = 40, by = 0.1))
```

```
## [1] 400
```

times.

Optimization using golden search metgod

```
optimize(f = myMSE,
         pars = pars,
         iterationCounterID = 1,
         interval = c(0.1,40),
         tol = 0.01)
```

```
## $minimum
## [1] 10.69361
##
## $objective
## [1] 0.1321441
```

```
paste0("number of evaluations: ", nIterations1)
```

```
## [1] "number of evaluations: 18"
```

The `optimize`-function identified *lambda = 10.69361* as the optimal lambda which does not represent the true optimal lambda. The reason for this relatively big difference between the identified lambda (*10.69361*) and the optimal lambdas (*11.7*, *11.8*, *11.9*, *12.0*, *12.1*) is the small difference between the function values. The identified lambda has a function value of

```
fIdentifiedLambda = myMSE(lambda = 10.69361,
                          pars = pars)
fIdentifiedLambda
```

```
## [1] 0.1321441
```

which is almost equal to the function value of the true optimal lambdas.
In total, identified by counting the number of iterations, 18 evaluations of the `myMSE`-function were required which is much less than within the loop-procedure from step 3.

Optimize using BFGS method

```
optim(par = 35,
      fn = myMSE,
      pars = pars,
      iterationCounterID = 2,
      method = "BFGS")
```

```
## $par
## [1] 35
##
## $value
## [1] 0.1719996
##
## $counts
## function gradient
##        1        1
##
## $convergence
## [1] 0
##
## $message
```

```
## NULL
```

```
paste0("number of evaluations: ", nIterations2)
```

```
## [1] "number of evaluations: 3"
```

In this case, in total three evaluations were necessary to identify *lambda = 35* as the optimal lambda. It is very different from the optimal lambda. The golden search conducted in step 5 lead to a much better result. The reason for that is the chosen starting point. Since the *BFGS* algorithm stops iterating if the function values of points close the current point do not differ enough in combination with fact that *lambda = 35* represents a local stationary point in the function, the algorithm returns the same lambda as the identfied minimum.

Using another starting point, e.g. *lambda = 3* as in step 1.4, it will lead to a better result:

```
optim(par = 3,
      fn = myMSE,
      pars = pars,
      iterationCounterID = 3,
      method = "BFGS")
```

```
## $par
## [1] 6.213345
##
## $value
## [1] 0.1383069
##
## $counts
## function gradient
##        3        3
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
paste0("number of evaluations: ", nIterations3)
```

```
## [1] "number of evaluations: 9"
```

In this setting, after nine evaluations, an optimal lambda of 6.21 is identified. However, the golden section search still returns the better result.

Question 2

Load data

The loaded data is simply a vector of numerical values with length of 100 elements.

```
load("data.Rdata")
x = data
```

Maximum likelihood derivation

The maximum likelihood derivation of a normal distributed variable can be done the following way:

1. Define pdf for Nf.
2. Likelihood; take the product of the pdf from i = 1:n (sample size).
3. Log-likelihood; log of the product. Makes derivation easier.
4. Set partial derivatives of μ and σ to zero respectively and calculate.

Pdf for N(μ, σ^2):

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x_i-\mu)^2}$$

Likelihood function:

$$l(\mu,\sigma) = \prod_{i=1}^{n} f(x) = \frac{1}{(\sqrt{2\pi\sigma^2})^n} e^{\frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2} = (2\pi\sigma^2)^{\frac{-n}{2}} e^{\frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2}$$

Log-likelihood function:

$$\begin{aligned} L(\mu,\sigma) = ln[l(\mu,\sigma)] &= \frac{-n}{2} ln(2\pi\sigma^2) + (-)\frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2 \cdot ln(e) = \\ &= \frac{-n}{2} ln(n\pi\sigma^2) - \frac{\sum_{i=1}^{n}(x_i-\mu)^2}{2\sigma^2} \propto \quad (*) \\ &\propto \frac{-n}{2} ln(\sigma^2) - \frac{\sum_{i=1}^{n}(x_i-\mu)^2}{2\sigma^2} \end{aligned}$$

$(*)$: The constants 2π can be removed since $ln(2\pi\sigma^2)$ can be rewritten as $ln(2\pi) + ln(\sigma^2)$, and in such case the derivative of the first logarithm expression is 0 w.r.t both μ and σ and can be removed. However, this only works when the analytical approach is used. When optimizing, all constants need to be included.

Partial derivative, w.r.t μ:

$$\begin{aligned} \frac{\partial L(\mu,\sigma)}{\partial\mu} = 0 \Rightarrow 0 - \frac{1}{2\sigma^2}2(-1)\sum_{i=1}^{n}(x_i-\mu) = 0 \Rightarrow \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i-\mu) = 0 \quad (**) \\ \Rightarrow \sum_{i=1}^{n} x_i - n\mu = 0 \Rightarrow \mu = \frac{\sum_{i=1}^{n} x_i}{n} \Rightarrow \hat{\mu} = \bar{x} \end{aligned}$$

$(**)$: The expression is also 0 if $\sigma = 0$, but emphasis is on μ. Again, the removed term (σ^{-2}) needs to be included when optimizing but not in the analytical approach.

Partial derivative, w.r.t σ:

$$\frac{\partial L(\mu,\sigma)}{\partial \sigma} = 0 \Rightarrow \frac{-n}{2}\cdot\frac{2}{\sigma} - \frac{1}{2}\cdot(-2)\cdot\frac{1}{\sigma^3}\sum_{i=1}^{n}(x_i-\mu)^2 = 0 \Rightarrow$$
$$\Rightarrow \frac{-n}{\sigma} + \frac{\sum_{i=1}^{n}(x_i-\mu)^2}{\sigma^3} = 0 \Rightarrow n = \frac{\sum_{i=1}^{n}(x_i-\mu)^2}{\sigma^2} \Rightarrow$$
$$\Rightarrow \hat{\sigma} = \sqrt{\frac{\sum_{i=1}^{n}(x_i-\mu)^2}{n}}$$

Minus log-likelihood

The minus log-likelihood is simply the negative of the likelihood function. It is usually nice to optimize the log-likelihood instead of the likelihood analytically because the logged distribution is easier to calculate. On top of that, the likelihood product is a large (n = 100) series of small values, prone to computational errors. The logged likelihood does however result in negative values, and taking the negative of them inverts it to a positive sign.

$$-L(\mu,\sigma) = -\left(\frac{-n}{2}ln(2\pi\sigma^2) + (-)\frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2\cdot ln(e)\right) =$$
$$= \frac{1}{2}\left(nln(2\pi\sigma^2) + \sigma^{-2}\sum_{i=1}^{n}(x_i-\mu)^2\right)$$

To optimize this function, it needs to be implemented first. Below is R code for the equation above.

```
# minus log-likelihood function. Pass input as one vector to 'fool'
# optim that we're optimizing a function of one variable rather than
# a function of two variables:
minus_log_like = function(pars){
  return((1/2) * (length(x)*log(2*pi*pars[[2]]^2) + (1/pars[[2]]^2) * sum( (x - pars[[1]])^2 )) )
}

# check output:
minus_log_like(pars = c(0,1))
```

```
## [1] 374.4395
```

```
# output should be lower than c(0,1) since the analytical
# optimum of the normal distribution has been derived...
minus_log_like(pars = c(mean(x), sd(x)))
```

```
## [1] 211.5095
```

```
# should return the same result as above...
my_mu    = sum(x) / length(x)
my_sigma = sqrt(sum((x - my_mu)^2) / length(x))
minus_log_like(pars = c(my_mu,
                        my_sigma))
```

```
## [1] 211.5069
```

The function output (minus log-likelihood) is minimized when the `pars` argument is set to `mean(x)` and `sd(x)` respectively, which is the corresponding built-in functions for the derivations from before... There is a slight difference in `minus_log_like` calculations between the built-in functions and the formulas for `my_sigma` and `my_mu` that is due to computational errors.

The gradient function for optimization is defined below. It is the negative derived formulas for $\frac{\partial L}{\partial \mu}$ and $\frac{\partial L}{\partial \sigma}$ from before.

```
# gradient function for optim call(s). Important to pass both mu and sigma
# in one vector to 'fool' optim that we're optimizing one param instead of 2!
grad_fun = function(pars){
  grad_mu    = -sum(x - pars[[1]]) / pars[[2]]^2
  grad_sigma = (length(x)) - (sum((x - pars[[1]])^2)/pars[[2]]^2)
  return(c(grad_mu, grad_sigma))
}
```

Check of optim results

With the above functions defined, it is possible to see whether or not the optimization algorithms can converge to the true minimum of $\mu = 1.2755276$ and $\sigma = 2.0160822$.

```
library(dplyr)
# optimization results:
optims = mapply(FUN = function(x, f){
  r = optim(par = c(0,1),
            fn = minus_log_like,
            gr = f,
            method = x)
  data.frame(
    "mu"              = r[["par"]][1] %>% round(5),
    "sigma"           = r[["par"]][2] %>% round(5),
    "value"           = r[["value"]][1] %>% round(5),
    "counts_function" = r[["counts"]][[1]],
    "counts_gradient" = r[["counts"]][[2]],
    "convergence"     = r[["convergence"]][[1]] == 0)
  },
  x = list("BFGS", "BFGS", "CG", "CG"),
  f = list(grad_fun, NULL, grad_fun, NULL),
  SIMPLIFY = TRUE, USE.NAMES = T)

colnames(optims) = c("BFGS w. gradient", "BFGS w. NULL-gradient",
                     "CG w. gradient", "CG w. NULL-gradient")
knitr::kable(optims, align = 'c')
```

	BFGS w. gradient	BFGS w. NULL-gradient	CG w. gradient	CG w. NULL-gradient
mu	1.27553	1.27553	1.27553	1.27553
sigma	2.00598	2.00598	2.00598	2.00598
value	211.50695	211.50695	211.50695	211.50695
counts_function	41	37	153	297
counts_gradient	15	15	45	45
convergence	TRUE	TRUE	TRUE	TRUE

In all cases, the algorithms converged. Also, all algorithms resulted in the same parameters. The main difference lies in the iterations, where the 'BFGS' algorithm had the fewest function- and gradient evaluations. Therefore, this method is considered the best one.

Appendix

All code is shown in the text, no need for an appendix!