Lab2

Andreas Stasinakis

January 24, 2019

Contents

Task 1: Optimizing a model parameter

1.1 Import CSV data and split them

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

```
# import the data
mortal = read.csv2("../mortality_rate.csv")
mortal$LMR = log(mortal$Rate)

# spliting the data into training and test
n = dim(mortal)[1]
set.seed(123456)
id = sample(1:n,floor(n*0.5))
train = mortal[id,]
test = mortal[-id,]
```

1.2 Function MSE

Write your own function myMSE() that for given parameters λ and list pars containing vectors $X, Y, Xtest, Ytest$ 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. The predictive MSE is the mean square error of the prediction on the testing data. It is defined by the following Equation (for you to implement):

$$predictiveMSE = \frac{1}{length(test)} \sum_{ithelementinthetestset} (Y_{test[i]} - fYpred(X[i]))^2$$

with fYpred(X[i]) is the predicted value of Y if X is $X[i]$. Read R's functions for the prediction so that you do not have to implement it yourself.

```
myMSE = function(lambda, pars){

  # fia a loess model with penalty factor lambda
  model = loess(formula = Y ~ X,data = pars,enp.target = lambda )

  #predict with the model
  pred_loess = predict(object = model, pars$Xtest)

  #compute the MSE
  MSE = (sum((pars$Ytest - pred_loess)^2))/length(pars$Ytest)

   #count the number of iterations
  iter <<- iter +1

  #return MSE
  return(MSE)
}
```

1.3 Use the function myMSE

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: $\lambda = 0.1, 0.2, ..., 40$

```
#vector to store all MSE's
all_MSE = c()
iter = 0

# a for loop to estimate MSE for all lambda's

k = 1
my_list = list(X = train$Day, Y = train$LMR, Xtest = test$Day, Ytest = test$LMR)

for (i in seq(0.1,40,by = 0.1)) {



  #use myMSE function
  all_MSE[k] = myMSE(lambda = i,pars = my_list )

  k = k +1


}
```

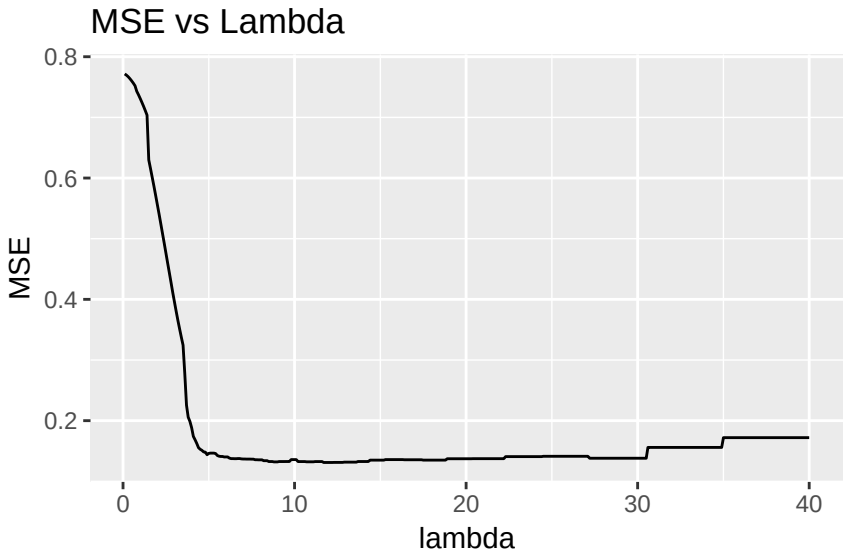
1.4 Plot MSE vs lambda

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?

```
library(ggplot2)

df_1 = data.frame(all_MSE,lambda = seq(0.1,40,0.1))

ggplot(df_1) +
  geom_line(mapping = aes(x = df_1$lambda,y = df_1$all_MSE)) +
  labs(title = "MSE vs Lambda", x = "lambda", y = "MSE")
```



```r
# minimum MSE
best_MSE = df_1$all_MSE[which.min(df_1$all_MSE)]

# lambda for the minimum MSE

best_lambda = df_1$lambda[which.min(df_1$all_MSE)]

# number of iterations
iter = which.min(df_1$all_MSE)
```

In this task we plot MSE values, which have been calculated by function myMSE, vs the λ. We can observe that there are 3 trends in the plot. The first one, from $\lambda = 0.1$ the $MSEerror$ reduces exponentially until $\lambda = 6$. After that the MSE remains almost the same with same not so important changes before starting increasing again after $lambda = 30$. The lowest MSE is 0.131047 for $\lambda = 11.7$ and this result came up after 117 iterations of the function myMLE.

1.5 Using optimize function

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.

```r
set.seed(12345)
iter = 0
opt_f = optimize(f = myMSE,interval = c(0.1,40), tol = 0.01, pars = my_list)

# Optimal MSE using the function optimize
opti_MSE = opt_f$objective

# minimum Lambda for MSE
opti_lambda = opt_f$minimum

# iterations for finding minimum MSE
iter
```

```
## [1] 18
```

In this task we try to find the lowest MSE using the function optimize. The MSE which optimize function found is 0.1321441 which is close to the optimal one, but not the lowest. The λ for this MSE value is 10.6936107 which is lower that the one above. Finally the number of iterations is 18 which is much lower than the 117 iterations we have in the previous task.

If we try to make conclusions between the two results we can say that the optimize function does a really good job in order to find the optimal MSE because the difference is not that important. The reason why someone should prefer optimize function is that the number of iterations is really low which means that this method is not computational heavy in contrast to the first task we run the function for 400 iterations. Finally the reason why optimize function could not find the exact optimal MSE value is because after $\lambda = 10$ the differences between the MSE error are really low so probably the algorithm reached a local minimum and stop there.

1.6 Using optim() function and compare with optimize function

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.

```
res_BFGS1 = optim(par = 35,fn = myMSE, method = "BFGS", pars = my_list)

# optimal MSE from optim function
optim_MSE = res_BFGS1$value

#lambda for the MSE
optim_lambda = res_BFGS1$par

# number of iterations
bfgs_iter = res_BFGS1$counts
```

In this task we implement the function optim in order to find the optimal MSE error. The optimal error obtained is 0.1719996 for $\lambda = 35$ after only one iteration of the algorithm. Compared to the other two tasks is the highest error and there is a really simple explanation for this. We can see from the plot above that $\lambda = 35$, which is the starting point for this algorithm, occurs in a flat line. That means what while the algorithm calculates the derivative in order to know which direction to choose, it is "trapped" in the flat line and stops there. So after one iteration, the optim() function thinks that the optimal MSE is 'r optim_MSE. This case a really good example to show us that the starting point, here is λ, is very important for the procedure because if we choose a wrong value we may "trapped" in local minimum or flat lines.

Finally, it is obvious that the third approach is the worst one because the MSE is much higher than the other two. Between the first and the second task, i think that the second one is better. Although the MSE is a little higher, the number of iterations is really low and this makes the algorithm less complex.

Task 2 : Maximizing likelihood

The file data.RData contains a sample from normal distribution with some parameters μ, σ. For this question read ?optim in detail.

2.1 Loading data

Load the data

```
# Use function load to load the data
load("data.Rdata")
```

2.2 Normal log -likelihood function, partial derivatives and parameter estimates

Write down the log-likelihood function for 100 observations and derive maximum likelihood estimators for μ , σ analytically by setting partial derivatives to zero. Use the derived formulae to obtain parameter estimates for the loaded data.

```
# implement the log - likelihood function

mylog_like = function(data,param ){

  #length of the data
  n = length(data)
  mu = param[1]
  sigma = param[2]

  # Log -likelihood for the Normal distribution

  likelihood = -(n/2)*log(2*pi) - (n/2)*log(sigma^2) - (sum((data - mu)^2))/(2*sigma^2)
```

```
  return(-likelihood)
}

# Use the formulas from the derivatives of Log likelihood to estimate
# parameters $\mu$ and $\sigma$


n = length(data)
mu = mean(data)
sigma = sqrt(sum((data - mu)^2)/n)

# function calculates the derivatives
# with respect to $\mu$, $\sigma$

gradient = function(data,param){

  # take as input the parameters $\mu$ and #\sigma$

  mu = param[1]
  sigma = param[2]
  n = length(data)

  #formula for the partial derivative for $\mu$
  grad_mu = (sum(data) - n*mu)/(sigma^2)

  #formula for the partial derivative for $\sigma$

  grad_sigma = (sum((data-mu)^2) - sigma^2*n)/(sigma^3)

  #return a vector for the parameters
  grad = c(grad_mu,grad_sigma)

  return(-grad)

}
```

In this task we implement a function in order to calculate the log - likelihood for a sample from normal distribution. The function will return the log-likelihood given the parameters μ and σ. The formula used for this function, after some mathematical procedure, is :

$$-\frac{n}{2}ln(2\pi) - (n/2)ln(\sigma^2) - \frac{1}{2\sigma^2}\sum_{j=1}^{n}(x_j - \mu)^2$$

We also create one function which return the gradient of the parameters. In order to prove the formulas we calculate the partial derivative with respect to each parameter. For the variance, after calculations, we have the formula bellow :

$$(\sum_{j=1}^{n}(x_j - \mu)^2 - n\sigma^2)/\sigma^3$$

and the derivative with respect to the mean is :

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

After that we set the partial derivatives equal to zero. With this way we have two equations in order to estimate the two parameters. Finally, after some calculations, we have the above formulas for the parameter estimation :

$$\hat{\mu_n} = \frac{1}{n}\sum_{j=1}^{n} x_j$$

, which is obviously the mean of our sample and

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

, which is the variance of the sample.

Finally, the obtained parameter estimates for the loaded data are $\mu = 1.2755276$ and $\sigma = 2.0059765$.

2.3 Optimize minus log-likelihood(CG, BFGS methods with and without parameters)

Optimize the minus log-likelihood function with initial parameters $\mu = 0$, $\sigma = 1$. Try both Conjugate Gradient method (described in the presentation handout) and BFGS (discussed in the lecture) algorithm with gradient specified and without. Why it is a bad idea to maximize likelihood rather than maximizing log-likelihood?

```
# parameters for the log likelihood
# $\sigma = 1$ and $\mu = 0$
pr = c(0,1)

# Conjugate Gradient method  without specifying the gradient function

res_CG1 = optim(par = pr,fn = mylog_like, method = "CG", data = data)

# Conjugate Gradient method  with specifying the gradient function
res_CG2 = optim(par = pr,fn = mylog_like, gr = gradient ,
                method = "CG", data = data)

# BFGS method without specifying the gradient function
res_BFGS1 = optim(par = pr,fn = mylog_like, method = "BFGS", data = data)

# BFGS method with specifying the gradient function
res_BFGS2 = optim(par = pr,fn = mylog_like,
                  method = "BFGS", data = data,gr = gradient)
```

In this task we optimize the minus log - likelihood with initial parameters $\mu = 0$ and $\sigma = 1$ for a sample of 100 normally distributed observations.

In most of the cases we prefer to maximize log - likelihood instead of just the likelihood function. We can do that because the logarithm is monotonically increasing function, so we do not change the results using it.

There are two important reasons why we prefer maximizing the log - likelihood instead of just the likelihood function. First of all is really convenient to work with the log because the user can take advantage of the properties of the logarithm. That really simplifies all mathematical implementations we have to do. Another reason is that it is helpful from the computer's perspective. For example, the total likelihood is the product of the likelihoods for each point(assuming that the dataset is i.i.d). This may cause underflow because we calculate the product of small numbers(probabilities). Using the logarithm and its properties instead, we can use the sum instead of the product in order to avoid this problem.

2.4 Compare CG, BFGS and analysis

Did the algorithms converge in all cases? What were the optimal values of parameters and how many function and gradient evaluations were required for algorithms to converge? Which settings would you recommend?

```
# create a data frame to store all the functions

cg1 = c(res_CG1$par,res_CG1$counts[1],res_CG1$counts[2])
cg2 = c(res_CG2$par,res_CG2$counts[1],res_CG2$counts[2])
bfgs1 = c(res_BFGS1$par,res_BFGS1$counts[1],res_BFGS1$counts[2])
bfgs2 = c(res_BFGS2$par,res_BFGS2$counts[1],res_BFGS2$counts[2])

df = data.frame(t(matrix(c(cg1,cg2,bfgs1,bfgs2),nrow = 4,ncol = 4)))

colnames(df) = c("mean", "standar_deviation", "fun_eval", "grad_eval")
rownames(df) = c("CG","CG_gra", "BFGS", "BFGS_grad")

knitr::kable(x = df, caption = "Comparing all the algorithms")
```

Table 1: Comparing all the algorithms

	mean	standar_deviation	fun_eval	grad_eval
CG	1.275528	2.005977	210	35
CG_gra	1.275528	2.005977	56	17
BFGS	1.275527	2.005977	37	15
BFGS_grad	1.275527	2.005977	38	15

In the previous task we try different algorithms in order to optimize the minus log - likelihood for the parameters μ, σ. In the above table we can take information about all the different algorithms, with and without specifying the gradient function. In all four cases the algorithms converge, and the results are almost the same for all cases. The differences between each algorithm is the number of evaluations for the function and the gradient. So in order to choose the best one we have to compare those numbers. It can be said that in both cases, with of without the gradient, *CG* algorithm is not the best choice because the number of iterations is really high(56 and 210). This makes the algorithm slow and computational expensive, so it is better to avoid using that. For the *BFGS* algorithm, it is obvious that it does not makes such a big difference if we specify or not the gradient function.